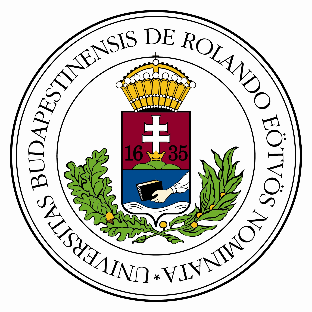
**Eötvös Loránd Tudományegyetem**

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

**P4 programok helyességének ellenőrzése bővíthető szabályrendszer alapján**

Szerző:

Nagy Rebeka

Programtervező Informatikus BSc

Témavezető:

Tóth Gabriella

Doktorandusz

Budapest, 2020

Tartalomjegyzék

[1. Bevezetés 4](#_Toc40641805)

[2. Felhasználói dokumentáció 5](#_Toc40641806)

[3. Fejlesztői dokumentáció 5](#_Toc40641807)

[3.1. Szabályrendszer 5](#_Toc40641808)

[3.2. Elemzés 5](#_Toc40641809)

[3.3. Hiba észlelés 5](#_Toc40641810)

[Parser.hs 6](#_Toc40641811)

[Preparation.hs 9](#_Toc40641812)

[Verification.hs 11](#_Toc40641813)

[3.4. Felhasználói felület 12](#_Toc40641814)

# 1. Bevezetés

Egy szakdolgozat elkészítése remek lehetőség arra, hogy az egyetemi évek alatt megszerzett tudást kamatoztassuk, valamint, hogy új eszközöket és módszereket sajátítsunk el.

Mindig is közel éreztem magamhoz a funkcionális nyelveket, így a projektem modelljét is abban szerettem volna implementálni. A felhasználó felületek elkészítésére sok módszert tanultam különböző kurzusok alkalmával, ezek közül terveztem választani a szakdolgozatomhoz is egy architektúrát.

A P4 programozázi nyelvvel néhány projektmunka alkalmával ismerkedtem meg.

Ez egy napjainkban fontos, hálózati csomagok feldolgozására szolgáló eszközök programozására készített programozási nyelv. Az ellenőrzése és a tesztelése ebből kifolyólag nehéz, a fejlesztők könnyen hibát tudnak véteni a fejlesztés során.

A témám tehát egy olyan komplex szoftver elkészítése, amely képes ellenőrizni ezeket a programokat, és valamilyen visszajelzést adni arról, hogy mely részek okozhatnak nem elvárt viselkedést.

A szakdolgozatom ez alapján két nagyobb részre osztható: az elemző részre, és megjelenítésre.

Előbbihez tartozik a P4 szoftver szintaktikus elemzése, az előkészítése, végül az ellenőrzése adott szabályrendszer alapján. Ez a rész Haskell funkcionális nyelven írodott, így lehetőségem volt elmélyíteni a nyelvről megszerzett tudásomat, új könyvtárakat, implementálási módszereket és nyelvi szerkezeteket ismerhettem meg.

Utóbbit az egyetemen elsajátított MVVM architektúrában építettem fel, ahol az üzleti logika az ellenőrző résztől kapott infomációt dolgozza fel.

# 2. Felhasználói dokumentáció

....

# 3. Fejlesztői dokumentáció

## 3.1. Szabályrendszer

A verifikáció működésének a Tóth Gabriella és Tejfel Máté által szerzett cikk[[1]](#endnote-1) az alapja. Ez egy axiomatikus szemantikához hasonló szabályrendszer, amelynek összetettebb a környezeti struktúrája, és mellékfeltételekkel van kiegészítve.

Ezeket a feltételeket tudja a felhasználó módosítani, ezzel bővítve a programon végzett ellenőrzéseket.

....

## 3.2. Elemzés

A dolgozat célja egy összetett verifikációs szoftver elkészítése, amely képes egy szintaktikailag helyes, leforduló P4 résznyelvbeli programot ellenőrizni, egy adott szabályrendszer alapján.

....

A programnak könnyen használhatónak, és átláthatónak kell lennie. Fontos a helyes és gyors működés is.

Mivel az ellenőrzés és a felhasználói felület két különböző nyelven és keretrendszerben van implementálva, így a köztük lévő gördülékeny kommunikációt is meg kell valósítani.

....

## 3.3. Hiba észlelés

Az ellenőrzés két szöveget kap a felhasználói felülettől, ezeket dolgozza fel, majd a belőlük kiszámított fontos információkat küldi vissza, szintén szövegként.

A megkapott stringek közül az egyik maga a P4 program, a másik a felhasználó által megadott mellékfeltételek.

### Parser.hs

Az első lépésben a program feldolgozása történik, amelyet a Text.ParserCombinators.Parsec[[2]](#endnote-2) könyvtár, valamint egy egyszerűbb példa[[3]](#endnote-3) segítségével oldottam meg.

A nyelvtani elemzésehez szükség van a nyelv definíciójára, vagyis a kommentek jelölésének, a változónevek leírásának, az operátorok és a kulcsszavak meghatározására. Ezzel az is definiálásra kerül, hogy a P4 milyen résznyelvével foglalkozik az ellenőrzés.

Implementálásra kerültek még bizonyos tokenek, amelyek magát az elemzést és annak olvashatóságát segítik.

#### Bevezetett adattípusok

##### Statement

A szintaktikus elemző célja, hogy a megkapott P4 programot megfelelő részekre bontsa, amelyet az előkészítő folyamat már könnyen át tud alakítani a hibák észleléséhez szükséges típusokra.

Ezeket a részeket a *Statement* típus foglalja magába. Az elemzés célja egy *Statement* listává alakítani a megkapott programot.

##### Variable

A *Variable* típus segítségével írjuk le mind a mezőket és fejléceket, mind pedig a táblák kulcsait és a *control* függvények paramétereiben szereplő változókat.

##### FunctionExpression

A programban történő függvényhívásokat *FunctionExpression*-ként írjuk le, melyek a *FuncExpr* konstruktorral alakíthatóak át *Statement*-re.

Ezek a függvényhívásokhoz tartózó változókból, valamint a függvény fajtájából tevődnek össze.

##### ArithmeticExpression

Az aritmetikai kifejezések leírására szolgáló típus. Ezt a *ParserAssignment* konstruktorral használjuk, amely a programban lévő értékadásokat írja le. Ehhez tartozik egy *string*, amely az egyenlőség bal oldalán szereplő változó nevét takarja, valamint egy *ArithmeticExpression*, amelyben az értékadás bal oldalán szereplő kifejezést tároljuk. Ez állhat egyetlen változóból, vagy bármilyen aritmetikai kifejezésből (negálás, összeadás, kivonás, szorzás, osztás).

##### BoolExpression

A logikai kifejezéseket írhatjuk le vele. Ezek a résznyelvben az elágazások feltételében szerepelnek. *Statement*-re a *BoolExpr* konstruktorral tudjuk hozni.

Állhat logikai konstansokból(igaz, hamis), valamint ezek és változók valamilyen logikai kifejezéséből(negálás, konjukció, diszjunkció, egyenlő, nem egyenlő, kisebb, nagyobb).

Egy P4 specifikus függvény, a *.isValid()* is ide tartozik.

#### Függvények

Az szintaktikus elemzőt képező függvények három nagyobb részre oszthatóak, amelyek lejjebb kerülnek kifejtésre. A *Parsec* könyvtárból származó *Parser* függvényparaméter segítségével tudjuk a függvényeket olyan alakra hozni, amellyel szöveget tudunk feldarabolni, mintát illeszteni rá és meghatározni hogy a programnak pontosan milyen alkotóeleme. Ezek a szeletelések mindig a legnagyobb illesztést hajtják végre, így cél volt az, hogy az elemzés során mindig egyértelmű legyen, hogy mi a következő lépés.

##### Fejléc-Struct elemző függvények

Ezek a függvények a program általában elején definiált fejlécekből és struktúrákból szedi ki a számunkra hasznos információkat, vagyis, maga az elnevezés melett, a mezők neveit. A header és a struct kulcsszavak megtalálásakor indul el az ehhez tartozó feldolgozás. *ParserHeader* és *ParserStruct* konstruktorral kerülnek tárolásra.

##### Parser elemző függvények

Ez egy P4 specifikus része a programnak. Az elemzés során ebből a részből fogjuk kinyerni a kezdő állapotokat. A *Parser* konstruktor paramétere egy lista, amelyben felsorolásra kerül minden abban található *state* kulcsszóval kezdődő függvény.

Ezek az állítások tartalmazhatják egy fejléc kibontását(*.extract())*, egy másik *state*-re való ugrást(*transition*), vagy pedig egy elágazást(*transition select*). Egy P4 program futásakor az elágazás során egy bizonyos értéket vizsgálva dől el, hogy melyik állítással folytatódik a *parser*. A hiba észlelés számításakor viszont ez az érték számunkra nem releváns, az elemző az összes lehetséges végkimenetelt vizsgálja, és a kezdő állapotok közé felveszi.

##### Control elemző függvények

A control kulcsszóval kezdődő függvények elemzésére használt függvények.

Ezek a benne található *action*, *table* és *apply* kulcsszavak mentén ismerik fel a definiált akciókat, táblákat valamint az alkalmazandó programot.

Ezek a *ParserAction*, *ParserTable* és *Apply* konstruktorokkal jönnek létre.

Az akcióknak a neve, és a hozzá tartozó szekvenciába rendezett értékadások és függvényhívások kerülnek rögzítésre.

A tábláknál fontos információ a kulcsok, valamint a hozzájuk rendelt akciók nevei.

Az *apply* függvényben általában elágazások(*if-else*), akció- és táblahívások, valamint *.emit(fejléc)* függvényhívások szerepelnek. Ezek is elmentésre kerülnek, egy listában.

##### Kifejezéseket elemző függvények

Az aritmetikai és logikai kifejezések, valamint a függvényhívások elemzéséhez szükséges függvények.

Mindegyik a hozzájuk tartozó, nevükkel megegyező adattípusokat ismeri fel, és bontja fel a megfelelő változóra és operátorra. Ebben az esetben a függvényhívások operátorként vannak definiálva.

Az operátorok esetében az is rögzítve van, hogy egy kifejezés elején, közepén vagy végén található-e meg. A jól ismert összeadás jel(+) például a kifejezés belsejében, de a függvényhívások, mint a fejléc validitásának beállítására használt .setValid(), a kifejezések végén fog előfordulni. Ez az információ a mintaillesztést segíti.

### Preparation.hs

Miután a szintaktikai elemző a szükséges információkat kiszedte a programból, átadja ezt a *Statement* listát az előkészítő lépésnek.

Ez rekurzív módon halad végig a listán, és minden egyes elemet megfelelő alakra alakít át. A fejlécekből és azok mezőiből hozza létre a kezdő- és végállapotokat, az *Apply*-ból és az ahhoz tartozó akciókból és táblákból pedig az elemzendő programot, amelyen végighaladva fognak a kezdőállapotok módosulni.

Továbbbá itt kerül átalakításra a felhasználói felülettől kapott mellékfeltételek szövege egy megfelelő adattípussá.

Ennek a résznek az a célja, hogy egy köztes nyelvre hozzuk a megkapott programunkat, amelyben jelen vannak a P4 specifikus elemek, de mégis könnyebb a későbbiekben elemezni.

#### Bevezetett adattípusok

Az előkészítő folyamat az elemzőben megtalálható adattípusokat konvertálja át az ellenőrző állományban lévő adattípusokra. Ezek a dokumentációban a saját alcímük alatt vannak bővebben taglalva.

A konstruktorok paraméterei általában *string*-ek, vagy további adattípusok *string* paraméterrel. Ebből kifolyólag az átírás nem okoz bonyolult konverziókat.

A Parser és a Verification kódállományokban lévő adattípusok konstruktorainak nevei nagyon hasonlóak, ezzel is könnyítve az átalakításokat.

#### Függvények

##### mainConversion

Az a rekurzív függvény, amely paraméterbeli listának az elemeihez a megfelelő kisebb átalakító függvényt rendeli. Ezt a konstruktor alapján dönti el.

Paraméterként megkapja az elemző által készített *Statement* listát, egy üres környezet listát a kezdeti állapotoknak, egy üres környezetet a végállapotoknak, valamint három üres *Program* listát, amelyben rendre az akciókat, a táblákat és az apply-ból kinyert programokat fogja tárolni.

Ha az átalakítás során bármely résznél hiba merülne fel, akkor a rekurzió leáll, és a paramétereknek megfelelő *error* konstruktorokat adja vissza. Ha nincs hiba, a rekurzió végeztél végleges alakra hozza a környezeteket.

A kezdőállapotok esetében ez azt jelenti, hogy az alap *Environment* típusról *IdEnvironment* adattípusra, mely az előzőt egészíti ki egy azonosítóval, mely a megjelenítésnél létrehozott gráfnál lesz fontos, valamint egy *EnvironmentType* típusú adattaggal, aminek értéke lehet Match, NoMatch és Stuck. Ezek rendre azt jelelök, hogy a környezet sikeresen elért a számítás végére és megegyezik a végállapotok valamelyikével, vagy elért a végére, de egyik végállapottal sem azonos, vagy a számítás során elakadt.

A végállapotnál kettő környezet kerül létrehozásra. Az egyik a program *deparser* részében megtalálható *.emit(fejléc)* függvények által, a másik a számítás végén kerül hozzáadásra. Ez egy alapértelmezett végállapot, amelyben a *drop* értéke*Valid*, így minden más fejléc és mező *Undefined* lesz.

##### Fejléceket átalakító függvények

Ha a *Statement* listában a *mainConversion* függvény *ParserHeader* vagy *ParserStruct* konstruktorokat talál, melyek a fejlécek nevét, valamint a hozzájuk tartozó mezők neveinek listáját tartalmazzák, akkor a kezdő- és végállapotokat tároló paramétereit a *headerConversion* függvénynek adja át.

Ez fogja a fejlécet a megfelelő alakra hozva beépíteni a környezetekbe. A fejléc nevéhez rendel egy validitást, amely a kezdőkörnyezetek esetében *Invalid*, a végállapotok környezeteiben pedig *Undefined*, majd ugyenezt végrehajta minden mezőn is, a fejléccel azonos validitással a *fieldsConversion* függvény segítségével.

##### parserConversion

*Parser* konstruktor esetén a *mainConversion* a kezdőállapotokat tartalmazó paraméterével hívja meg a *parserConversion* függvényt.

Ez a listaelem az elemzés során egy state elemeket tartalmazó listává lett átalakítva, így a feldolgozáskor ezen kisebb részeket haladunk végig.

Első lépésként megkeresi a *start* elnevezésű elemet, amely egy P4 program futtatásakor minden esetben legelőször fut le. Ha nem talál ilyet, akkor egy hibát jelző adattípussal tér vissza, amely a későbbiek során egy hibaüzenetté lesz átalakítva, így továbbítva a felhasználó felé.

Ha sikeresen megtalálta a kezdő elemet, akkor megkezdődik az elemek feldolgozása a *stateConversion* függvény segítségével.

Egy state utasításai közül számunkra három lényeges. Ezek a *transition*, amely a meghatározza, hogy mely elemen folytatódik a feldolgozás, a *transition select*, amely az előzőhöz hasonló, csak feltételes elágazással, valamint az *.extract()* függvény, amely egy fejlécre meghívva kibontja azt. Ennek reprezentálása a fejléc és annak mezőihez tartozó validitások megváltoztatása *Valid*-ra.

A *stateConversion* minden esetben megvizsgálja, hogy a state tartalmaz-e kicsomagoló függvényt. Ha igen, akkor azt alkalmazza a környezetre, ha nem, akkor a környezet változatlanul marad.

A feldolgozás folyatása pedig a *transition* és *transition select* utasítások szerint halad. Ez a kettő egyszerre nem szerepel, tehát vagy egyértelműen halad a feldolgozás, vagy pedig elágazik. Ha egyik sem található meg az elemben, akkor ott a rekurzió megáll, miután az *.extract()* függvények alkalmazásra kerültek, ha voltak.

A *transition* esetében a kezdő környezetek száma változatlan marad, a *stateConversion* újrahívása a megadott nevű state elemmel folytatódik.

A *transition select* feltételeket és *state* név párokat tartalmaz. A feldolgozás során a feltételekkel nem foglalkozik, az elágazás minden lehetséges ágán végighalad, így a környezetek mennyisége ezek számával arányosan növekszik.

Miután a függvény a fent leírt módon végighaladt a *Parser* összes elemén, a kezdőállapotokat leíró paraméter az összes lehetséges környezetet tartalmazni fogja, amellyel a P4 program lefutásra kerülhet. Az ellenőrzés során a program az ebben lévő környezeteken fogja a számításokat elvégezni.

##### controlConversion

Ha a *Control* konstruktor a következő a *mainConversion* futása során, akkor az erre megírt *controlConversion* függvényt fogja meghívni. Ennek a kezdő – és végállapotokat, az akciókat, táblákat, és a programot tartalmazó paramétereket fogja átadni.

A feldolgozás ebben az esetben is a lista elemein fog végighaladni.

A *ParserAction* esetében az *actionConversion* kerül meghívásra. Egy akciónak van neve, valamint utasításai, ezek a megfelelő átírás után szekvenciálisan vannak eltárolva. Ezután az átalakítás után az akciókat tartalmazó paraméterlista végére fűzzük, mint új akciót.

A *tableConversion* függvényt hívjuk meg, ha *ParserTable* típusú a következő elem. Az átírás során a tábla neve és kulcsai mellett tárolásra kerülnek a táblához tartozó akciók. Ezeket az előzőleg említett akció listából másoljuk át a tábla paramétereihez.

Az *Apply* tartalmazza a programot, ha ilyen konstruktort talál a *controlConversion* függvény, akkor az *applyConversion* függvényt fogja meghívni. Ezáltal lesz majd az a program létrehozva, amin végighaladva az ellenőrzés végrehajtódik.

Az Apply paraméterében megkapott programot alakítja át, majd adja vissza úgy, hogy minden akcióhívás alkalmával beilleszti az akciót, a táblahívások során a táblát, annak akcióival együtt. Az elágazások és szekvenciák a programnak megfelelően megmaradnak.

Ha a control kulcsszavas függvény egy *deparser*, akkor ott az *Apply* konstruktor paraméterében *.emit()* függvényhívások szerepelnek. Ezért, minden esetben, amikor *Apply* konstruktort talál, a *controlConversion* egy másik függvényt is meghív a végállapotokat tartalmazó paraméterével.

Ez az *emitConversion* függvény, amely minden *.emit()* hívásban lévő fejléc esetében *Valid*-ra állítja annak minden értékét. Így kerül létrehozásra, a *drop* *Valid* környezet mellett, a másik végállapot. Utóbbi tehát az átalakítás közben, míg előbbi majd csak az átalakítások végén jön létre.

##### sideConditionConversion

Fontos még a mellékfeltételek átalakítása, melyet a program felhasználói felület részétől kapunk meg, számok sorozataként.

A sorozat minden eleme egy programfüggvényhez tartózó mellékfeltételt jelöl, míg az értéke azt, hogy van-e ellenőrzés, és ha igen, akkor *Valid* vagy *Invalid* érték esetén teljesül a feltétel. Egy adott programfüggvényhez több mellékfeltétel is tartozik, ez a sorozatban a számokat elválasztó & szimbólummal van elválasztva.

A programfüggvények, amelyekhez rendeljük a mellékfeltételeket rendre az elágazás, tábla, értékadás, fejléc és drop. Ezek az ellenőrzés során fontosak, így a Verification.hs részben vannak bővebben kifejtve.

##### Adattípusokat szöveggé alakító függvények

A még ehhez a részhez tartozó, de kisebb egység a szöveggé alakító függvények. Ezek a Haskellben definiált és használt adattípusokat konvertálja szöveggé, hogy az eredményeket vissza lehessen küldeni a felület rétegnek.

### Verification.hs

Miután az előkészítő folyamat sikeresen a köztes nyelvre hozta a programot, négy fontos adat áll rendelkezésünkre az ellenőrzéshez. A végállapotok, kezdőállapotok, a program és a mellékfeltételek.

A hiba észlelés folyamata a program mentén zajlik. Rekurzív hívásokkal mindig az pillanatnyilag illeszkedő szabályt választjuk ki, és alkalmazzuk azt a környezetekre, ha a mellékfeltételek teljesülnek.

#### Bevezetett adattípusok

##### Validity

A környezetekben szereplő fejlécek és azokhoz tartozó mezők érvényességének jelöléséhez létrehozott adattípus. Értéke lehet Valid, ha érvényes, Invalid, ha nem az, Undefined, ha nem meghatározott a fejléc vagy mező és None, ami a mellékfeltételekhez szükséges, ha nincs ellenőrzés. Az egyenlőség vizsgálat során az Undefined validitás minden mással egyenlő.

##### Field

A fejlécekhez tartozó mezők típusa. Egy név és validitás párból áll.

##### Header

A fejléc típusa. A nevén és a validitásán kívül a hozzá tartozó mezők listájából épül fel.

##### Environment

A környezet adattípusa lényegében fejlécek listája. Konstruktora az *Env*. Minden esetben tartalmaz egy *drop* nevű fejlécet, amelynek mező listái üresek.

Az ellenőrzés során csak a végállapotok lesznek ilyen típusúak, mivel azok számossága mindig kettő lesz. A deparser által kiszámított, valamint a valid drop érték esetén meghatározott környezetek. A kezdő, valamint azokból kiszámított környezetek azonosítóval vannak ellátva.

Másik konstruktora az *EnvError*, mely hiba észlelése esetén egyértelműsítheti hogy a számítás során a környezetek alakítása során történt hiba. Ha az átalakítások vagy átírások során *EnvError* a visszakapott eredmény, az éppen zajló folyamat megáll, és ezt küldi vissza a felhasználói felületnek.

##### IdEnvironment

Az *IdEnvironment* a környezet típus olyan kiegészítése, amely a fejléc lista mellett tartalmaz egy egyedi azonosítót és egy típust, amely a környezet állapotát jelöli.

Az azonosító kezdetben csak egy szám, a kezdőkörnyezeteket sorszámozzuk. Ezután minden lépésben, amikor alkalmazva van egy programfüggvény, akkor annak nevét hozzáfűzzük az azonosító végére. Így a számítás végére minden kiszámolt környezet azonosítójából kinyerhető, hogy milyen lépések során jutottunk el abba az állapotba. Ez a felhasználói felület során a gráf megépítésénél tölt be fontos szerepet.

##### Program

A köztes nyelv reprezentációja. Konstruktorai egy-egy programstruktúrát határoznak meg, a számításhoz szükséges módon leegyszerűsítve.

Az *EmptyProg* a köztes nyelvre való átírás során hasznos. Jelzi, ha a felhasználó olyan programot adott, amely vagy teljesen üres, vagy nincs olyan része, amely alkalmas lenne számítások végzésére.

A *ProgError* az *EnvError* kontruktorhoz hasonlóan hibajelzésre alkalmas, mely nagyobb szintaktikai hibák vagy a résznyelven kívül eső programrészekre figyelmeztet, és szintén leállítja a folyamatokat. Feltételezve van, hogy a beadott program egy leforduló, helyes P4 program, így a hibaészlelés nem a teljes P4 nyelvre kiterjedő.

A *Skip* a standard ugrás programstruktúra, amely során nem történik semmilyen változtatás. *If-else* struktúra üres *else* ágában fordul elő a leggyakrabban, de a szekvencia második részeként is sokszor megjelenik.

A *Seq* a szekvencia konstruktora. Ennek paramétere két *Program*, így egymás után bármilyen programstruktúra előfordulhat.

Az elágazást az *If* konstruktorral van reprezentálva. Paraméterei az feltételben szereplő változók, és két *Program* amik az igaz illetve a hamis ágban vannak.

A P4 specifikus tábla konstruktora a *Table*. A tábla neve, a kulcsainak valamint az akcióinak a listája a hozzá tartozó paraméterek.

Az *Action* az akció konstruktora. Ennek paramétere az akció neve, valamint a szekvenciálisan egymásba ágyazott utasítások. Ezek a következő három valamelyike lehetnek.

Az értékadást *Assignment* konstruktorral reprezentáljuk. Az egyenlőség bal oldalán szereplő változó az első paraméterei, és minden más változó, amely a jobb oldalon szerepel a második paraméterként meghatározott listában van tárolva. Ha az értékadás során a jobb oldalon konstans érték szerepel, akkor ez a lista üres.

A *Drop* konstruktor a *mark\_to\_drop()* függvényhívást jelöli. Nincs paramétere, az egyértelmű *drop* fejléc módosítása miatt.

A fejlécekre meghívható *.setValid()* és *.setInvalid()* függvények reprezentálása a  *SetHeaderValidity* konstruktor. Paraméterei a fejléc neve, valamint a megfelelő validitás érték.

##### SideCondition

A felhasználók szigoríthatják az ellenőrzéseket különböző feltételek megadásával. A programfüggvényekhez több feltétel is rendelhető, ezek mindegyikének helyesnek kell lennie ahhoz, hogy a módosítás végrehajtásra kerüljön.

A mellékfeltételek reprezentációja egy validitás lista ötös. Ezek *None*, *Valid* és *Invalid* értékeket vehetnek fel. Ezek a feltételek konjukcióban állnak egymással, ha bármelyik None értékkel rendelkezik, akkor arra a feltételre nem történik ellenőrzés, és alapértelmezetten igaz lesz.

Az első kételemű lista az elágazás mellékfeltételeit tartalmazza. Az egyik az elágazás feltételében szereplő mezők validitását vizsgálja, míg a másik a fejlécekét.

A második szintén egy kételemű lista a táblák mellékfeltételeihez.

....

##### Rule

A *Rule* egy lambda függvény, amelynek mintaillesztésével választjuk ki a megfelelő programfüggvényt a program pillanatnyi állapotához.

Ez mindig az adott konstruktor alapján kerül kiválasztásra. A szabályokból tehát több van, amely egy *Rules* listában van tárolva.

A lambda felépítése a következő. Paraméterei a számítás során tárolt környezetek (*IdEnvironment*), a program (*Program*), a mellékfeltételek (*SidCondition*) valamint egy szám, amely a környezet azonosítójának egyediségének megtartásához szükséges.

....

#### Függvények

....

## 3.4. Felhasználói felület

1. Cikk linkje [↑](#endnote-ref-1)
2. A könyvtár dokumentációjának linkje

   <https://hackage.haskell.org/package/parsec-3.1.14.0/docs/Text-ParserCombinators-Parsec.html> [↑](#endnote-ref-2)
3. Egyszerű imperatív nyelv elemzése

   <https://wiki.haskell.org/Parsing_a_simple_imperative_language> [↑](#endnote-ref-3)