



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSI NYELVEK ÉS FORDÍTÓPROGRAMOK

TANSZÉK

Táblázatkezelő szoftver implementálása Haskell nyelven

Témavezető:

Dr. Kaposi Ambrus

egyetemi docens

Szerző:

Széles Márk

programtervező informatikus BSc

Budapest, 2021

Az eredeti szakdolgozati / diplomamunka témabejelentő helye.

Tartalomjegyzék

1. Bevezetés	3
2. Felhasználói dokumentáció	5
2.1. Rendszerkövetelmények	5
2.2. Telepítés és indítás	5
2.3. A felhasználói felület áttekintése	6
2.4. Cellák és kiértékelés	7
2.4.1. A kifejezések	7
2.4.2. A futásidejű reprezentáció	8
2.4.3. Lehetséges hibák	9
2.5. A standard könyvtár	10
2.5.1. Kombinátorok	10
2.5.2. Alapértelmezett függvénypéldányok	11
2.5.3. A könyvtár bővítése	11
2.6. A parancssorban használható parancsok	12
2.6.1. Kifejezés kiértékelése GHCi-ban	12
2.6.2. Cellatartalom másolása és áthelyezése	12
3. Fejlesztői dokumentáció	14
3.1. A fejlesztői dokumentáció felépítése	14
3.2. A szoftver felépítése	14
3.2.1. Felhasznált technológiák összefoglalása	14
3.2.2. A globális állapot	15
3.2.3. A GUI	17
3.2.4. Programkomponensek és modulszerkezet	18
3.3. Spreadsheet	20

3.3.1.	Spreadsheet.Types	20
3.3.2.	Spreadsheet.Parser	22
3.3.3.	Spreadsheet.Interface	25
3.4.	Eval	27
3.4.1.	Eval.CodeGeneration	27
3.4.2.	Eval.EvalMain	30
3.4.3.	Eval.Ghci	31
3.4.4.	Eval.CommandLine	32
3.5.	Persistence	33
3.6.	App	34
3.6.1.	App.RunApp	34
3.6.2.	App.CreateEnv	34
3.6.3.	App.Setup	35
3.6.4.	App.Setup.Global	36
3.6.5.	App.Setup.CommandLine	38
3.6.6.	App.Setup.Editor	38
3.6.7.	App.Setup.Menubar	38
3.6.8.	App.Setup.Table	39
3.7.	Tesztelés	39
3.7.1.	A tesztelés folyamata	39
3.7.2.	Test.Spreadsheet.*	40
3.7.3.	Test.Eval.CodeGeneration	41
3.7.4.	Felhasználói tesztek	42
3.7.5.	A tesztelés tanulságai	45
4.	Összegzés	46
4.0.1.	Ami megvalósult	46
4.0.2.	Távlati tervek	46
	Irodalomjegyzék	48
	Táblázatjegyzék	48
	Forráskódjegyzék	49

1. fejezet

Bevezetés

Ha indokolni szeretnénk egy új táblázatkezelő szoftver elkészítésének létjogosultságát, két kérdésre kell választ adnunk:

1. Milyen funkciókat kell ellátnia egy táblázatkezelő szoftvernek?
2. Mi az, ami hiányzik a jelenleg elterjedt szoftverekből? (Pl. Microsoft Excel)

Az első kérdésre talán az a legegyszerűbb válasz, hogy egy táblázatkezelő lehetőséget ad adatok tárolására és a bevitt adataink alapján újabb adatok kiszámítására. Ez a valóságban számtalan alkalmazási lehetőséget jelent. Az Excel-lel például lehet színes, táblázatos formájú órarendet készíteni, egy gyakorlati csoport eredményeit számontartani, családi költségvetést vezetni, stb.

Egy táblázatkezelőben minden cella tartalma egy funkcionális program. Egy cellába írhatunk egy egyszerű kifejezést (adat), vagy egy összetettebb programot, ami korábbi adatok függvényében számít ki egy új adatot. A táblázatkezelő tehát nem más, mint egy könnyen használható interfész a háttérben meghúzódó funkcionális nyelvhez. Ennek a nyelvnek az intuitív használatát számos funkció segíti. Lehetővé válik az összetett program komponensekre bontása, és az egyes komponensek eredményeinek hatékony vizualizációja.

Ha tekintjük napjaink legnépszerűbb táblázatkezelő szoftverét, az Excel-t, azt láthatjuk, hogy a fent leírt feladatot kiválóan ellátja. Bővelkedik megjelenítéssel kapcsolatos opciókban, a felhasználói felület használata intuitív, az elérhető dokumentáció közérthető. Fő hiányosságát nem is ebben látom, hanem az általa használt programozási nyelvben. Az Excel-ben a szoftver saját programozási nyelvét használhatjuk, az eszköztár bővítésére viszont a beágyazott VBA programnyelv használata-

tával van lehetőség. Így az összetettebb, egyedi funkciók megvalósításához egyszerre két programozási nyelvet kell használni, sőt két programozási paradigmát is: a cellákba funkcionális kód kerül, a VBA viszont csak korlátozottan támogatja a funkcionális paradigmát.

Ezen probléma megoldására tesztek kísérletet dolgozatomban. Egy olyan táblázatkezelő szoftvert készítettem el, aminek a celláiba – a táblázatkezelői funkciók megfelelő ellátása érdekében kissé kiegészített – Haskell nyelven lehet programokat írni. Így közvetlenül a felhasználó rendelkezésére áll egy általános célú, tisztán funkcionális programnyelv teljes eszköztára.

2. fejezet

Felhasználói dokumentáció

2.1. Rendszerkövetelmények

A program futtatásához szükséges, hogy a *pgrep* program telepítve legyen, és a helye hozzá legyen adva a *PATH*-hoz. Ez Ubuntu 20.04 LTS operációs rendszer esetén alapértelmezett.

2.2. Telepítés és indítás

A program fordításához az alábbiakra van szükség:

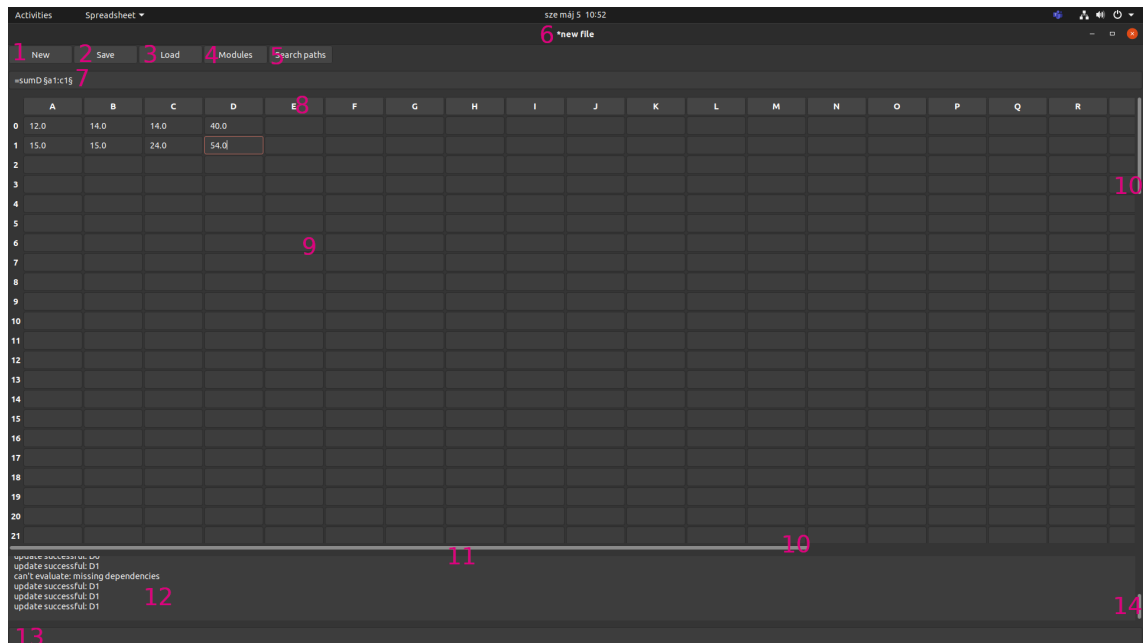
1. GHC, legalább 8.6.5 verzió (valószínűleg korábbiakra is működik)
2. Haskell csomagok: *base*, *cereal*, *fgl*, *ghcid*, *gtk2hs*, *microlens-platform*, *parsec*, *split*, *unliftio*, illetve ezen csomagok függőségei.

A fentiek megléte esetén a program az alábbi paranccsal fordítható:

```
ghc Main.hs -o insertNameHere
```

insertNameHere helyére írható a futtatható állomány kívánt neve. Az alkalmazás az így kapott futtatható állomány futtatásával indítható el. Fontos, hogy az *Empty.hs* fájl a futtatható állománnyal egy mappában helyezkedjék el, hogy a kiértékelés az ezen dokumentációban leírt módon működjék. **EZ AM FULL GÁZ, KONFIGURÁLHATÓVÁ KÉNE TENNI.**

2.3. A felhasználói felület áttekintése



2.1. ábra. A felhasználói felület

1. Ezzel a gombbal lehet új számológépet létrehozni. Ekkor a számológép üres lesz. Billentyűkombináció: Alt-N.
2. Ezzel a gombbal lehet elmenteni a számológép aktuális állapotát. A fájl az alkalmazás saját *.fsandor* kiterjesztésében kerül mentésre. Ha a tábla még nem volt mentve, felugrik egy dialógus, ahol meg lehet adni a mentés útvonalát. Ha már be van töltve egy fájl, a mentés frissíti a mentett fájl tartalmát. Billentyűkombináció: Alt-S.
3. Ezzel a gombbal lehet fájlt betölteni. Billentyűkombináció: Alt-L.
4. Ezzel a menüponttal lehet beállítani a háttérben futó GHCi-be betöltendő modulok listáját. A felugró szövegmező minden sora egy modult jelent. **KELL ÁBRA?**
5. Ezzel a menüponttal lehet beállítani, hogy az alapértelmezett útvonalakon kívül hol keresse a GHCi a betöltendő modulokat. Minden sor egy elérési útvonalat jelent. Az elérési útvonal lehet abszolút (ez utóbbi a javasolt) vagy relatív a futtatható állomány helyéhez képest.
6. A betöltött fájl neve. Ha épp nincs elmentve a számológép, a "*new file" szöveg jelenik meg. Ha a betöltött fájl neve előtt szerepel egy "*", az azt

jelenti, hogy a legutóbbi mentés óta történt módosítás.

7. Kódszerkesztő. Ebbe a sorba lehet kódot írni az aktuálisan kijelölt cellához. A kijelölt cella az a cella, amelyre a felhasználó legutóbb kattintott. A beírt kód akkor kerül kiértékelésre, ha a felhasználó egy másik elemre kattint vagy leüti az entert.
8. Az oszlop tetején levő gombra kattintva lehetséges beállítani az oszlop karakterekben megadott szélességét. Az oszlopok legfeljebb 100 karakter szélesek lehetnek. Megjegyzés: az átméretezés végrehajtása jelenleg meglehetősen lassú, akár 10 másodpercet is igénybe vehet.
9. A számolótábla. A számolótábla mérete fixen 26 oszlop és 100 sor. A cellába beírt kód akkor kerül kiértékelésre, ha a felhasználó egy másik elemre kattint vagy leüti az entert.
10. A számolótábla alján és jobb oldalán található görgőkkel lehet görgetni a számolótáblában.
11. A számolótábla és a log ablak relatív mérete húzással állítható.
12. A log mutatja a végrehajtott akciók (pl. cella kódjának átírása, GHCi query eredménye) eredményét. Jobb oldalt görgethető.
13. Az alkalmazáshoz tartozó parancssor. A beírt parancs akkor kerül kiértékelésre, ha a felhasználó leüti az entert.
14. A log üzenetek között a log ablak jobb oldalán található görgővel lehet görgetni.

2.4. Cellák és kiértékelés

2.4.1. A kifejezések

Egy cella tartalma négyféle lehet: üres cella, szám, szöveg vagy formula. A cellába beírt kifejezés pontosan akkor formula, ha az első (nem szóköz) karaktere "=". Amennyiben az első (nem szóköz) karakter nem "=", és a beírt szöveg értelmezhető tizedestörtként, annyiban a cella tartalma számként kerül értelmezésre. Minden más esetben a cella tartalma egy string. Példák:

1. Szám: "0.12", "-11", "-12.", ".123"
2. Formula: "=sum [1..10]", "= §\$a\$0§+sumD §a2:b4§"

3. String: "alma", ".12aa"

A formulákba lehetséges cellahivatkozásokat írni. Kétféle cellahivatkozás létezik:

1. Egyszerű hivatkozás. Pl: §a0§,§B\$1§, §§c\$20§, §§B10§
2. Listahivatkozás. Pl: §a0:B4§, §c\$2:c\$10§, §§a0:b\$10§

A cellahivatkozások nem kisbetű-nagybetű érzékenyek.

A hivatkozásokban található § karakter minden hivatkozás (listahivatkozás esetén mindkét határ) oszlop- és sorneve elé is beszúrható. A § jelnek cellák másolásakor és mozgatásakor van szerepe (részletesebben lásd 2.6.2).

Az alkalmazás egyelőre nem támogatja számformátum beállítását, ennek megfelelően a belső reprezentáció által biztosított maximális számú tizedesjegy jelenik meg.

A formula összes többi része Haskell kódként kerül értelmezésre. A típushelyesség a kiértékelés során kerül ellenőrzésre.

Megjegyzés: A cellákba csak numerikus és szöveges adat írható és számítható. Amennyiben a megadott formula más típusú, úgy *Show* példánnyal rendelkező típusok esetén az érték *String* reprezentációja lesz a számítás eredménye, ellenkező esetben pedig hiba lép fel. Pl. az " $=[1..5]$ " formula az " $[1,2,3,4,5]$ " string-re értékelődik ki, az " $=\backslash x \rightarrow x$ " kifejezés eredménye hiba.

2.4.2. A futásidejű reprezentáció

Egy cella futásidejű reprezentációja egy *Maybe a* típusú érték. Az üres cella reprezentációja *Nothing*, a nemüres cella reprezentációja *Just val*, ahol *val* a cella értéke. A kiszámított értékek is mindig *Just*-ba csomagoltatnak, hogy a GHCi továbbszámolhasson velük. A valamely cella megváltozásának hatására kezdődő kiértékelési folyamat végrehajtása után az értékek visszaolvasásakor ez a háttérben egy *fromJust* hívással kicsomagolódik, az értékek a cellákban *Just* nélkül jelennek meg. Egész szám esetén a kódgeneráló algoritmus figyel arra, hogy olyan literált generáljon a cellaértékből, ami értelmezhető egész típusúként (azaz ilyenkor levágja a tizedesrészt).

A cellahivatkozások feloldását példákon keresztül mutatjuk be. Bal oldalt található a hivatkozás, jobb oldalt a generált kód.

1. $\S a0\S \rightarrow \text{fromJust } v0$

2. $\S a0:a3 \rightarrow [v0, v2, v5, v10]$

3. $\S a1:b0 \rightarrow []$

A cellahivatkozások feloldásakor a cellákhoz egy $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ bijekción keresztül egy azonosító kerül hozzárendelésre (innen származnak a kódban írt vi változónevek). A függvény definíciója megtalálható a fejlesztői dokumentációban. Listahivatkozás feloldásakor a hivatkozott téglalap elemei fentről lefelé és balról jobbra haladva, sorfolytonosan kerülnek a listába. Pl: $\S a2:c3$ esetén a lista elemeinek sorrendje: $[a2, a3, b2, b3, c2, c3]$

A fentiek alapján a típusú cellára mutató egyszerű hivatkozás ugyanúgy használható, mint egy a típusú érték. Az üres cellára való egyszerű hivatkozás hibát okoz. Példák (beírt kód és generált kód):

1. A0 kódja: `"12.0"`, B1 kódja: `"=\S a0\S+11"` \rightarrow B1 értékét kiszámító Haskell kód:
$$v3 = Just \$ fromJust (Just 12) + 11$$
2. A0 kódja : `" "` B1 kódja: `"=\S a0\S+11"` \rightarrow B1 értékét kiszámító (hibát eredményező) Haskell kód:
$$v3 = Just \$ fromJust Nothing + 11$$

A listahivatkozások feloldásakor azonban a lista elemei *Maybe* a típusúak maradnak. Példa:

A0 kódja: `"12"`, A1 kódja: `"13"`, A2 kódja: `=sum \S a0:a1\S` \rightarrow A2 értékét kiszámító (típushibás) Haskell kód:
$$v5 = Just \$ sum [Just 12, Just 13]$$

Ez látszólag komoly problémákat eredményez, de az \notin operátor (lásd 2.5.3) egy könnyen használható megoldást ad a problémára.

2.4.3. Lehetséges hibák

Egy cella kódjának megváltoztatása után előfordulhatnak hibák, az alábbiakban ezek foglaltatnak össze:

1. Ha egy cella kódját nem sikerül parseolni, a cellában az "FNoParse" szöveg jelenik meg. Parseolási hiba csak formula esetén léphet fel, ha szerepel a kódban ' \S ' karakter, de nem egy értelmes hivatkozás részeként. Példák: `"=\S a\S"`, `"=a1\S"`. Amennyiben a referenciák helyesek, de a Haskell kód mégsem értelmes, akkor az kiértékelési hibaként jelenik meg.
2. Ha a cella megváltoztatásával hivatkozási kör jönne létre a számolótáblában, a megváltoztatni kívánt cellában az "FCycleRefError" szöveg jelenik meg.

3. Ha a megváltoztatott cella olyan cellára hivatkozik, amely hibás (azaz nem nyerhető ki az értéke), akkor a cellában az "FNoCache" hibaüzenet jelenik meg.
4. Ha a kiértékelés során hiba történt (Haskell szintaxishiba/típushiba/futásidejű hiba), akkor a cellában az és leszármazott celláiban az "FGhciError" hibaüzenet jelenik meg.
5. Ha a cella kiértékelése egy másodpercnél tovább tartana, a kiértékelés leáll, és a cellában az "FTimeoutError" szöveg jelenik meg. A leszármazott cellákban az "FGhciError" szöveg jelenik meg. Megjegyzés: a végtelen GHCi számítások megszakítása nem működik teljesen megbízhatóan, így a legjobb tudatosan elkerülni az ilyen számítások futtatását. (A probléma részletes leírása megtalálható a 3.3.5 szakaszban.)

Megjegyzés: a hibaüzenetek az alkalmazás jelen verziójában nem tartalmazznak extra információt a fent leírtakhoz képest.

2.5. A standard könyvtár

2.5.1. Kombinátorok

A fentiekben már láttuk, hogy a típusú cellák listájának futásidejű reprezentációja egy $[Maybe\ a]$ típusú érték. Ez a megoldás teszi lehetővé, hogy a program kezelni tudja az üres cellákat. A probléma, ami ilyenkor felmerül, hogy egy $[a] \rightarrow b$ függvényt szeretnénk alkalmazni egy $[Maybe\ a]$ típusú értékre, valamilyen módon kezelve az üres cellákat.

Az első megoldás az üres cellák helyettesítése egy alapértelmezett értékkel. Erre használható az \mathbb{C} kombinátor:

```
1 infix 1  $\mathbb{C}$ 
2 ( $\mathbb{C}$ ) :: ([a] -> b) -> a -> [Maybe a] -> b
3 f  $\mathbb{C}$  b = f . map (maybe b id)
```

2.1. forráskód. Az \mathbb{C} kombinátor

Példa:

"=sum \mathbb{C} (-1) \$ \$a0:a5\$" \rightarrow az a0-a5 cellalista összege, üres cella esetén levon egyet az összegből.

Előfordulhat, hogy az üres cellákat egyáltalán nem szeretnénk figyelembe venni. Erre szolgál az *onJusts* kombinátor:

```
1 onJusts :: ([a] -> b) -> [Maybe a] -> b
2 onJusts f = f . map fromJust . filter isJust
```

2.2. forráskód. Az onJusts kombinátor

Példa: `"=onJusts (concat . map tail) $a0:a5$"` \rightarrow az `a0-a5` cellalista nemüres elemeinek konkatenálása, de mindegyikből elhagyva az első karaktert.

De természetesen közvetlenül is kihasználható a listahivatkozások *[Maybe a]* reprezentációja. A következő kódrészlettel például megszámolhatók a nemüres cellák: `"=length $ filter isJust $ $a0:a5$"`.

2.5.2. Alapértelmezett függvénytípusok

Bizonyos függvényeket gyakran szeretnénk egy alapértelmezett módon használni. Például cellák összegének kiszámításához a *sum* $\in 0$ függvényt, vagy cellák maximumának megkereséséhez az *onJusts maximum* függvényt.

Az *Empty* modul (**ÁT LESZ NEVEZVE!!**) biztosítja a *Prelude* függvényeinek egy-egy az alkalmazás reprezentációjához igazított alapértelmezett változatát. Az alapértelmezett függvények neve az eredeti függvény neve és utána egy "D". Például: *sumD*, *maximumD*, *foldMapD*.

2.5.3. A könyvtár bővítése

A felhasználónak lehetősége van saját modulokat írni az alkalmazáshoz, és azokat használni. (A modulok importálásáról a 2.3 szakaszban esett szó.) Saját polimorf függvények írásakor érdemes explicit kiírni a típusokat, ugyanis a monomorfizmus-megszorítás (**KELL LINK?**) miatt a Haskell fordító gyakran túl specifikus típust következtet ki. A standard könyvtár definiál két típuszinomimát listafüggvényekhez:

```
1 type LFun a b = [Maybe a] -> b
2 type NLFun a b = Num a => [Maybe a] -> b
```

2.3. forráskód. LFun és NLFun

Emellett természetesen tetszőleges Haskell modul betölthető az alkalmazásba.

2.6. A parancssorban használható parancsok

2.6.1. Kifejezés kiértékelése GHCi-ban

A program a cellák kódjának kiértékeléséhez a háttérben egy GHCi példányt futtat. Lehetőség van kifejezések közvetlen kiértékelésére a GHCi-ben a *g* parancs segítségével. A kiértékelés eredménye megjelenik log üzenetként. Példák:

1. *g sum [1..10] * 2*
2. *g :t map*

Ezen parancs segítségével van elméleti lehetőség modulokat importálni és definiciókat megadni közvetlenül a GHCi-ban. Azonban minden egyes cellakiértékelés előtt a bindingok elvesznek, a betöltött modulok pedig azok lesznek, amelyek a "Modules" menüpontban meg lettek adva. Így a gyakorlatban a GHCi ezen funkciói nem használhatók.

2.6.2. Cellatartalom másolása és áthelyezése

A *cp* parancs használható cellák egy blokkjának átmásolására, az *mv* parancs pedig egy cellablokk áthelyezésére. A két parancs szintaxisa nagyon hasonló formátumú: *<parancs> <listahivatkozás> <egyszerű hivatkozás>*

A listahivatkozás adja meg a másolandó/áthelyezendő tartományt (bal felső és jobb alsó sarok), az egyszerű hivatkozás pedig a céltartomány bal felső sarkát. Példák:

1. *mv §a0:a5§ §b10§*
2. *cp §b1:c3§ §e9§*
3. *cp §a0:a3§ §a1§*

Cellák másolásakor a cellák kódjában található hivatkozások alapértelmezetten relatív hivatkozásként kerülnek értelmezésre. A hivatkozásokban §-ral lehet jelölni azokat az oszlop- és sorazonosítókat, melyeket abszolút hivatkozásként szeretnénk kezelni.

Példa: Legyen *C0* kódja *=sumD §a0:b5§ + §§C§5§*. Ekkor a *c0* cellát *d1*-be másolva, a *d1* cella kódja *=sumD §b1:c6§ + §§C§5§*

A parancsot lehet úgy paraméterezni, hogy a kiinduló és a céltartomány átfedésben legyen (fenti 3. példa). Ebben az esetben a parancs az új értékekkel felülírja a

metszetben lévő cellákat. Ha pl. a fenti 3. példában $\$a0:a4\$$ rendre $[1,2,3,4,\ddot{U}RES]$, akkor a parancs végrehajtása után $[1,1,2,3,4]$ lesz a cellák tartalma.

Megjegyzés: a parsolási hibás cellák másolása/mozgatása során nem garantált, hogy a célcella kódja megegyezik a forráséval. A hiba elkerülésének érdekében azt javaslom a felhasználónak, hogy sokszorosítás helyett inkább javítsa ki a hibát.

3. fejezet

Fejlesztői dokumentáció

3.1. A fejlesztői dokumentáció felépítése

A fejlesztői dokumentáció három nagy részből áll.

A 3.2 részben ismertetésre kerülnek a szoftver készítése során felhasznált technológiák, valamint nagy vonalakban a szoftver logikai felépítése. (Milyen programkomponensek vannak, milyen feladatokat látnak el, hogyan kapcsolódnak egymáshoz.)

A 3.3-3.6 részek tartalmazzák az egyes komponensek részletesebb leírását. Minden komponens esetén ismertetésre kerül a más komponensek felé nyilvánosságra hozott interfész, valamint a komponensek működési elve, beleértve a használt típusok leírását és a fontosabb algoritmusok működési elvét. A rész sorrendileg úgy van felépítve, hogy először szerepelnek a vezérlő komponens (*App*) által használt komponensek leírásai, majd ezt követően az *App* moduljai. Így "lentről fölfelé" haladva először meg lehet érteni az egyes kisebb részek működését, majd azok segítségével az egész alkalmazását.

A 3.7 rész tartalmazza a tesztelési eljárás leírását és a tesztelés eredményeit.

3.2. A szoftver felépítése

3.2.1. Felhasznált technológiák összefoglalása

Az alkalmazás Haskell nyelven íródott. A grafikus megjelenítés *GTK+* alapú, a *gtk2hs* csomag által biztosított bindingokat használtam a grafikus felület kezelésé-

hez. Ez a csomag a *GTK+* osztályhierarchiáját Haskell típusosztályok hierarchiájaként reprezentálja. Az egyes osztályok metódusainak a típusosztályok definíciójában szereplő függvények felelnek meg. A *GTK+* típusai *foreign pointer*ek segítségével vannak megvalósítva, és *IO*-ban használhatók.

Az alkalmazás a *GTK+* logikájának megfelelően eseményvezérelt. A felhasználó akciói eseményeket váltanak ki, amelyek hatására eseménykezelők futnak le. A handlerok minden esetben *IO* akciók, amelyek valamilyen módon módosítják a globális állapotot (lásd 3.2.1. Globális állapot). A szoftver fejlesztése során fontos volt, hogy minél kevesebb legyen az tisztátalan (impure), *IO*-n belül elvégzett számítás. Igyekeztem a program logikájának minél nagyobb részét egy tiszta, nem monadikus környezetben megvalósítani. Így a számítások helyessége könnyebben tesztelhető/-verifikálható, az eseménykezelők már keveset számolnak az *IO*-ban.

Az alkalmazás a parseolási feladatokhoz a *parsec* csomagot, a gráfok kezeléséhez az *fgl* csomagot, a *ghci* futtatásához pedig a *ghcid* csomagot használja. A modell adatainak könnyebb kezeléséhez a *microlens-platform* csomagot használtam, ami a jól ismert *lens* csomag egy kevesebb funkciót és kevesebb függőséget tartalmazó változata. A függőségek pontos listája elérhető a Felhasználói dokumentációban, illetve az egyes programkomponensek részletes leírásakor is említésre kerülnek a fontosabb felhasznált csomagok.

3.2.2. A globális állapot

Az alkalmazás fő felépítését egy az FP Complete blogján megjelent cikk (**IDE KÉNE EGY REFERENCIA**) inspirálta. Az alkalmazás a globális (olvasható) állapotot a *ReaderT* monád transzformer segítségével valósítja meg, az alkalmazás vezérlése így egy *ReaderT Env IO* környezetben történik, ahol *Env* a globális állapotot leíró adattípus. Fontos megjegyezni, hogy bár az *Env* típus komponensei az inicializálás után sosem módosulhatnak, a mögöttes állapot még változhat, hiszen a komponensek módosítható referenciák. Ez hasonló a Java nyelvben használható konstans referencia koncepciójához: a referencia nem változhat, de a referált adat igen.

A fentebb referált cikk által inspirálva a (GUI komponensein kívüli) globális állapot egy *StateT* transzformer helyett módosítható referenciákkal (*IORef* és *MVar*)

kezeltek. Ugyanis hiába tiszta, ha globálisan használjuk a *StateT*-t, valójában – a programlogika szintjén – ugyanúgy egy globális, módosítható állapotot vezetünk be. Szintén egy szempont, hogy a *GTK+* alapú *GUI* miatt eleve szerepelnek módosítható referenciák (foreign pointer) a globális állapotban, így ez a probléma semmiképpen nem kerülhető el teljesen. Egy további érv a globális *StateT* ellen, hogy egy nagyobb monad stack szükségszerűen bonyolítja a programot. A *ReaderT IO* ellenben még kifejezetten könnyen kezelhető. A cikk konkurenciához köthető problémákat is említ a *StateT*-vel kapcsolatban. Ez a szoftver jelenlegi verziójában még nem olyan jelentős (lévén a mostani implementáció nagyon kis mértékben épít konkurenciára). Azonban a jövőre nézve mindenképpen előnyös, ha a szoftvert könnyen lehet a konkurens paradigma szerint bővíteni.

Ezen bevezető után tekintsük a globális állapot definícióját! Az alábbi típusdefiníció az *App.Types* modulban található:

```
1 data EvalConfig = EvalConfig { modules :: [String]
2                               , paths  :: [String]
3                               }
4
5 data EvalControl = EvalControl { eGhci      :: MVar Ghci
6                                , eCommand  :: MVar String
7                                , eResult   :: MVar (Either String [
8                                    String])
9                                , eConfig   :: MVar EvalConfig
10                               } deriving Eq
11
12 data SaveStatus = Saved | Modified
13
14 data File = File FilePath SaveStatus
15
16 data Env = Env { evalControl  :: EvalControl
17                 , gui         :: Gui
18                 , state       :: IORef Spreadsheet
19                 , file         :: IORef (Maybe File)
20                 }
```

3.1. forráskód. Az Env típus

Az *evalControl* mező tartalmazza a kifejezések ghci-ban való kiértékeléséhez szükséges erőforrásokat. Az *eConfig* mező tartalmazza a GHCi-hoz tartozó konfigurációs beállításokat. (Betöltött modulok listája, és a modulok keresési útvonalainak listája.) Az *eGhci* mező tartalmazza a háttérben futó GHCi példányra való hivatkozást. Az *eCommand* és az *eResult* valósítják meg a kommunikációt a kiértékelést végző szál és az alkalmazás fő szála között. Az *eCommand*-nak a fő szál a termelője, és a kiértékelő szál a fogyasztója, az *eResult*-nak pedig fordítva.

A *gui* mező tartalmazza a GUI komponenseit. A pontos típusdefiníció a GUI leírásánál fog szerepelni.

A *state* mező egy módosítható referencia, ami a számológépet reprezentáló, *Spreadsheet* típusú adatot referálja. A file adattag tartalmazza az éppen a táblázatkezelőbe betöltött fájl fontosabb adatait, amennyiben be van töltve egy fájl. (Fájl neve, és állapota.)

Az *EvalConfig* típus példánya *Generic* és *Serialize* típusosztályoknak. Erre az adatok mentéséhez van szükség (lásd 3.3.8.).

3.2.3. A GUI

Az alábbi kódrészlet a GUI definíciója az *App.Types* modulból:

```
1 data Menubar = Menubar { newButton :: Button
2                               , saveButton :: Button
3                               , loadButton :: Button
4                               , modulesButton :: Button
5                               , pathsButton :: Button }
6
7 data Gui = Gui { mainWindow    :: Window
8                   , logWindow  :: ScrolledWindow
9                   , log        :: TextBuffer
10                  , table      :: Table
11                  , entryKeys   :: [(Entry, (Int, Int))]
12                  , colButtonKeys :: [(Button, Char)]
13                  , editor      :: Entry
14                  , commandLine :: Entry
15                  , menu        :: Menubar }
```

3.2. forráskód. A Gui típus

A *Gui* típus tartalmazza a *Gui* azon komponenseit, amelyekre szükség van az eseménykezelők hozzáadásakor. A *mainWindow* komponens tartalmazza az alkalmazás fő ablakát.

A számológép megjelenítése egy *Table* segítségével történik. A cellákat a *Table*-ben elhelyezett *Entry*-k reprezentálják. Minden *Entry*-hez egy (Int, Int) kulcs is el tároltatik. Ez mutatja, hogy a tábla melyik pozíciójához tartozik az adott *Entry*. Ez egy objektumorientált nyelvben megoldható lenne egy leszármazott widgettel (amelynek van egy extra mezője), azonban a *gtk2hs* keretein belül ezt körülményes lett volna megoldani, így inkább ezt a megoldást választottam. Az *Entry*-kulcs párokat az *entryKeys* mező tartalmazza. Az oszlopok tetején levő gombokat a *colButtonKeys* mező tartalmazza. Ebben a mezőben az is el van tárolva, hogy melyik gombhoz melyik oszlop tartozik.

A grafikus felület alján található log egy *ScrolledWindow*-ban elhelyezkedő *TextView*. Magára a *TextView*-ra nincs szükség, csak az általa használt *TextBuffer*-re, ezért azt nem is tartalmazza a *Gui*. Az editor és a parancssor egy-egy *Entry*-vel vannak implementálva. A *menu* komponens tartalmazza felső menüsoron elérhető gombokat.

A GUI layout ennél alaposabb dokumentációja az *App.CreateEnv* modul leírásában érhető el.

3.2.4. Programkomponensek és modulszerkezet

Az alábbiakban röviden összefoglalom a szoftver moduljainak fő feladatát:

- **Main** – főprogram
- **App** – az alkalmazás fő logikája, eseménykezelés
 - **App.CreateEnv** – a globális állapot inicializálása, funkcionalitás nélküli GUI létrehozása
 - **App.RunApp** – a főprogram definiálása, a main loop terminálásakor végrehajtandó IO akciók megadása
 - **App.Setup** – funkcionalitás hozzárendelése a GUI komponenseihez
 - * **App.Setup.CommandLine** – a parancssor eseményeinek kezelése

- * **App.Setup.Editor** – a kódszerkesztő eseményeinek kezelése
- * **App.Setup.Global** – több GUI komponens által is használt akciók
- * **App.Setup.Menubar** – menüsor gombjaihoz tartozó események kezelése
- * **App.Setup.Table** – a számolótáblát megjelenítő táblázat eseményeinek kezelése
- **App.Types** – a globális állapothoz tartozó típusdefiníciók
- **Eval** – kifejezések GHCi-ban történő kiértékelése
 - **Eval.CodeGeneration** – kódgenerálás a kiértékeléshez (ezen modul dokumentációjában szerepel a kiértékelési modell leírása is)
 - **Eval.CommandLine** – a parancssorba beütött parancsok reprezentációjának előállítása
 - **Eval.EvalMain** – a tényleges kiértékelést végző szál főprogramja
 - **Eval.Ghci** – az **App** számára biztosított interfész a kiértékeléshez
- **GraphFunctions** – a *DynGraph* típusosztályhoz kapcsolódó, más modulokban felhasznált segédfüggvények
- **Persistence** – az **App** számára biztosított interfész fájlok mentéséhez és betöltéséhez
- **Spreadsheet** – a számolótábla reprezentációja és műveletei
 - **Spreadsheet.Interface** – a számolótábla műveletei, amiket az **App** használhat
 - **Spreadsheet.Parser** – felhasználó által írt kód reprezentációjának előállítása
 - **Spreadsheet.Types** – a számolótábla és kapcsolódó kivételek típusdefiníciói

Az alkalmazáshoz emellett hozzátartozik az **Empty** modul, ami az alkalmazás standard könyvtára. Ez a modul nem része az alkalmazásnak (abban az értelemben,

hogy egyik fordítási egység sem hivatkozik a modul tartalmára), de a háttérben futó GHCi folyamatban mindig elérhetők a modul által exportált függvények. Az **Empty** modulról bővebben a **2.5** szakaszban esik szó.

3.3. Spreadsheet

3.3.1. Spreadsheet.Types

Az alkalmazás a Spreadsheet típussal reprezentálja a számológépek állapotát. Alább látható a Spreadsheet.Types modulban szereplő definíció:

```
1 type CellID = Int
2
3 data Cell' = Str String | Number Double | EmptyCell
4
5 data ForPiece = Code String | Refs [CellID]
6
7 data FormulaError = FNoParse
8                   | FCycleRefError
9                   | FNoCache
10                  | FListTypeError
11                  | FMissingDepError
12                  | FGhciError
13                  | FTimeoutError
14
15 data Formula = Formula { _code :: String
16                        , _cache :: Either FormulaError Cell'
17                        , _value :: Maybe [ForPiece]
18                        }
19
20 data Cell = Val {_cellV :: Cell'} | For {_cellF :: Formula}
21
22 data Spreadsheet = SS { _sheet :: Gr Cell Int
23                       , _selected :: Maybe CellID
24                       , _logMessage :: Maybe String
25                       }
```

3.3. forráskód. A Spreadsheet típus

A Spreadsheet egy rekord típus, amelynek három mezője van. A *_selected* mező jelenti az aktuálisan kijelölt cellát. Ez a mező kerülhetett volna a globális állapotba is, azonban a tervezés korai fázisában más döntés született, és a későbbiekben körülményes lett volna refaktorálni a kódot. (Ugyanakkor ennek átírása tervben van az alkalmazás egy későbbi verziójában.) A *_logMessage* mező tartalmazza a legutóbbi művelet kiértékeléséből származó szöveges (a GUI-ban a logra írandó) üzenetet.

A *_sheet* mező reprezentálja a tényleges számolótáblát. A számolótábla egy irányított gráf, aminek a csúcsai *Cell* típusú értékekkel vannak címkézve. Az élek egész számokkal vannak címkézve. (A számszerű élcímkekre csak a használt gráfcsomag által biztosított útkereső algoritmus megszorításai miatt van szükség. Az alkalmazás implementációjában minden él címkéje 1.) A gráfban minden csúcs a számolótábla egy cellájának felel meg. Egy *A* csúcsból pontosan akkor megy él egy *B* csúcsba, ha a *B* csúcsban található cella kódja hivatkozik az *A* csúcsban található cellára.

Egy cella pontosan akkor szerepel a gráfban, ha nemüres vagy van olyan cella, amelyik hivatkozik rá. Így az egyszerre üres és nemhivatkozott cellák tárolására nincs szükség.

A fent megadott gráfrepresentációnak két további előnye is van. Egyrészt könnyű körfigyelést implementálni, így elkerülve, hogy cellák körkörösén hivatkozzanak egymásra; másrészt ha módosul egy *A* cella tartalma, akkor pontosan az *A*-ból elérhető csúcsoknak megfelelő cellákat kell újra kiértékelni.

A gráfrepresentáció megvalósításához az *fgl* csomagot használtam. A *Spreadsheet* típus definíciójában a *PatriciaTree* alapú *Gr* típust használtam. A műveletek a *DynGraph* típusosztály tetszőleges megvalósítására működnek, így lehetséges a belső gráfrepresentáció cseréje tetszőleges másik, *DynGraph* példánnyal rendelkező típusra.

Egy cella tartalmát a *Cell* típus reprezentálja. Egy cella tartalma lehet érték (*Cell'*) vagy formula (*Formula*). Az érték jelenleg háromféle lehet: szám (*Double*), string vagy üres.

Ha egy cella formulát tartalmaz, az a háromelemű *Formula* rekorddal reprezentáltatik. A *_code* mező tartalmazza a felhasználó által megadott kódot. Ez egy kényelmi funkció, hogy a kód megjelenítéséhez ne kelljen visszakonvertálni a reprezentációból. A *_cache* mezőben szerepel, hogy mi a formula legutóbbi kiértékelésé-

nek eredménye (ha egyáltalán már ki lett értékelve). A cache értéke vagy egy érték (*Cell*) vagy valamilyen hiba (*FormulaError*).

A *_value* jelenti a formula kódgeneráláshoz szükséges reprezentációját. Ez a reprezentáció *ForPiece*-ek (formuladarabok) listája. Egy formuladarab vagy egy kódrészlet (*String*) vagy cellaazonosítók listája. A *_value* mezőről részletesebben lesz szó a *Spreadsheet.Parser* és a *Spreadsheet.CodeGeneration* modulok leírásában.

A *Formula* típushoz tartozik egy invariáns állítás: a program futása során egy *Formula* mindig a 3.1 táblázatban leírt állapotok valamelyikében figyelhető meg.

Érdemes megjegyezni, hogy ez az invariáns típuszinten is garantálható lett volna (feladat az olvasó számára!). A jelenlegi megoldás a korai tervezési fázis eredménye, a későbbiekben már erőforrásigényes lett volna refaktorálni a kódot.

További megjegyzések a *Spreadsheet* típussal kapcsolatban:

1. A *Spreadsheet.Types* modul alapértelmezett nevű lenseket is exportál a *Cell*, *Formula* és *Spreadsheet* típusokhoz.
2. A *Spreadsheet.Types* modulban szereplő összes típus (a kivételek kivételével) példánya a *Generic* és *Serialize* típusosztályoknak (ez utóbbit a *cereal* csomag exportálja). Erre a perzisztencia implementációjához van szükség.
3. A *FormulaError* típus *FListTypeError* konstruktora jelenleg nincs használva. Ez a konstruktor azt jelezné, ha egy hivatkozáslista nem értelmes, mert nem homogén. Ez jelenleg nem ellenőriztetik, az inhomogén listák által okozott hibák csak a kiértékelés során jelentkeznek. Amennyiben a későbbiekben implementálásra kerülne ez a funkció, lehetséges az *FListTypeError* konstruktor felhasználása.

3.3.2. Spreadsheet.Parser

A modul feladata egy a felhasználó által egy cellához megadott kód (*String*) reprezentációjának (*Cell*) előállítás. A modul egy függvényt exportál: (*rep :: String -> Cell*)

Legyen a felhasználó által megadott kód *str*. A *rep* függvény az alábbi specifikáció szerint állítja elő a kód cellareprezentációját. A reguláris kifejezések megadásakor a kifejezésre illeszkedő string előtt és után tetszőlegesen sok szóköz lehet, ha nincs másképp jelezve.

Minta			Jelentés
Formula	_	(Left FNoParse) Nothing	parseolási hiba
Formula	_	(Left FCycleRefError) Nothing	sikeres parseolás, azonban a formula körkörös referenciákat adott volna a táblához
Formula	_	(Left FNoCache) (Just _)	sikeres parseolás, érvényes referenciák, de a formula még nem lett kiértékelve
Formula	_	(Left FListTypeError) (Just _)	<i>nem használt</i>
Formula	_	(Left FMissingDepError) (Just _)	a formula nem értékelhető ki, mivel egy hivatkozott cella nem volt cache-elve.
Formula	_	(Left FGHCLError) (Just _)	a formula egyéb okokból nem volt kiértékelhető (pl. típushiba, Haskell szintaxis-hiba)
Formula	_	(Left FTimeoutError) (Just _)	időtúllépés miatt sikertelen kiértékelés, valószínűleg végtelen ciklus miatt
Formula	_	(Right cell') (Just _)	sikeres kiértékelés, az eredmény cell'

3.1. táblázat. Egy *Formula* lehetséges állapotai

1. Ha $str = ""$, $rep\ str = Val\ EmptyCell$
2. Ha str illeszkedik a $((+| - |\varepsilon)D^+|.D^+)$ reguláris kifejezésre, ahol $D=(0|1|2|3|4|5|6|7|8|9)$, akkor $rep\ str = Val\ (Num\ n)$, ahol n a literál által ábrázolt lebegőpontos szám.
3. Ha a fenti esetek egyike sem áll fent, és str nem $(=C^*)$ alakú (ahol C az összes karakterek halmaza), akkor $rep\ str = Val\ (Str\ str)$
4. Ha B a betűk halmaza (a *Data.Char* modul *isLetter* függvényének igazsághalmaza), $C' = C \setminus \{\$ \}$ és $str = ((\S BD^+ : BD^+ \S)|(\S BD^+ \S)|(C'^+))^+$ alakú, akkor a kód formulaként parseolható. $rep\ str = Formula\ str\ (Left\ FNoCache)\ (Just\ ps)$, ahol ps definíciója alább szerepel.
5. Ha egyik fenti eset sem áll fent, akkor a parseolás sikertelen. Ekkor $rep\ str = Formula\ str\ (Left\ FNoParse)\ Nothing$

Ha str formulaként parseolható (fenti 4. eset), egy egyszerű szintaktikus elemzés segítségével kaphatjuk a reprezentációjának *_value* komponensét. A parser először elhagyja az $=$ karaktert. Ezután sorban parseol substringeket a szó elejéről az alábbi módon:

1. Először megpróbálja cellahivatkozásként olvasni a soron következő részt: $((\S BD^+ : BD^+ \S)|(\S BD^+ \S))$. Ha sikerült, a hivatkozást cellaazonosítók sorozatává konvertálja (lásd alább), és a kapott rs azonosítólistát *Refs rs* módon az eredménylista végére fűzi.
2. Ha a soron következő substring nem olvasható cellahivatkozásként, akkor a parser végigolvassa a lehető leghosszabb $s = C'^+$ substringet, és az eredménylistához egy *Code s*-t ír.

A cellahivatkozások feloldásához kihasználjuk, hogy a karakterek injektíven az egész számok halmazára képezhetők (az *Enum* típusosztály műveleteivel). A kis- és nagybetűket nem különböztetjük meg. Emellett definiálunk egy *Enum* példányt (Int, Int) párokra. **STACKOVERFLOW LINK KELL** Ez a példány csak nem-negatív elemű párokra működik helyesen, de a program számára ez is elég. Jelölje $fromEnum^C$ a karaktert Int -té kódoló függvényt, és $fromEnum^P$ az (Int, Int) párt Int -té kódoló függvényt. Jelölje $a, b :: Int$ esetén $[a..b]$ az $a \leq n \leq b$ n egész számok rendezett listáját! Ekkor a cellahivatkozások feloldása az alábbiak szerint történik:

1. Ha a hivatkozás $(\S BD^+ \S)$ alakú, legyen b a betű, és n a számjegyek ál-

tal reprezentált egész szám. Ekkor a kapott rs lista egyelemű. $rs = [fromEnum^P (fromEnum^C, n)]$

2. Ha a hivatkozás ($\S BD^+ : BD^+\S$) alakú, felbontható a $:$ mentén két egyszerű hivatkozásra. Legyenek a betűk d_1 és d_2 , a számjegyek által reprezentált számok pedig rendre n_1 és n_2 ! Ekkor haskelles listakifejezésként a következő módon definiálhatjuk a kapott azonosítólístát:

$$rs = [fromEnum^P(r, c) | r \leftarrow [fromEnum^C(d_1)..fromEnum^C(d_2)], c \leftarrow [n_1..n_2]]$$

3.3.3. Spreadsheet.Interface

Ebben a modulban szerepelnek az *App* komponens számára elérhető, a *Spreadsheet* típushoz kapcsolódó függvények. A 3.2 táblázatban szerepel a modul által exportált függvények és azok feladatainak listája.

setCellState :: CellID -> String -> Spreadsheet -> Spreadsheet

A *setCellState* függvény feladata, hogy a megadott cellaazonosítóhoz tartozó csúcsban lévő cella állapotát a felhasználó által megadott *String*-nek megfelelően módosítsa, valamint felülírja a *_logMessage* mező tartalmát a művelet sikerességétől függően.

Ehhez szükség van a megadott *String Cell* reprezentációjára, amit a *Spreadsheet.Parser* modul által exportált *rep* függvény számít ki. A kapott reprezentáció alapján az alább leírtaknak megfelelően viselkedik a függvény:

1. Ellenőrzi, hogy a cella állapotának megváltoztatásával keletkeznék-e körkörös referencia. Pontosabban akkor keletkeznék, ha a megváltoztatandó c azonosítójú cellához tartozó csúcsba bemenő összes él kitörlésével keletkezett gráfban van olyan n azonosítójú csúcs, hogy c kódja hivatkozik n -re és a gráfban már van $c \rightarrow n$ út. (Ez utóbbi feltétel azt jelenti, hogy az n cella értéke függ c értékétől.) Ezt a feltételt az *isLegal* segédfüggvény ellenőrzi. Érdekes megjegyezni, hogy ilyen hiba csak akkor fordulhat elő, ha a kapott reprezentációnk egy formula.
2. Amennyiben az *isLegal* eredménye *False*, a c csúcsban levő cella reprezentációja *For (Formula str' (Left FCycleRefError) Nothing)* lesz, ahol str' a paraméterként kapott *String*. A *_logMessage* mezőbe egy hibäüzenet kerül.

Függvény	Típus	Feladat
emptySpreadsheet	Spreadsheet	üres számolótábla (0 csúcsú gráf, nincs kijelölt cella, nincs log üzenet)
getCellText	CellID -> Spreadsheet -> String	egy cellában megjelenítendő szöveg lekérdezése
getCellCode	getCellCode :: CellID -> Spreadsheet -> String	egy adott cellába legutóbb beírt kód lekérdezése
setCellState	CellID -> String -> Spreadsheet -> Spreadsheet	a megadott cella állapotának módosítása egy a felhasználó által megadott String alapján
cacheCell	CellID -> Either EvalError String -> Spreadsheet -> Spreadsheet	kiértékelés eredményének cachelése
getSelected	Spreadsheet -> Maybe CellID	kijelölt cella azonosítója
setSelected	CellID -> Spreadsheet -> Spreadsheet	kijelölt cella azonosítójának beállítása
getLogMessage	Spreadsheet -> String	legutóbbi log üzenet lekérdezése

3.2. táblázat. A *Spreadsheet.Interface* által exportált függvények

3. Amennyiben az *isLegal* segédfüggvény *True* eredményt ad, a gráfból kitöröltetik az összes *c*-be menő él, és új *c*-be menő élek kerülnek behúzásra a *c* kódja által referált celláknak megfelelő csúcsokból. (Ezeket a *references* segédfüggvény számolja a reprezentációból.) A *_logMessage* mező tartalma egy sikert jelző üzenet lesz.
4. Amennyiben a módosított cella üressé vált, az összes olyan üres cellához tartozó csúcs kitöröltetik a gráfból, melyekből csak a módosított cellába ment él.
5. Amennyiben a módosított cella üressé vált, és a hozzátartozó csúcs kifoka 0 (nem hivatkozik más cella az üressé tett cellára), úgy a csúcs töröltetik a gráfból.

Megjegyzés: az utolsó két pont garantálja, hogy nem a program nem tárol feleslegesen cellákat a gráfban.

cacheCell :: CellID -> Either EvalError String -> Spreadsheet -> Spreadsheet

A függvény feladata, hogy a megadott cellaazonosítóhoz tartozó csúcsban lévő cella állapotát frissítse a kiértékelés során kapott eredménnyel. A frissítendő cella szükségszerűen egy formula, a *cacheCell* ennek *_cache* mezőjét módosítja az alábbiak szerint:

1. Ha a kiértékelés eredménye hiba (a második paraméter *Left*), akkor a cache-be a megfelelő hiba kerül.
2. Ha a kiértékelés eredménye értelmes (a második paraméter *Right*), akkor a függvény értékként parseolja a kapott eredményt, és a parseolt értéket írja a cache-be.

3.4. Eval

3.4.1. Eval.CodeGeneration

A kiértékelési folyamat az alábbi szempontok alapján került megtervezésre:

1. Listán értelmezett függvények esetén legyen lehetőség könnyen kezelni az üres cellákat, alapértelmezett értékek megadásával.

2. Ha a kiértékelés során hiba történik, annak hatása minimális legyen.

Az első szempont megvalósításához a kiértékelés során egy kiszámított vagy paraméterként kapott a típusú értéket egy *Maybe a* típusú érték reprezentálja. Az üres cellákat *Nothing* reprezentálja, az értékkel rendelkező cellákat pedig egy *Just* érték. Egy a kiértékelés során kiszámított cellaérték is mindig egy *Just*-ba csomagoltatik. Ennek megfelelően a típusú cellaértékek listájának futásidejű reprezentációja egy *[Maybe a]*. A Haskellben megszokott listafüggvények a felhasználói dokumentációban részletesen leírt függvények segítségével használhatók ezen a reprezentáción.

A második szempont megvalósításához a kódgenerálás során értékadások sorozata jön létre, és a kiértékelés során ezen értékadások egyesével hajtódnak végre. Így ha valamelyik cella kiértékelésének eredménye egy hiba, csak a tőle függő cellák értéke lesz hiba.

Ha sikeres a generálás, a *generateCode* függvény eredménye két (egy *Right* konstruktorba csomagolt) lista *([String],[(String, CellID)])*. Az első listába kerülnek az értékadások, amelyek az úgynevezett külső függőségekhez lettek generálva. A második lista tartalmazza az olyan értékadásokat, amelyek a megváltoztatott id-jű cellától függenek. A *Spreadsheet.Types* modulnál tárgyalt gráfrepresentáció segítségével az előbbi két fogalmat az alábbi módon tehetjük precízzé:

Legyen *id* a megváltoztatott cella azonosítója, és legyen *lab : CellID → Cell* a gráf csúcsaihoz a megfelelő cellát hozzárendelő függvény! Legyenek *For, Lab : Cell → ℒ* predikátumok, amelyek akkor adnak igazat, ha a paraméterük a megfelelő konstruktorral jött létre. Legyen *b* egy létező cellaazonosító! Ekkor:

$$\begin{aligned} b \text{ külső függőség} &\Leftrightarrow \exists c : (\exists (id \rightarrow c \text{ út}) \wedge \exists (b \rightarrow c \text{ út}) \wedge \neg (id \rightarrow b \text{ út})) \\ &\quad \vee ((b = id) \wedge Val(lab(b))) \\ b \text{ függ id-től} &\Leftrightarrow ((b \neq id) \wedge \exists (id \rightarrow b \text{ út})) \\ &\quad \vee ((b = id) \wedge For(lab(b))) \end{aligned}$$

Az *id*-től függő olyan módon kell sorrendbe rendezni, hogy az eredménylistában egy cella csak az őt a listában megelőző celláktól függjön. Ekkor az értékadásokat lehetséges a lista által megadott sorrendben kiértékelni. A sorba rendezéshez a leghosszabb utak algoritmus szerinti szintek használatosak, ezek alapján kerülnek

növekvő sorrendbe az *id*-től függő cellák. Könnyű látni, hogy ez a sorrend megfelel a fent megfogalmazott elvárásnak.

A kódgeneráláshoz szükséges ellenőrizni, hogy a kapott külső függőségek értéke kiolvasható-e. Ez akkor lehetséges, ha a külső függőség egy *Val* (de nem *Val EmptyCell*), vagy egy olyan *For*, amelybe van cachelve érték.

Az *id*-től függő cellák esetén (ezek szükségszerűen formulák) azt kell ellenőrizni, hogy sikerült-e őket parseolni. Ez ekvivalens azzal, hogy a *_value* mező értéke *Just*.

A függőségek listáinak kiszámítása és a fenti ellenőrzések elvégzése a *dep-List* függvény feladata. A leghosszabb utak szerinti szinteket kiszámító függvény a *GraphFunctions* modulban szerepel.

Az értékadások generálásáért a *codeG* függvény felel. Ez az *n* azonosító cellához egy *vn = someCode* formájú értékadást generál. A *someCode* részt külső függőség esetén a *cacheG*, *id*-től függő cella esetén a *cellG* függvény számítja ki.

A *cacheG* függvény üres cellákhoz a *"Nothing"* stringet rendeli. Számokhoz és stringekhez pedig egy *"Just val"* stringet, ahol *val* a megfelelő szám/string. Amennyiben egész számról van szó, a függvény levágja a tizedesrészt a számlite-ről, hogy a GHCi egész típusúként értelmezhesse a literált.

A *cellG* függvény a formula *_value* komponensének elemeiből egy stringet állít elő. Ehhez a *_value* mező minden eleméhez egy stringet rendel, és ezeket konkatenálja, majd a legvégén elír egy *"Just \$ "*-t. A függvény az egyes elemekhez az alábbi módon rendel stringeket:

1. *Code code* \rightarrow *code*
2. *Refs [n]* \rightarrow *"fromJust vn"*
3. *Refs ids*, ha $|ids| > 1 \rightarrow$ az *ids*-ben szereplő cellaazonosítókból generált változónevek listája. Pl. *ids=[1,2,3]* esetén *"[v1,v2,v3]"*

A fenti leírásból jól látszik, hogy egy cella értéke mindig *Just*-ba lesz csomagolva. Ha egy cella értékét akarjuk használni, az a "szokásos módon" megtehető, mivel a változónév elé egy *fromJust* kerül. Cellák listája esetén azonban *Just* értékek listáját kapjuk. Érdemes megjegyezni, hogy a cellákhoz kiszámított érték mindig *Maybe* a típusú lesz, így az eredmény kinyeréséhez a *fromJust vn* kód szükséges. Ha az eredmény *Nothing*, ez futásidejű hibát okoz, amit a hívó ki tud olvasni a GHCi-ből, és cachelhet egy hibát a kiértékelt cellához (*EGhciError*). Ez a viselkedés az

App.Setup.Global.evalAndSet függvényben van definiálva.

A modul által exportált *generateCode* függvény segítségével végezhető el a fent leírt kódgenerálás.

3.4.2. Eval.EvalMain

A modul által exportált *evalMain* függvény a kifejezések GHCi-ben való kiértékelését végző szál főprogramja. A szál az alábbiak szerint működik:

1. Várakozik, ameddig a globális állapot *evalControl* mezőjének *eCommand* változójába egy GHCi utasítást nem ír a fő szál.
2. Kiüríti a változót, és kiértékeli a kapott utasítást a GHCi-ben.
3. Ha egy megadott idő után nem ér véget a kiértékelés (jelenleg 1 másodperc), lekérdezi a GHCi folyamathoz tartozó PID-et, majd megkeresi annak a gyerekfolyamatát (*childPid*), és az *eResult* változóba *Left childPid*-et ír. Ezt a folyamatot aztán a fő szál fogja kilőni.
4. Amennyiben időben véget ér a kiértékelés, a GHCi által eredményül adott sorok *result* listáját *Right result* módon az *eResult* változóba írja.

Az időtúllépés utáni viselkedés bonyolultsága egy szerencsétlen helyzet eredménye. A magyarázathoz meg kell ismerni a *ghcid* csomag által biztosított GHCi interfészt. A *startGhci* függvény a dokumentáció alapján elindít egy GHCi háttérfolyamatot, amellyel innentől egy megadott szálról kell interaktálni (a megszakítást kivéve). A valóságban azonban azt tapasztaltam, hogy két folyamatot indít el, amelyek közül az egyik gyereke a másiknak.

Időtúllépés esetén meg kell állítani a háttérben futó számítást. Erre szolgálna az *interrupt* függvény, ami egy *SIGINT* jelzést küld a GHCi folyamatnak. A GHCi folyamat azonban bizonyos esetekben ezt kimaszkolja, ilyenkor a számítást nem lehetséges az *interrupt* segítségével megszakítani. **IDE KÉNE LINK ERRŐL A DISKURZUSRÓL** A GHCi leállítására szolgáló *stopGhci* függvény pedig csak az egyik (a szülő) folyamatot terminálja a *startGhci* által indított két folyamatból. A másik folyamat pedig tovább folytatja a számítást. Az alkalmazás egy Haskell szálát tartalmazó verziójában ez kiéheztette a fő folyamatot.

Ezért van szükség arra, hogy a kiértékelés külön szálon fusson. Ugyanis időtúllépés esetén a fő szál, miután ütemezésre kerül, a kapott PID alapján a lehető legag-

Függvény	Típus
<code>execGhciCommand</code>	<code>String -> ReaderT EvalControl IO</code> (Either EvalError String)
<code>loadModules</code>	<code>ReaderT EvalControl IO ()</code>

3.3. táblázat. Az *Eval.Ghci* által exportált függvények

resszívabban (*SIGKILL*) terminálja a második GHCi folyamatot. Ez a tapasztalat szerint az első folyamatnak is véget vet, és semmilyen erőforrást nem szivároztat. A folyamat terminálása után új GHCi folyamat indítható.

A fenti megoldást a szükség szülte, és tapasztalatok alapján, próbálgatás útján állt össze. Nem ismert, hogy miért indít a *startGhci* két folyamatot. (Egy konzolból indított GHCi folyamathoz például csak egy PID tartozik.) A megoldás az én számítógépemen, Ubuntu 20.04 LTS operációs rendszer mellett működött, de nincs rá garancia, hogy más Linux rendszer (vagy akár egy másik számítógép!) esetén működni fog. (A működés feltétele, hogy ütemezésre kerüljön a fő szál.) Ráadásul csak e miatt az interakció miatt kellett konkurenciát adni az alkalmazáshoz. Valódi hatékonyságot ezzel nem nyertünk, hiszen az egyik szál mindig blokkolt állapotban lesz. (Mivel mindkét szál a fogyasztóként hozzá tartozó *MVar*-ra várakozik, ha éppen a másik szál dolgozik.)

A gyerekfolyamat megtalálásához a program rendszerhívást hajt végre, a *pgrep* parancsot használja *-P* kapcsolóval.

3.4.3. Eval.Ghci

A modul fő feladata, hogy kiértékeljen egy GHCi parancsot, és az eredményt értelmezze. Emellett lehetőséget biztosít a modulok és keresési útvonalak újratöltésére a globális konfiguráció alapján (*EvalConfig*).

A kiértékelés folyamata a (nem exportált) *execG* függvényben van leírva, és az alábbi módon zajlik:

1. A paraméterként kapott parancs a *eCommand* változóba kerül.
2. Kiolvasásra kerül az eredmény az *eResult* változóból.

3. Ha az eredmény *Left pid*, a kapott PID-hez tartozó folyamat terminálásra kerül, és a kiértékelés eredménye *Left ETimeoutError*. Új GHCi folyamat indul, és a hivatkozása bekerül a globális állapot *evalConfig* mezőjének *eGhci* mezőjébe.
4. Ha az eredmény *Right result*, ez a kiértékelés eredménye. *result :: [String]* a GHCi által eredményül adott sorok listája.

Az exportált *execGhciCommand* függvény ezt a viselkedést egészíti ki egy extra ellenőrzéssel.

1. Ha az *execG* eredménye *Left ETimeoutError*, akkor ez az eredmény.
2. Ha az *execG* eredménye *Right results*:
 - Ha *results* üres, az eredmény *Right ""*
 - Ha *results* egyelemű, az eredmény *Right (head (results))*
 - Ha *results* több elemű, akkor a GHCi több sornyi eredményt adott vissza.

Ezt a függvény hibának tekinti, és az eredmény *Left (EGhciError results)*

A *loadModules* akció beállítja a globális állapot *evalControl* mezőjének *eConfig* mezője alapján az elérési utakat, majd betölti a megadott modulokat. A korábban betöltött modulokat először kitölti (*:m*). Ilyenkor a felhasználó által közvetlenül a GHCi-ba megadott definíciók is elvesznek. Az akció betölti az *Empty* modult is (standard könyvtár).

3.4.4. Eval.CommandLine

A modul feladata a felhasználó által a parancssorba beütött parancsok reprezentációjának előállítás, amely alapján a parancsok végrehajthatók. A parancsok reprezentációját a *ClCommand* típus adja:

```

1 data ClCommand = ClGhci String
2                 | ClMv ((Int,Int),[(CellID, CellID)])
3                 | ClCp ((Int,Int),[(CellID, CellID)])

```

3.4. forráskód. A *ClCommand* típus

A *ClGhci* konstruktorral létrehozott parancs a GHCi-ban értékelendő ki. A másik két konstruktor az *mv* és *cp* parancsokhoz szolgál. Ezen konstruktorok második paramétere a forrás-cél cellaazonosító-párok listája, az első paraméter pedig ezen

azonosítópárok koordinátánkénti távolsága. A parancsok parseolása megfelel a felhasználói dokumentációban leírtaknak. A referenciák feloldása a *Spreadsheet.Parser* dokumentációjában leírtak szerint történik.

A modul exportálja a *parseCommand :: String -> Maybe ClCommand* függvényt, amely a fent leírtaknak megfelelően elvégzi a parseolást. Hiba esetén az eredmény *Nothing*, sikeres parseolás esetén pedig *Just*.

A modul emellett exportálja a *shiftCode :: (Int,Int) -> String -> String* függvényt, amely egy cella (feltételezetten helyesen parseolható) kódját (*String* paraméter) "eltolja" az *(Int,Int)* paraméter által megadott vektorral (oszlop eltolás, sor eltolás). Ez azt jelenti, hogy az összes relatív hivatkozást módosítja, az abszolút hivatkozásokat azonban helybenhagyja.

3.5. Persistence

A modul feladata, hogy fájlokat mentsen és betöltsön. Egy adatot akkor lehet elmenteni, ha az adat típusa példánya a *Serialize* típusosztálynak. Ezt a típusosztályt a *cereal* csomag biztosítja. Egy *Serialize* példány minden eleme bytearray-gé szerializálható. A program a számológépek és a konfigurációs fájlok perzisztálásához is bytearray formátumot használ.

Jelenleg a program egy konfigurációs fájlt használ, ennek nevét a *moduleConfigFile* konstans definiálja. Az alkalmazás bezárásakor ebbe a fájlba kerül mentésre a globális állapot *evalControl* mezőjének *eConfig* mezője.

saveSheet :: Serialize a => String -> a -> IO ()

Ez a függvény elment egy szerializálható adatot a megadott fájlnevével. Létező fájl esetén felülírás történik.

loadSheet :: Serialize a => String -> IO (Either String a)

Betölti a paraméterként kapott fájlból az adatot. Ha nem létezik a fájl, az eredmény egy hibaüzenet, ami jelzi, hogy a paraméterként kapott fájl nem létezik.

saveModuleConfig :: EvalConfig -> IO ()

A *saveSheet* speciális esete. A kapott paramétert a *moduleConfigFile* konstans által megadott fájlba menti.

loadModuleConfig :: IO (Maybe EvalConfig)

Betölti a modulkonfigurációs fájl tartalmát. Ha a fájl nem létezik, az akció eredménye *Nothing*.

3.6. App

3.6.1. App.RunApp

Ez a modul definiálja a főprogramot (*appMain :: IO ()*), amellyel egyenlő a *Main*-ben definiált *main* függvény. Az *appMain* két akcióból áll. Először inicializálja az olvasható globális állapotot (*App.CreateEnv.createEnv*), majd végrehajtja a *runApp :: ReaderT Env IO ()* akciót, az olvasható globális állapotot az előbb inicializált környezetre állítva. A *createEnv* akció elindítja a háttérben futó kiértékelő szálakat is. (*Eval.EvalMain.evalMain*).

A *runApp* akció a következőképpen határozza meg a program működését:

1. Hozzárendeli a GUI elemeihez az eseménykezelőket (*App.Setup.setupGui*).
2. Betölti a beállított keresési útvonalakat és modulokat a GHCi-be (*Eval.Ghci.loadModules*). Ha nem található a konfigurációs fájl, egy figyelmeztetést ír a standard kimenetre.
3. Beállítja, hogy az alkalmazás bezárásakor álljon le a GHCi és a modulok konfigurációja kerüljön mentésre.
4. Megjeleníti a GUI-t és elindítja a main loopot.

3.6.2. App.CreateEnv

Ez a modul exportálja az *App.RunApp*-ban használt *createEnv :: IO Env* akciót.

A *createEvalControl :: IO EvalControl* segédakció hozza létre az *evalControl* mező tartalmát. Az *eGhci* mezőhöz elindít egy GHCi-t. A GHCi alapértelmezett munkakönyvtára az alkalmazás futtatásának helye. Az *eCommand* és *eResult* mezőkhöz

létrehoz egy-egy üres *MVar*-t. Az *eConfig* mező tartalmát beolvassa a konfigurációs fájlból (*Persistence.loadModuleConfig*).

A *state* mező az üres számolótáblával kerül inicializálásra (*Spreadsheet.Interface.emptySpreadSheet*). A *file* mező *Nothing*-gal kerül inicializálásra, mivel kezdetben nincs betöltve fájl az alkalmazásba.

A *createGui :: IO Gui* segédakció építi fel a felhasználói felületet. Az alkalmazás grafikus felületét egy *Window* tartalmazza (*mainWindow*). Ennek az ablaknak a gyereke egy *VBox*, amely a további widgeteket tartalmazza. Ezek rendre a menüsor (egy *HBox*, ami *Button*-öket tartalmaz), az egysoros kódszerkesztő (*Entry*), valamint egy *VPaned*, aminek felső komponense a cellákat tartalmazó táblázat (*Table*), alsó komponense pedig a logot megjelenítő *ScrolledWindow*. Ezek alá kerül a parancssor (*Entry*).

A *Table* létrehozásakor jönnek létre a cellák megjelenítésére szolgáló *Entry*-k, melyek a pozíciójukat leíró kulccsal együtt kerülnek mentésre.

A menüsor gombjaihoz itt kerülnek hozzárendelésre a billentyűkombinációk.

Az akció *"*new file"*-ra állítja a fő ablak címét, mivel az alkalmazás elindításakor nincs betöltött fájl.

3.6.3. App.Setup

Ez a modul exportálja a *setupGui* akciót, amelynek feladata, hogy a GUI-hoz eseménykezelőket rendeljen. Ezek az eseménykezelők definiálják az alkalmazás lényegi működését. A *setupGui* akció rendre végrehajtja a *setupEditor*, *setupCommandLine*, *setupMenubar* és *setupTable* akciókat, amelyek definiálják az egyes GUI komponensek működését. Ezek az akciók a megfelelő nevű *App.Setup.** almodulban vannak definiálva.

Az eseménykezelők megadásához egy *IO ()* akcióra vagy egy *Event-> IO ()* függvényre van szükség. Ez azért hátrányos, mert nem lehetséges az eseménykezelőket közvetlenül a *ReaderT* kontextusban definiálni. Ezért az eseménykezelők megadására az alábbi minta használatos:

```
1 setupSomeWidget :: ReaderT Env IO ()
2 setupSomeWidget = do
3   widget <- ...
```

```
4   env <- ask
5   ...
6   void $ lift $ onSomeEvent widget $ runReaderT handlerAction env
7
8 handlerAction :: ReaderT Env IO ()
```

3.5. forráskód. Az eseménykezelők hozzárendelése

Tehát a eseménykezelőt hozzárendelő függvény lekérdezi a globális állapotot, és az eseménykezelő egy ReaderT IO akció futtatása a globális állapottal. Így viszonylag kényelmesen használható a globális állapot az eseménykezelők megírásakor. A fent leírt minta akkor is alkalmazható, ha van egy extra *Event* paraméter (pl. *onFocusOut*). Ilyenkor a *handlerAction*-nek paraméterként adható az *Event*.

3.6.4. App.Setup.Global

Ez a modul olyan akciókat definiál, melyeket az *App.Setup* további almoduljai felhasználhatnak. Az alább leírtak mellett a modul exportál néhány, a globális állapot egyes komponenseinek lekérdezését kényelmesebbé tevő akciót is.

setTitle :: ReaderT Env IO ()

Ez az akció állítja be az ablak címét a globális állapot *file* mezője alapján:

1. Ha a *file* változó tartalma *Nothing*, az ablak címe *"*new file"*.
2. Ha a *file* változó tartalma *Just fname* és a fájl állapota *Modified*, akkor az ablak címe *"*<fname>"*.
3. Ha a *file* változó tartalma *Just fname* és a fájl állapota *Saved*, akkor az ablak címe *"<fname>"*.

logAppendText :: String -> ReaderT Env IO ()

A paraméterként kapott *String*-et új sorként hozzáadja a log aljához, majd legörget a logot tartalmazó *ScrolledWindow*-ban. A legörgetés többsoros üzenet esetén csak az első sorig görget le. Ez egy ismert hiba, amit egyelőre nem sikerült javítani. Jelenleg nincs számontartva a log mérete, és nincs is maximális mérete. Ennek értelmében a logüzeneteket tartalmazó buffer mérete tetszőlegesen nagy lehet.

updateView :: ReaderT Env IO ()

A globális állapot *state* mezője alapján frissíti a cellákban megjelenő szöveget. Ehhez felhasználja a *Spreadsheet.Interface.getCellText* függvényt. A táblanézet frissítése után az ablak címét is frissíti a *setTitle* akció segítségével.

evalAndSet :: CellID -> ReaderT Env IO

Kiértékeli a megadott azonosítójú cellához generált kódrészleteket, majd a kiértékelés eredményét cacheli a számolótáblába. A betöltött fájl állapotát *Modified*-ra módosítja.

A kiértékeléshez először kódot generál a kapott azonosítóhoz (*Spreadsheet.CodeGeneration.generateCode*). Kódgenerálási hiba esetén logolja a hibát jelző üzenetet.

Ha sikeres volt a kódgenerálás, akkor a kapott sorrendben kiértékeli az utasításokat a GHCi-ben. Ehhez először törli a korábbi GHCi bindingokat az *Eval.Ghci.loadModules* akcióval. Erre azért van szükség, hogy ha egy cellához rendelt változót nem sikerült kiszámítani (pl. típushiba miatt), akkor a leszármazott cellához ne legyen felhasználható egy korábbi kiértékeléskor kiszámított, elavult érték. Ezután kiértékeli a külső dependenciákat, azaz azon cellákat, amelyek nem függenek a megváltoztatott cellától, de függőségei valamely a megváltoztatott cellától függő cellának. (Ez a kódgenerálás által adott első lista.)

Ezután következik a második lista utasításainak kiértékelése. Minden utasítás esetén először végrehajtja az értékadást. Ha az eredmény hiba, a kiértékelés eredménye hiba. Ha az eredmény nem hiba, akkor lekérdezi a kiértékelte változót. A kódgenerálás garanciát ad arra, hogy egy cella mindig a függőségei után kerül kiértékelésre.

Az összegyűjtött eredmények ezután cacheltetnek (*Spreadsheet.Interface.cacheCell*). A betöltött fájl állapota *Modified* lesz.

updateView :: ReaderT Env IO

A számolótábla állapota alapján frissíti a cellákban megjelenített szöveget. Ehhez felhasználja a *Spreadsheet.Interface.getCellText* függvényt. A cellatartalmak lekérdezése a GUI *entryKeys* komponensében tárolt kulcsok alapján történik.

3.6.5. App.Setup.CommandLine

A modul a parancssor (*Entry*) *onEntryActivate* eseményéhez (enter billentyű leütése) rendel eseménykezelőt. Az esemény hatására bekövetkező viselkedés a következő:

1. A parancssorban lévő szöveg parancsként parseoltatik.
2. Ha *GHCi* parancsként értelmezhető (a parseolás eredménye *ClGhci cmd*), akkor végrehajtatik a *GHCi* parancs, és az eredmény logolásra kerül.
3. Ha *cp* vagy *mv* parancsként értelmezhető (*ClCp ls* vagy *ClMv ls*), akkor a kapott cellaazonosító-párok alapján végrehajtódik a másolás/mozgatás. Az akció a kapott lista szerinti sorrendben hajtódik végre. Először lekérdezzük a forráscella kódja, majd a relatív hivatkozások eltolásra kerülnek az *Eval.CommandLine.shiftCode* függvénnyel, végül a célcella állapota ez az "eltolt kód" lesz. Az összes akció végrehajtása után frissül a nézet.
4. Ismeretlen parancs esetén logolásra kerül a hiba.
5. A parancsorból eltűnik a szöveg.

3.6.6. App.Setup.Editor

A modul a kódszerkesztő (*Entry*) *onFocusOut* (fókusz elvesztése), *onFocusIn* (fókusz megszerzése) és *onEntryActivate* eseményeihez rendel eseménykezelőt.

A fókusz elvesztésekor és az enter leütésekor az alábbi viselkedés következik be:

1. Amennyiben nem volt kiválasztva cella a táblában, nem történik semmi.
2. Amennyiben volt kiválasztott cella, a cella állapota módosításra kerül a bevitt szöveg alapján (*Spreadsheet.Interface.setCellState*). Ha ezzel megváltozott a cella állapota, végrehajtódik egy kiértékelés, aminek a gyökere a megváltoztatott cella. (*App.Setup.Global.evalAndSet*).
3. Frissül a táblanézet ((*App.Setup.Global.updateView*))

A fókusz megszerzésekor amennyiben volt kijelölt cella, úgy annak a legutóbb megadott kódja jelenik meg a szerkesztőben.

3.6.7. App.Setup.MenuBar

A modul eseménykezelőket rendel a menüsor gombjainak (*Button*) *onClicked* eseményéhez.

A funkciók megvalósítása dialógusok segítségével történik (*FileChooserDialog* és *Dialog*). A "Modules" és "Search paths" gombokhoz egy közös akció tartozik, ami paraméterezhető a változtatandó beállítással (*ModuleActionType*).

A modul definiál egy *runAreYouSureDialog* segédakciót, amely egy felugró ablak segítségével kér a felhasználótól egy igen-nem választ ("biztos-e ebben" dialógus). Több eseménykezelő is használja azon esetekben, amikor fennálna a lehetőség, hogy a végrehajtandó akció végrehajtása során elvesznének nem mentett információk.

3.6.8. App.Setup.Table

A modul eseménykezelőket rendel a cellák megjelenítésére szolgáló *Entry*-k *onFocusIn*, *onFocusOut* és *onEntryActivate* eseményeihez, valamint az oszlopok tetején található gombok *onClicked* eseményéhez.

Ha egy cella megszerzi a fókuszt, az editorban megjelenik a kiválasztott cellába legutóbb beütött kód. Emellett az állapotban beállításra kerül az aktuálisan kijelölt cella. (*Spreadsheet.Interface.setSelected*).

A fókuszt elvesztésekor/enter leütésekor ugyanaz történik, mint az editor esetén.

Az oszlopok tetején levő gombokra kattintva lehet állítani a megfelelő oszlop szélességét. Ehhez a gombra kattintva létrejön egy *Dialog*, amiben egy *SpinButton* található. A *SpinButton*-ben található érték a dialógus létrehozásakor az oszlop celláinak jelenlegi szélessége. Miután lefut a dialógus, a *SpinButton*-ben megadott érték lesz az oszlop celláinak új szélessége.

3.7. Tesztelés

3.7.1. A tesztelés folyamata

A tesztelés első fázisában az egyes komponensek kerültek tesztelésre. Az egyes komponensekhez tartozó tesztelési tervekben szerepelnek a tesztelt függvények, és

a függvényenkénti tesztelési szempontok. A tesztesetek kézzel készültek a megadott szempontok alapján.

Az egyes modulokhoz tartozó tesztesetek a *Test.<modulnév>* modulokban található. Mindegyik tesztmodul exportál egy *run<modulnév>Tests :: IO ()* akciót, amely a standard outputra logolja a tesztelés eredményét. **IDE MÉG ÍROK, HA TÉNYLEGESEN KÉSZ LESZNEK A TESZTEK.**

A tesztelés második fázisában került tesztelésre a szoftver tényleges működése. A felhasználói tesztelés kézzel történt a megfelelő szakaszban **IDE KELL SZÁM** felsorolásszerűen leírt szempontok alapján. **Vastag betűvel jelzem a nem teljesített teszteseteket.**

3.7.2. Test.Spreadsheet.*

Test.Spreadsheet.Parser

A *Spreadsheet.Parser* modulból az általa egyedülként exportált *rep* függvény teszteltetett. A tesztelés input-elvárt output párok segítségével történt, és az alább leírt eseteket fedte le:

1. ""
2. Számliterálok – egész, tizedestört (. előtti és utáni rész nélkül is), pozitív és negatív számok
3. Formulák – referencia nélkül, egyszerű és többszörös referenciák, csak referenciát tartalmazó formulák, formula elején/végén levő referencia, referencia kisbetűvel és nagybetűvel is.
4. Szintaxis hibák – "=", bezáratlan §, §-on belüli hibás szintaxis.
5. Stringliterálok

Test.Spreadsheet.Interface

A *Spreadsheet.Interface* modul által exportált függvények közül csak a *setCellState* és *cacheCell* függvények lettek tesztelve. A további függvények helyes működésének ellenőrzésére elégséges elvégezni a felhasználói teszteket. Ezek többnyire triviálisan implementált getter/setter jellegű függvények.

A tesztesetek műveletek sorozatával transzformálnak egy kezdeti üres számolótáblát. Minden művelet végrehajtása után ellenőrzésre kerül, hogy a művelet megőrizte-e a 3.1. táblázatban leírt típusinvariánst. A nem tesztelt függvények esetén az invariáns megőrzése triviálisan teljesül. Alább felsorolásszerűen szerepelnek a *setCellState* függvényhez tartozó tesztelési szempontok:

1. Értékek, hivatkozást nem tartalmazó formulák hozzáadása, módosítás (*setSimple* teszteset)
2. Formulák hozzáadása, körmentesen (*refsNoCycle* teszteset)
3. 1, 2 és több hosszú körök előidézésének megkísérlése. (*tryCycle** tesztesetek)
4. Nem hivatkozott cella üressé tétele, ekkor a megfelelő csúcsnak el kell tűnnie a gráfból. (*makeEmptyNoRef* teszteset)
5. Az üressé tett, nem hivatkozott cella egyedülként hivatkozik legalább egy üres cellára. Ekkor a hivatkozott üres cella is eltűnik a gráfból. (*makeEmptyNoRef* teszteset)
6. Hivatkozott cella üressé tétele.
7. Cella módosításakor a cellát reprezentáló csúcsba bemenő és kimenő élek ellenőrzése (több tesztesetben is).

3.7.3. Test.Eval.CodeGeneration

Az *Eval* komponensből csak az *Eval.CodeGeneration* modulhoz készültek egysegtesztek.

A parancssori parancsok parseolására szolgáló *Eval.CommandLine* tesztelésére nincs szükség, mivel a referenciák feloldása már a *Spreadsheet.Parser* modulban teszteltetett, az implementáció maradéka pedig egyszerű, az esetleges hibák pedig jól megtervezett felhasználói tesztekkel szűrhetők.

Az *Eval.Ghci* és *Eval.EvalMain* modulok funkcionalitása a kiértékeléshez kapcsolódik. Az ezen két modulban definiált akciók helyes működése szintén a felhasználói tesztekkel ellenőrizhető. (Csak a kiértékelés eredményét kell figyelni, és a háttérben futtatott folyamatokat monitorozni.)

A tesztelés során az *Eval.CodeGeneration* modul által egyedülként exportált *generateCode* függvény teszteltetett. A tesztesetek először felépítenek egy számolótáblát (*setCellState* és *cacheCell* segítségével), majd kódot generálnak a megadott

cellákhoz, és az eredményt összehasonlítják az elvárt eredménnyel. A tesztesetek manuálisan lettek előállítva az alábbi szempontok alapján:

1. Kódgenerálás értékhez, amelytől nem függ cella.
2. Elvégezhető kódgenerálás formulához, amelytől nem függ cella.
3. Elvégezhető kódgenerálás értékhez/formulához, amelytől függnének cellák.
Különböző gráftípusok: egyenes, fa, általános irányítatlan körmentes gráf. A formulának lehet külső, formula függősége (ekkor a teszteset felépítésekor azt cachelni kell.)
4. Egyszerű- és listahivatkozások.
5. Üres cellára is hivatkozó formula.
6. Cella hiányzó függőséggel (nem cachelt formula).
7. Parsolási hibás cella.

3.7.4. Felhasználói tesztek

1. Az alkalmazás indítása
 - A futtatható állomány futtatásakor elindul a szoftver. Megjelenik az üres számológéptábla. A fejléc szövege "*new file".
 - A háttérben elindul a GHCi folyamat.
 - Indításkor betöltésre kerülnek a beállított GHCi modulok és keresési útvonalak. **Nem található konfigurációs fájl esetén egy hibaüzenet kerül megjelenítésre.**
2. Az alkalmazás leállítása
 - A bezárás gombra kattintva a szoftver futása megáll.
 - **Mentetlen munka esetén felugrik a "biztos-e ebben" dialógus.**
 - A szoftver leállítása után leállnak a háttérben futó GHCi folyamatok.
 - Leállításkor mentésre kerülnek a beállított GHCi modulok és keresési útvonalak. (./modules.sanyi)
3. Új tábla létrehozása
 - A megfelelő gombra kattintva, illetve az "Alt-N" billentyűkombináció leütésére elkezdődik egy új tábla létrehozása.

- Mentetlen munka esetén felugrik a "biztos-e ebben" dialógus.
- **Új tábla létrehozásakor beállíthatók a tábla dimenziói. Nagy táblaméret esetén lehet görgetni az ablaknézetet.**
- A létrejött új tábla üres.

4. Tábla mentése és betöltése

- A megfelelő gombokra kattintva, illetve az "Alt-S"/"Alt-L" billentyűkombinációk hatására megjelenik a mentési/betöltési dialógus.
- Mentés után a felhasználó által megadott helyen létrejön egy fájl. A fájl neve <megadott fájlnev>.fsandor.
- Betöltéskor mentetlen munka esetén felugrik a "biztos-e ebben" dialógus.
- Betöltéskor helyesen töltenek be az üres cellák, értéket tartalmazó cellák, sikeresen kiértékelt formulák és a hibás cellák is.

5. Modulok és keresési útvonalak

- A megfelelő gombokra kattintva megjelenik a GHCi-ba betöltendő modulok beállítására szolgáló dialógus/a keresési útvonalak beállítására szolgáló dialógus.
- A módosítások hatása egyből érvényesül. (Hozzáadás és eltávolítás esetén is.)

6. Parancssor *cp* és *mv* parancsainak tesztelése.

- Üres cellatartomány megadása.
- Formulák, értékek, üres és hibás cellák másolása/mozgatása.
- Esetek, amikor a kiinduló- és céltartomány részben/teljesen egybeesik.
- **Mi történik, ha kilóg a céltartomány?**
- Körkörös hivatkozás előidézése a céltartományban.

7. Parancssor további tesztelése

- *g* parancs tesztelése
- Érvénytelen parancs esetén megjelenik egy log üzenet.

8. Cellák tesztelése – helyes cella

- Értékek beírása cellákba (szám/string).
- Hivatkozást nem tartalmazó formula beírása.
- Helyes, hivatkozást tartalmazó kifejezések beírása, egyszerű- és listahivatkozás, többelemű hivatkozási láncok létrehozása.
- Hivatkozott cella módosítása.

9. Körfigyelés

- 1, 2 és több hosszú körkörös hivatkozási lánc létrehozása.
- Megfelelő log üzenet megjelenésének figyelése.
- Körkörös hivatkozás megszüntetése.

10. Parse hibák előidézése

- A hibás cellában megjelenik az "FNoParse" szöveg
- A hibás cellára hivatkozó cellákban megjelenik az "FNoCache" szöveg.
- **Egy korábban helyes cellát parseolási hibásra átírva a hibás cellára hivatkozó cellák állapota megváltozik.** A parseolási hibát megszüntetve a leszármazott cellák hibája is megszűnik. (amennyiben a beírt kód helyes).

11. GHCi típus- és futásidejű hibák előidézése

- A hibás cellában és a belőle leszármazó cellákban megjelenik az "FGhciError"/"FNoCache" szöveg. **A hibaüzenetek még nem túl jók a leszármazott cellában.**
- Egy korábban helyes cellát GHCi hibásra átírva a hibás cellára hivatkozó cellák állapota megváltozik. A hibát megszüntetve a leszármazott cellák hibája is megszűnik. (amennyiben a beírt kód helyes).

12. GHCi időtúllépési hibák előidézése

- A hibás cellában és a belőle leszármazó cellákban megjelenik az "FTimeoutError"/"FNoCache"/"FGhciError" szöveg. **A hibaüzenetek még nem túl jók a leszármazott cellában.**

- Egy korábban helyes cellát időtúllépés miatt hibásra átírva a hibás cellára hivatkozó helyes cellák állapota megváltozik. A hibát megszüntetve a leszármazott cellák hibája is megszűnik (amennyiben a beírt kód helyes).
- Az időtúllépés kezelése után nem szivárog ki a leállítandó korábbi GHCi folyamatok egyike sem.

13. A kódszerkesztő tesztelése

- A 8-12. tesztek elvégzése a kódszerkesztőt (is) használva, nemcsak közvetlenül a cellákba írva a kódot.
- Új fájl megnyitásakor cella kijelölése előtt a kódszerkesztőbe írásnak nincs hatása.

3.7.5. A tesztelés tanulságai

A tesztelés segítségével számos apró hibát sikerült feltárni. Ezek többnyire elírásokból, vagy olyan szélsőséges esetekből adódtak, amelyekre nem gondoltam a tervezési fázis során. A 3.7.1-3.7.4 alfejezetekben leírt tesztelési szempontok alapján a program átment a teszteken, biztosítja az elvárt funkcionalitást.

Az egységtesztelés során a legtöbb problémát az *Eval.CodeGeneration* modul okozta. A jelenlegi implementáció bár működik a szélsőséges esetekre is, kissé kaotikus. Itt indokolt lehet a modul újratervezése. (Az újratervezés mögött más érvek is szólnak, erről részletesebben a szoftverrel kapcsolatos távlati terveket részletező 4.2 alfejezetben esik szó.)

A felhasználói tesztek fő tanulsága, hogy a felület még nem kellően felhasználóbarát. Számos funkció hiányzik, ami kényelmesebbé tenné a felhasználó munkáját (pl. formázással kapcsolatos beállítások, beszédes hibaüzenetek). (Erről is részletesebben esik szó a 4.2 alfejezetben.) Ugyanakkor a megvalósított funkciók intuitívan használhatók, így egy jó alapot biztosítanak a későbbiekben megtervezendő, összetettebb felhasználói felülethez.

4. fejezet

Összegzés

4.0.1. Ami megvalósult

Sikerült implementálni egy táblázatkezelő szoftver első verzióját. A szoftver architektúrája lehetővé teszi a későbbi bővítést, vagy az egyes komponensek lecserélését.

Az alkalmazásban közvetlenül használhatók Haskell nyelven megírt modulok. Ezek segítségével lehetséges számításokat végezni a bevitt szöveges és numerikus adatokon. A munkát támogatja néhány egyszerű parancs (cellablokkok mozgatása és másolása, GHCi utasítások közvetlen kiértékelése).

Lehetőség van számolótáblákat menteni és betölteni, az alkalmazás beállításai (GHCi-ba betöltendő modulok és keresési útvonalak) is automatikusan mentésre kerülnek az alkalmazás bezárásakor.

Az alkalmazás felhasználói felülete ugyanakkor még kezdetleges, és további fejlesztésre szorul. A fő funkciók használhatók, de a használat körülményes, a szoftver jelenlegi verziójának gyakorlati használata nem életszerű.

4.0.2. Távlati tervek

A bevezetőben szerepelt az a gondolat, hogy a táblázatkezelő lényegében egy interfészt biztosít egy funkcionális nyelvhez. Ez azonban még csak korlátozottan valósul meg, mivel a cellákban csak numerikus és szöveges adatok tárolhatók. Ezt a későbbiekben célszerű lenne oly módon kiegészíteni, hogy a cellákban tetszőleges (akár *Show* példánnyal nem rendelkező) típusú értéket lehessen tárolni. Emellett

azt is meg lehetne valósítani, hogy a cellák értéke ténylegesen a háttérben futtatott GHCi példányban legyen tárolva, és ne csak az alkalmazás által használt gráfrepresentációban. Így az alkalmazás tényleg magához a GHCi-hoz lenne biztosítana egy interfészt, nem pedig csak a GHCi segítségével végezne számításokat.

A felhasználói felülettel kapcsolatban számos funkció megvalósítása szükséges. Jelenleg teljesen hiányoznak a formátumra vonatkozó beállítások (pl. betűszín, cellák szélessége). Emellett hasznos lenne, ha a mozgatáshoz és másoláshoz lehetne használni az Excelben megszokott megoldást (cellablokk kijelölése és billentyűkombinációk használata). Lehetne támogatást implementálni ábrák megjelenítéséhez, kihasználva, hogy a Haskellben implementált grafikus csomagok mind elérhetők az alkalmazáson belül. Ezek mellett még egy fontos cél lehet a kódírás segítése hasznosabb hibaüzenetekkel, és egy többsoros, a Haskell nyelvhez igazított szövegszerkesztővel.

Táblázatok jegyzéke

3.1. Egy <i>Formula</i> lehetséges állapotai	23
3.2. A <i>Spreadsheet.Interface</i> által exportált függvények	26
3.3. Az <i>Eval.Ghci</i> által exportált függvények	31

Forráskódjegyzék

2.1. Az \mathbb{E} kombinátor	10
2.2. Az onJusts kombinátor	11
2.3. LFun és NLFun	11
3.1. Az Env típus	16
3.2. A Gui típus	17
3.3. A Spreadsheet típus	20
3.4. A <i>ClCommand</i> típus	32
3.5. Az eseménykezelők hozzárendelése	35