

# TDK-dolgozat

**Páter-Részeg Attila**



Eötvös Loránd Tudományegyetem  
Informatikai Kar  
Programozási Nyelvek és  
Fordítóprogramok Tanszék

---

# Funkcionális szemantikus program gráf leképezése imperatív szintaxisfára <sup>1</sup>

*Témavezető:*

Tóth Melinda  
egyetemi tanársegéd  
ELTE-IK, PNYF

*Szerző:*

Páter-Részeg Attila  
Programtervező informatikus Bsc  
III. évfolyam

Budapest, 2013

---

<sup>1</sup>Támogatta a TÁMOP-4.2.2/B-10/1-2010-0030

# Tartalomjegyzék

|                                                         |           |
|---------------------------------------------------------|-----------|
| <b>1. Bevezetés</b>                                     | <b>1</b>  |
| <b>2. RefactorErl</b>                                   | <b>3</b>  |
| 2.1. Szemantikus Program Gráf . . . . .                 | 3         |
| 2.1.1. A gráf bemutatása . . . . .                      | 3         |
| 2.1.2. Matematikai reprezentáció . . . . .              | 5         |
| 2.1.3. A gráf sémája . . . . .                          | 6         |
| 2.1.4. Példa . . . . .                                  | 6         |
| <b>3. Set of Software Quality Static Analyzers</b>      | <b>8</b>  |
| 3.1. eCST . . . . .                                     | 8         |
| 3.1.1. A gráf bemutatása . . . . .                      | 9         |
| 3.1.2. Matematikai modell . . . . .                     | 10        |
| 3.1.3. A gráf sémája . . . . .                          | 11        |
| 3.1.4. Példa . . . . .                                  | 11        |
| <b>4. A leképezés</b>                                   | <b>13</b> |
| 4.1. Fogalmak, jelölések . . . . .                      | 13        |
| 4.1.1. Szemantikus Program Gráfok részhalmaza . . . . . | 13        |
| 4.1.2. <i>eCST</i> -k részhalmaza . . . . .             | 13        |
| 4.1.3. Gyerekcsúcsok halmaza . . . . .                  | 13        |
| 4.1.4. Levélcsúcsok az <i>eCST</i> -ben . . . . .       | 14        |
| 4.1.5. Segédfüggvények a mintaillesztésekhez . . . . .  | 14        |
| 4.1.6. Egyéb segédfüggvények . . . . .                  | 16        |
| 4.1.7. A leképezés általános alakja . . . . .           | 16        |
| 4.2. A leképezés konkrét esetei . . . . .               | 16        |

|                                                                                         |           |
|-----------------------------------------------------------------------------------------|-----------|
| 4.2.1. Modulok ( $SG_{module}$ ) . . . . .                                              | 16        |
| 4.2.2. Függvénydefiníció ( $SG_{func}$ ) . . . . .                                      | 18        |
| 4.2.3. Névtelen függvények ( $SG_{fun\_expr}$ ) . . . . .                               | 27        |
| 4.2.4. Függvényhívás, applikáció ( $SG_{application}$ ) . . . . .                       | 28        |
| 4.2.5. Értékadás kifejezés ( $SG_{match\_expr}$ ) . . . . .                             | 29        |
| 4.2.6. Elágazás kifejezések ( $SG_{case\_expr}$ , $SG_{if\_expr}$ ) . . . . .           | 32        |
| 4.2.7. Try kifejezések ( $SG_{try\_expr}$ ) . . . . .                                   | 33        |
| 4.2.8. Listagenerátor kifejezés ( $SG_{list\_comp}$ ) . . . . .                         | 35        |
| 4.2.9. Tuple kifejezés ( $SG_{tuple}$ ) . . . . .                                       | 37        |
| 4.2.10. Lista kifejezés ( $SG_{cons}$ ) . . . . .                                       | 38        |
| 4.2.11. Zárójeles kifejezések ( $SG_{parenthesis}$ ) . . . . .                          | 39        |
| 4.2.12. Blokk kifejezések ( $SG_{block\_expr}$ ) . . . . .                              | 39        |
| 4.2.13. Konstansok és operátorok ( $SG_{infix\_expr}$ , $SG_{prefix\_expr}$ ) . . . . . | 39        |
| 4.3. A tárolt információ . . . . .                                                      | 40        |
| <b>5. Megvalósítás</b>                                                                  | <b>41</b> |
| 5.1. Az átalakító program használata . . . . .                                          | 42        |
| 5.2. Tesztelési esettanulmány . . . . .                                                 | 43        |
| <b>6. Összegzés</b>                                                                     | <b>51</b> |
| 6.1. Áttekintés . . . . .                                                               | 51        |
| 6.2. Továbbfejlesztési lehetőségek . . . . .                                            | 52        |
| <b>Ábrák jegyzéke</b>                                                                   | <b>53</b> |
| <b>Irodalomjegyzék</b>                                                                  | <b>54</b> |

## **Kivonat**

A különböző statikus forráskód elemző eszközök eltérő reprezentációkat használnak az információ tárolására. Adottak a RefactorErl és a SSQSA statikus elemző szoftverek saját funkcionalitással, melyek egyaránt gráfban tárolják az információt, azonban ez a két gráf eltér egymástól. Ha összekapcsolnánk a két eszközt, akkor új funkciókkal bővülne a RefactorErl eszköztára. Az összekapcsolás megvalósításához szükségünk van egy leképezésre a két eszköz által használt gráf között, dolgozatomban ezt definiáltam.

# 1. fejezet

## Bevezetés

Az Erlang egy dinamikusan típusos, funkcionális, elosztott programozási nyelv, amelyet a 80-as években kezdtek el fejleszteni az Ericsson egyik kutatólaboratóriumában. A nyelvet abból a célból alkották meg, hogy könnyen és hatékonyan lehessen telekommunikációs szoftvereket fejleszteni benne. Ennek eredményeképpen a nyelv fő jellemzőjévé vált a konkurens, elosztott, valós idejű, hibatűrő programok írására való lehetőségek biztosítása.

A RefactorErl eszköz Erlang programok statikus elemző és transzformáló eszköze, mely a forráskódot egy Absztrakt Szintaxis Fára épülő Szemantikus Program Gráfban tárolja. A RefactorErl projekt kezdetben Erlang programok refaktorálására koncentrált, később azonban a kutatás és fejlesztés más jellegű feladatokra is kiterjedt, mint például a programmegértés támogatása.

A kódmegértésben fontos szerepet kap a metrika, ami a kód bonyolultságának mérőszáma. Egy magas metrikaértékű kód nehezebben érthető. Ezért megfogalmazhatók olyan kódolási szabályok, amelyek bizonyos határok között tartják ezt az értéket. A bonyolultság mellett a kód minőségének is mérőszámai a metrikák.

Az SSQSA projekt (Set of Software Quality Static Analyzers) nyelvfüggetlen módon végez metrika számításokat. A forráskódot ehhez egy XML-ben tárolt eCST-ben ábrázolja (enriched Concrete Syntax Tree), és ezen a reprezentáción számolja ki a metrikákat.

A dolgozatom célja, hogy a RefactorErl szintaxisfája és az SSQSA szintaxisfája között definiáljak egy leképezést. Megvizsgálom azokat a kérdéseket, hogy hogyan lehet egy funkcionális Szemantikus Program Gráf elemeit megjeleníteni egy impera-

tív programokhoz igazított szintaxisfa leíró eszközeinek segítségével.

## 2. fejezet

# RefactorErl

A *RefactorErl* egy nyílt forrású, statikus forráskód elemző eszköz, amely *Erlang* nyelven írt programok elemzését, klaszterezését, átalakítását és különböző metrikus információinak kiszámítását támogatja. A szoftvert az Eötvös Loránd Tudományegyetem Informatikai Karán a Programozási Nyelvek és Fordítóprogramok Tanszéken egyetemi projekt keretein belül, hallgatók bevonásával fejlesztik.

Az elemzések és transzformációk elvégzéséhez szükség van a szintaxisfára, és a programkódból kinyerhető szemantikus információkra. Az *Erlang* lehetőséget nyújt az absztrakt szintaxisfa [7] előállítására, és a legtöbb refaktoráló eszköz ezt használja fel. A *RefactorErl* ezzel ellentétben saját maga elemzi a kódokat, és az így kinyert információt *Szemantikus Program Gráf* [8, 9] formájában tárolja az adatbázisban.

### 2.1. Szemantikus Program Gráf

#### 2.1.1. A gráf bemutatása

A *Szemantikus Program Gráf* egy gyökeresúccsal ellátott, csúcs- és élcímkezett, irányított gráf, amelynek csúcsai az egyes program elemeket, élei pedig az ezek között fennálló kapcsolatokat reprezentálják. A gráf három rétegből tevődik össze:

- *Lexikális réteg*: Tartalmazza a forráskód lexikális információit. Minden a kódban szereplő *tokenhez* tartozik egy gráfcsúcs, amely többek között az adott elemet megelőző, és követő fehér szóközöket illetve megjegyzéseket tárolja. Ez a réteg egyszerre tartalmazza a *tokenek* eredeti állapotát, illetve az előfeldol-

gozás utáni állapotát (header beillesztés, makró definíció behelyettesítés).

- *Szintaktikai réteg*: A forráskódból készült *Absztrakt Szintaxisfa (AST)*, ebbe ágyazódik a lexikális reprezentáció. Négy csúcstípust sorolhatunk ide:
  - *file*: *Erlang* forrásfájl (modul, vagy header). A gyökércsúcsból a *file* címkéjű éleken keresztül érhetők el.
  - *form*: Az *Erlang* modulok építőkövei. A *file* csúcsokból a *form* címkéjű élekkel érhetők el. A csúcsból kivezető élek a *form* típusától függően *attr* vagy *funcl* címkéjűek lehetnek.
  - *clause*: Kifejezések sorozata (pl. függvény argumentumok, mintaillesztések, őrfeltételek). A *clause*-ok láthatósági mezőt definiálnak. Attól függően, hogy milyen programrészhez tartozó *clause*-ról van szó, különböző prefixű, *cl* címkéjű élen keresztül lehet elérni (pl. *funcl*, *aftercl*, *exprcl*). A *clause*-okból általában négy különböző címkéjű élen lehet tovább haladni a gráfban: *name*, *pattern*, *guard* és *body*. Nem minden *clause* rendelkezik mind a négy részfával, sőt a *receive* kifejezés *aftercl* élen elérhető *clause*-ából vezethet ki egy *tmout* címkéjű él is.
  - *expr*: Kifejezések. Három csoportba soroljuk őket:
    - \* Mintaillesztések
    - \* Őrfeltételek
    - \* Egyéb kifejezések
- *Szemantikai réteg*: Az *AST*-re épül, és kiegészíti a gráfot bizonyos szemantikus információkkal. Az alábbi csúcstípusokat vezeti be:
  - *module*: Egyetlen név attribútummal rendelkező, modul információkat leíró csúcs, aminek bemenő élei: *module* a gyökércsúcsból, *moddef* a modult tartalmazó *file* csúcsból, *modctx* a modulban definiált összes *clause*-ból. A modulra történő hivatkozás esetén a hivatkozó kifejezésbe vezet egy *modref* címkéjű él. A modul függvényeit reprezentáló szemantikus csúcsokba a *func* éleken keresztül lehet eljutni.
  - *function*: Két attribútumú (név, aritás), függvény információkat leíró csúcs. Bemenő élek: *func* a függvényt tartalmazó modul csúcsából, *fun-*

*def* a függvényt definiáló *form* csúcsból. Kimenő élei: *funexp* a modul csúcsba, ha exportált, *funimp* olyan modul csúcsba, amely importálja a függvényt és *funref* olyan kifejezés csúcsba, amely kifejezésben a függvény meghívásra kerül.

- *variable*: Egyetlen attribútumú (név), a változók szemantikus információit leíró csúcs. A csúcsból a *varintro* él az egyetlen kimenő él, ami az adott változót bevezető kifejezés csúcsába megy. Bemenő élek: *varvis* azokból a *clause*-okból, ahol a változó látható, *varbind* abból a kifejezésből, amelyben a változó értéket kap, és *varref* a változóra hivatkozó kifejezésekből.
- *egyéb*: rekord, adatfolyam, dynfun, stb.

### 2.1.2. Matematikai reprezentáció

Az alábbiakban az előző részben tárgyalt *Szemantikus Program Gráf* matematikai modelljét [9] láthatjuk:

$$SG \subseteq \mathcal{P}(N_S) \times \mathcal{P}(AN_S) \times \mathcal{P}(AV_S) \times \mathcal{P}(\Phi_S) \times \mathcal{P}(T_S) \times \mathcal{P}(\Psi_S) \times N_S$$

$$\begin{array}{ccccccc} nodes_S & anames_S & avals_S & \varphi_S & tags_S & \psi_S & roots_S \end{array}$$

ahol

- $nodes_S \subseteq N_S$  a gráf csúcsainak halmaza, amely tartalmazza az *AST* csúcsait, és a szemantikus csúcsokat is,
- $anames_S \subseteq AN_S$  az attribútum nevek halmaza,
- $avals_S \subseteq AV_S$  a lehetséges attribútum értékek halmaza,
- $\varphi_S : N_S \times AN_S \rightarrow AV_S$  a csúcsokat címkéző parciális függvény,
- $tags_S \subseteq T_S$  az élcímkék halmaza,
- $\psi_S : N_S \times T_S \times \mathbb{N}_0 \rightarrow N_S$  egy parciális függvény, ami a csúcsok közötti rendezett és címkézett éleket reprezentálja,
- $roots_S \in nodes_S$  a gráf szemantikus rétege nélkül tekintett fa gyökércsúcsa.

Ez a modell alkalmas a *Szemantikus Program Gráf* leírására, azonban ehhez bizonyos megkötéseket kell tennünk. Erre a célra definiálunk egy sémát, majd a séma segítségével szigorítjuk a gráfrepresentációt.

### 2.1.3. A gráf sémája

Definíció:

$SCH_S = (C_S, N_{C_S}, \varphi_{C_S}, \psi_{C_S})$ , ahol

- $C_S$  a megengedett csúcsosztályok halmaza,
- $N_{C_S} : N_S \rightarrow C_S$  a csúcsokat osztályozó függvény,
- $\varphi_{C_S} \subseteq C_S \times AN_S$  az egyes csúcsosztályokhoz tartozó attribútumokat tartalmazó reláció,
- $\psi_{C_S} : C_S \times T_S \rightarrow C_S$  az egyes osztályok között vezető megengedett éleket megadó parciális függvény.

Egy megadott  $sg \in SG$  gráf komponenseit a  $nodes_S(sg)$ ,  $anames_S(sg)$ , stb. projekciókkal adjuk meg. Azt mondjuk, hogy  $sg$  megfelel a sémának ( $SCH$ ), ha teljesíti az alábbi megkötéseket:

- $\mathcal{D}_{\varphi_S(sg)} = \{(n, a) \mid (N_{C_S}(n), a) \in \varphi_{C_S}\}$ : az adott csúcs attribútumai meg-  
egyeznek a séma által az adott osztályhoz rendelvekkel,
- $\forall n, t, i : N_{C_S}(\psi_S(sg)(n, t, i)) = \psi_{C_S}(N_{C_S}(n), t)$ : minden él megfelel a séma  
által definiált éleknek

### 2.1.4. Példa

Az alábbi példa kódot leíró *Szemantikus Program Gráfot* a 2.1 ábrán láthatjuk:

```
-module(example).  
f(A, B) ->  
    A + B.
```



## 3. fejezet

# Set of Software Quality Static Analyzers

Az *SSQSA* [11] egy statikus forráskód elemző szoftvercsomag, aminek legfőbb jellemzője a nyelvfüggetlenség. Kezdetben csupán egy programozási nyelvektől független metrika számoló eszköz, a *SMILE* (Software Metrics Independent of Input Language) létezett, és ezt egészítették ki a későbbiekben a *SNEIPL* (Software Network Extractor Independent on Programming Language) és a *SSCA* (Software Structure Change Analyzer) eszközökkel. A projekt motivációját az adta, hogy a fellelhető elemző eszközök általában egy nyelvet támogatnak, és az elemzéshez használt módszerük nem általános, így különböző nyelven megírt azonos szemantikájú programok elemzése különböző eredményeket adnak. Ez különösen azon programok esetében jelent gondot, amelyek különböző nyelven megírt komponensekből épülnek fel, mert az egyes programrészek különböző eszközökön történő elemzése nem ad egységes eredményt a teljes programra nézve. A *SSQSA* célja, hogy egy olyan szoftver elemző csomagot nyújtson a felhasználók számára, amely nyelvfüggetlen módon képes a szoftvereket elemezni, a többnyelvű programok esetén is egységes eredményt biztosítva.

### 3.1. eCST

A nyelvfüggetlenség legfontosabb kelléke egy olyan köztes reprezentáció, ami képes tárolni a forráskódból kinyerhető lényeges információkat, és ezzel egyidejűleg alkal-

mas az egyes programszerkezetek nyelvfüggetlen leírására is. Erre a célra egy XML-ben [6] tárolt kibővített szintaxisfa szolgál, amit röviden *eCST*-nek [12] (enriched Concrete Syntax Tree) nevezünk. Ez egy általános reprezentációja a vizsgált programstruktúráknak, amit a klasszikus konkrét szintaxisfából (*CST* [13]) kapunk úgy, hogy kibővítjük egy általános gráfcsúcs halmazzal, mely csúcsok a nyelvi szemantikai elemeket írják le. Ennek köszönhetően a forráskódból kinyert *eCST* tartalmazza az adott programnak az absztrakt struktúráját, illetve minden konkrét programbeli elemet, amik szükségesek lehetnek apróbb elemzések elvégzéséhez. Jelentős eltérések lehetnek az egyes nyelvek között szintaktikai szinten, mint például a *Pascalos* repeat-until és a *C*-s do-while ciklusok között, azonban a két szerkezet szemantikája megegyező, így absztrakt szinten ugyanúgy kerülnek bele az *eCST* gráfba, csupán a konkrét nyelvi elemekben fognak különbözni.

### 3.1.1. A gráf bemutatása

Az *eCST* egy irányított, csúcscímkezett gráf, melynek kétféle csúcstípusa van:

- *Konkrét szintaktikai elemek*: A forráskódban szereplő tényleges *tokenek*, a programok szintaxisát írják le.
- *Általános szemantikus elemek*: A kódban nem szereplő elemek, melyek az egyes konstrukciók szemantikáját írják le a gráfban, ezzel egy nyelvfüggetlen reprezentációját adva a vizsgált forráskódnak.

A szemantikus információ tárolásához egy általános csúcshalmazt használ az *SSQSA*, az alábbiakban néhány fontosabb csúcsot és jelentését láthatjuk:

- *COMPILATION\_UNIT*: Fordítási egység, ez a gyökércsúcs.
- *PACKAGE\_DECL*: Csomagok, névterek részfájának gyökércsúcsa, a fordítási egység közvetlen leszármazottja. A nevet leíró részfa gyökere (*NAME*) a közvetlen gyereke.
- *CONCRETE\_UNIT\_DECL*: Osztályok, modulok részfájának gyökércsúcsa, a névtér közvetlen gyereke. A részfa első szintjén szerepel a név részfája.
- *FUNCTION\_DECL*: Függvény definíciót leíró részfa gyökere. Leszármazottai:

- *MODIFIER\_LIST*: Módosító lista részfája (public, private, static, stb).
  - *TYPE*: Visszatérési érték típusát leíró részfa gyökere.
  - *NAME*: Az egyes programkonstrukciók nevét leíró részfa gyökércsúcsa.
  - *FORMAL\_PARAM\_LIST*: Paraméterlista részfája, melyben az első szinten minden részfa egy-egy paramétert ír le.
  - *BLOCK\_SCOPE*: A definíció törzsének részfája.
- *PARAMETER\_DECL*: A paraméterlista csúcsának közvetlen leszármazottja, és egy paraméter leírására alkalmas részfa gyökere. A paraméter leírását változó deklarációként reprezentálja, így tartalmaz egy-egy *MODIFIER\_LIST*, *TYPE* és *NAME* csúcsokból eredő részfát.
  - *VAR\_DECL*: A változó deklaráció részfájának gyökércsúcsa. A paraméter részfához hasonló módon reprezentálja a deklarációt, ami kiegészülhet egy kezdeti értékadással is, amit a *VALUE* ág ír le.
  - *BRANCH\_STATEMENT*: Az elágazásokat ábrázoló részfa gyökere, aminek közvetlen gyerekei egy vagy több *BRANCH* csúcs, amik az egyes ágakat vezetik fel a gráfban. Egy ágnak lehet neve, feltétele, és törzse.
  - *CONDITION*: Valamilyen kiértékelendő feltételt ábrázoló részfa gyökere. Szülői lehetnek *BRANCH*, vagy *LOOP\_STATEMENT* csúcsok.
  - *FUNCTION\_CALL*: Függvényhívás gráfcsúcs. A részfa tartalmazza a hívott függvény nevét, és argumentumlistáját (*ARGUMENT\_LIST*).
  - *ARGUMENT*: Az *ARGUMENT\_LIST* közvetlen leszármazottja, gyerekei lehetnek konstans értékek, változók, függvényhívások.
  - *LOOP\_STATEMENT*: Ciklus részfájának gyökércsúcsa.

### 3.1.2. Matematikai modell

Az alábbiakban az *eCST* matematikai modelljét láthatjuk:

$$ECST \subseteq \mathcal{P}(N_E) \times \mathcal{P}(AN_E) \times \mathcal{P}(AV_E) \times \mathcal{P}(\Phi_E) \times \mathcal{P}(\Psi_E) \times N_E$$

$$\text{nodes}_E \quad \text{anames}_E \quad \text{avals}_E \quad \varphi_E \quad \psi_E \quad \text{root}_E$$

ahol

- $nodes_E \subseteq N_E$  a gráf csúcsok halmaza, amely tartalmazza a  $CST$  csúcsait, és az általános szemantikus csúcsokat,
- $anames_E \subseteq AN_E$  az attribútumok halmaza:  $\{column, index, line, text, type\}$ ,
- $avals_E \subseteq AV_E$  a lehetséges attribútum értékek halmaza,
- $\varphi_E : N_E \times AN_E \rightarrow AV_E$  a csúcsokat címkéző teljes függvény,
- $\psi_E \subseteq N_E \times N_E$  a csúcsok között vezető irányított éleket megadó reláció,
- $root_E \in nodes_E$  a gráf gyökércsúcsa.

### 3.1.3. A gráf sémája

Definíció:

$SCH_E = (C_E, N_{C_E}, \varphi_{C_E}, \psi_{C_E})$ , ahol

- $C_E$  a megengedett csúcsosztályok halmaza,
- $N_{C_E} : N_E \rightarrow C_E$  a csúcsokat osztályozó függvény,
- $\varphi_{C_E} \subseteq C_E \times AN_E$  az egyes csúcsosztályokhoz tartozó attribútumok,
- $\psi_{C_E} \subseteq C_E \times C_E$  az egyes osztályok között megengedett élek halmaza.

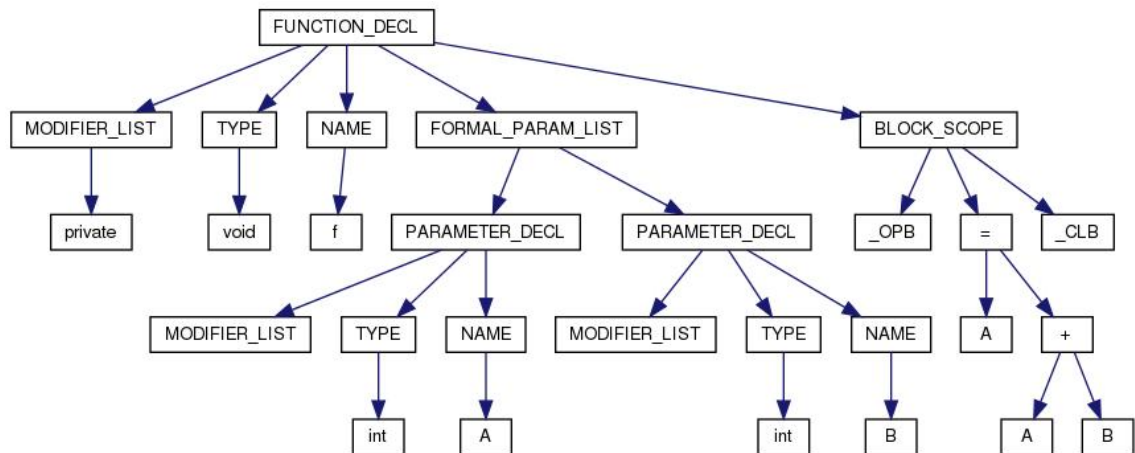
Egy megadott  $ecst \in ECST$  gráf megfelelő komponenseit rendre a  $nodes_E(ecst)$ ,  $anames_E(ecst)$ , stb. projekciókkal definiáljuk. Azt mondjuk, hogy az  $ecst$  megfelel a sémának  $(SCH_E)$ , ha teljesíti az alábbi megkötést:

- $\forall n, m \in nodes_E : (n, m) \in \psi_E(ecst) \Rightarrow (N_{C_E}(n), N_{C_E}(m)) \in \psi_{C_E} \wedge (m, n) \notin \psi_E(ecst)$ : minden él megfelel a séma által definiált éleknek,

### 3.1.4. Példa

Az alábbi *Java* példakódhoz generált *eCST*-t a 3.1 ábrán láthatjuk:

```
private void f(int A, int B)
{
    A = A + B;
}
```



3.1. ábra. eCST - példa

## 4. fejezet

# A leképezés

Ebben a fejezetben definiálom a két gráf közötti leképezés egyes lépéseit, és bevezetem a megfelelő fogalmakat illetve jelöléseiket.

### 4.1. Fogalmak, jelölések

#### 4.1.1. Szemantikus Program Gráfok részhalmaza

Jelölje  $c \in C_S$  esetén  $SG_c \subseteq SG$  a *Szemantikus Program Gráfok* azon részhalmazát, amelynek elemei a  $c$  típusú csúcsban gyökerező rész-szintaxisfák a hozzájuk tartozó lexikális elemekkel és szemantikai réteggel. Vagyis  $SG_c = \{sg \in SG \mid N_{C_S}(root_S(sg)) = c\}$ .

#### 4.1.2. $eCST$ -k részhalmaza

Jelölje  $ec \in C_E$  esetén  $ECST_{ec} \subseteq ECST$  az  $eCST$  gráfok azon részhalmazát, melynek elemei az  $ec$  típusú csúcsban gyökerező rész-fák. Azaz  $ECST_{ec} = \{ecst \in ECST \mid N_{C_E}(root_E(ecst)) = ec\}$ .

#### 4.1.3. Gyerekcsúcsok halmaza

Definiáljuk az alábbi függvényeket:

- $children_S : SG \rightarrow N_S$ ,  $sg \in SG$  esetén  $children_S(sg) = \{n_S \in nodes_S(sg) \mid \exists t \in tags_S(sg) : \exists i \in \mathbb{N}_0 : \psi_S(sg)(root_S(sg), t, i) = n_S\}$

- $children_E : ECST \rightarrow N_E$ ,  $e \in ECST$  esetén  $children_E(e) = \{n_E \in nodes_E(e) \mid (root_E(e), n_E) \in \psi_E(e)\}$

#### 4.1.4. Levélcsúcsok az $eCST$ -ben

Egy  $e \in ECST$  fa esetén, ha  $n \in nodes_E(e)$  csúcs levélcsúcs, akkor ezt az  $isleaf_E(n)$  kifejezéssel jelöljük.

#### 4.1.5. Segédfüggvények a mintaillesztésekhez

A mintaillesztések, szerepeljenek bármely pontján egy *Erlang* programnak, feltételeket fogalmaznak meg, amelyeket meg kell tudnunk fogalmazni imperatív eszközökkel is. Egy egyszerű minta alkalmas arra, hogy tömören fogalmazzon meg rengeteg feltételt egyidejűleg a vizsgált kifejezéssel kapcsolatban, ilyenek lehetnek a típusra, méretre, tartalomra vonatkozó feltételek, de még azt is ellenőrizhetjük, hogy egy összetett kifejezésen belül vannak-e további összetett kifejezések. Minden egyes ilyen megfogalmazható állítást ki kell fejeznünk, és ezek konjunkciója fogja lefedni a minta által meghatározott megkötéseket. A minták kiértékelését mindig kívülről befelé, az egyes szinteken balról jobbra haladva fogjuk megvalósítani. Az egyes szinteken lévő különböző elemekre az alább definiált függvények megfelelő kompozíciójával tudunk majd hivatkozni. A vizsgált kifejezést általában egy fiktív változónak adjuk értékül, így ezen keresztül tudunk rá hivatkozni az állításaink megfogalmazásakor. Jelölje  $Expr$  a vizsgálandó kifejezések halmazát,  $FVars$  pedig a felhasználható fiktív változók halmazát, és létezik egy  $\nu : Expr \rightarrow FVars$  bijekció, ami egy  $expr_j \in Expr$  kifejezéshez a hozzárendelt  $Var_j \in FVars$  változót adja eredményül ( $\nu(expr_j) = Var_j$ ,  $\nu^{-1}(Var_j) = expr_j$ ). Az egyszerűség kedvéért, ahol egy ilyen kifejezést használnánk, egyszerűen csak a megfelelő változóval hivatkozunk rá, ezért az alábbi függvényeket is bár  $Expr$  halmazon értelmezzük, a használatuk során az azonosításra bevezetett változót adjuk paraméterül.

#### A típus (szerkezet) ellenőrzése

Egy minta első ránézésre meghatározhatja a vizsgált kifejezés típusát, ami lehet *tuple*, *lista*, *rekord*, stb. Az ilyen tulajdonságok kifejezésére az alábbi logikai függvényeket vezetjük be, amelyek már beleillenek az imperatív eszközkészletbe:

- $is\_list : Expr \rightarrow Boolean$
- $is\_tuple : Expr \rightarrow Boolean$
- $is\_record : Expr \rightarrow Boolean$
- $is\_binary : Expr \rightarrow Boolean$

## A méret ellenőrzése

Ha tudjuk, hogy egy összetett szerkezetű kifejezést vizsgálunk, a minta alapján annak méretére is tudunk állításokat (egyenlőségek és egyenlőtlenségek) megfogalmazni. Ezekben az állításokban a  $size : Expr \rightarrow \mathbb{N}_0$  függvény segítségével fogunk a kifejezések méretére hivatkozni.

## Az egyes elemek ellenőrzése

Egy-egy összetett szerkezetű kifejezés egyes elemeit is ki kell tudnunk fejezni, hogy azokra is állításokat fogalmazhassunk meg. Erre a célra a  $sel : Expr \times \mathbb{N}_0 \rightarrow Expr$  szelektor függvényt vezetjük be. A definícióból látszik, hogy ez a függvény komponálható az összes eddig definiálttal, ennek köszönhetően tudjuk a rekurzív adatszerkezetek összes szintjére vonatkozó állításokat megfogalmazni (4.1.5.1 példa).

**4.1.5.1. Példa.** *Fogalmazzuk meg a  $Var_1$  kifejezésre a  $\{1, [a, 3]\}$  minta által definiált összes állítást:*

- $is\_tuple(Var_1),$
- $size(Var_1) == 2,$
- $sel(Var_1, 1) == 1,$
- $is\_list(sel(Var_1, 2)),$
- $size(sel(Var_1, 2)) == 2,$
- $sel(sel(Var_1, 2), 1) == a,$
- $sel(sel(Var_1, 2), 2) == 3.$

#### 4.1.6. Egyéb segédfüggvények

Bizonyos *Szemantikus Program Gráf*beli információk és műveletek kifejezése összetettebb, ezért bevezetünk néhány függvényt, melyek segítségével könnyebben hivatkozhatunk ezekre:

- $getvar : SG_{Expr} \times \mathbb{N}_0 \rightarrow N_S$ . Legyen  $s \in SG_{Expr}$  és  $n \in \mathbb{N}_0$ , akkor  $getvar(s, n)$  leképezés az  $s$  gráf által leírt kifejezés  $n$ . sorszámú változóját reprezentáló gráfcsúcsot adja eredményül.
- $clnum : N_S \rightarrow \mathbb{N}_0$ . Legyen  $node \in N_S$ , ekkor a  $clnum(node)$  eredménye, annak a függvény *clause*-nak a sorszáma, amelyikben  $node$  szerepel.
- $rename : N_S \times \mathbb{N}_0 \rightarrow AV_E|_{text}$ . Legyen  $node \in N_S$ , ekkor  $rename(node, 2) = \varphi_S(node, name) + \text{"\_2"}$ , azaz a kapott csúcs nevét kiegészíti egy aláhúzással és a kapott számmal.

#### 4.1.7. A leképezés általános alakja

Legyen  $f : C_S \rightarrow C_E$  a két gráf megengedett csúcsosztályai közötti injektív leképezés. Az  $f$  függvény segítségével megadhatjuk a gráfok közötti leképezés általános definícióját:  $convert_* : SG_* \rightarrow ECST_{f(*)}$ . A  $convert_*$  függvény a *Szemantikus Program Gráf* részgráfjai által leírt *Erlang* programszerkezetekhez hozzárendeli a velük szemantikusan ekvivalens<sup>1</sup> imperatív struktúrát leíró *eCST* részfákat. Az alábbiakban ezen leképezés definícióját láthatjuk a különböző  $SG_*$  halmazokra.

### 4.2. A leképezés konkrét esetei

#### 4.2.1. Modulok ( $SG_{module}$ )

A *Szemantikus Program Gráf* egyetlen gráfban tárolja az összes adatbázisban szereplő *Erlang* modult, egy közös gyökércsúcsból kiindulva. Az *eCST* viszont egyetlen fordítási egységet ír le, ezért a  $sg \in SG_{Root}$  minden moduljából készül egy-egy kü-

---

<sup>1</sup>Ez az ekvivalencia nem tekinthető teljesnek, ugyanis az *Erlang* programok szemantikus szabályai tartalmazznak olyan elemeket, amelyeknek nincs imperatív megfelelőjük

lönálló  $eCST$ . Egy modul megfelel az imperatív struktúra fordítási egységének, és a leképezés definiálásához vezessük be az alábbi jelöléseket:

- $s \in SG_{module}$ ,
- $e \in ECST_{COMPILATION\_UNIT}$ ,
- $ep \in ECST_{PACKAGE\_DECL}$ ,
- $ec \in ECST_{CONCRETE\_UNIT\_DECL}$  és
- $en \in ECST_{NAME}$ .

Ekkor a  $convert_{module}(s) = e$  leképezés szabályai a következők:

**4.2.1.1. Szabály.** Minden fordítási egység gráfjának első három szintje egyforma:

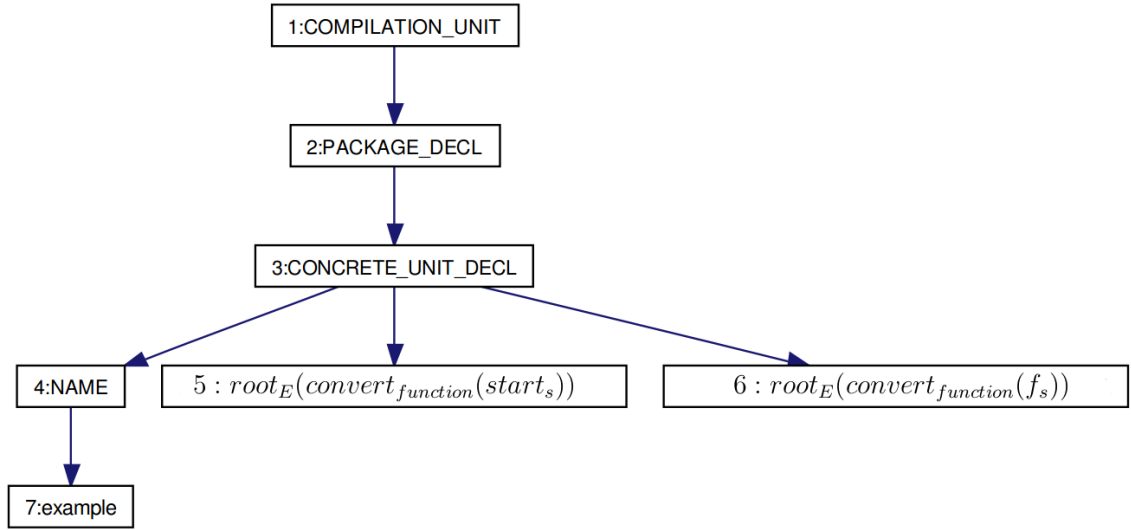
- $children_E(e) = \{root_E(ep)\}$ ,
- $children_E(ep) = \{root_E(ec)\}$ ,
- $children_E(ec) = \{root_E(en)\} \cup \{root_E(convert_{function}(s')) \mid root_S(s') \in children_S(s) \wedge N_{C_S}(root_S(s')) = function\}$ ,
- $children_E(en) = \{n' \mid \varphi_E(e)(n', text) = \varphi_S(s)(root_S(s), name) \wedge isleaf_E(n')\}$ .

**4.2.1.1. Példa.** `-module(example).`

```
start() ->
    f().
f() ->
    ok.
```

A fenti példa gráfja legyen  $s \in SG_{module}$ ,  $start()$  és  $f()$  függvényeit pedig jelöljük:  $start_s, f_s \in SG_{function}$ . Ekkor a  $convert_{module}(s) = e^{(1)} \in ECST_{COMPILATION\_UNIT}$  leképezés eredménye a 4.2.1.1 szabály szerint a következő (4.1. ábra):

- $children_E(e^{(1)}) = \{2\}$ ,
- $children_E(e^{(2)}) = \{3\}$ ,
- $children_E(e^{(3)}) = \{4, 5, 6\}$ ,
- $children_E(e^{(4)}) = \{7\}$ .



4.1. ábra.  $e \in ECST_{COMPILATION\_UNIT}$

#### 4.2.2. Függvénydefiníció ( $SG_{func}$ )

Az *Erlang* függvényei állhatnak egy, vagy több *clause*-ból, ahol az egyes *clause*-ok egy elágazás egy-egy törzsének felelnek meg. Ezen elágazás ágainak feltételeit a paraméterlista mintaillesztései és az őrfeltételek adják. Mivel az imperatív struktúrában nem ismert ez a fajta szerkezet, a függvények átalakítása összetett feladat. Első lépésben létrehozunk egy olyan függvénydefiníciót, amelynek egy törzse van. Ezt követően a törzsben rekurzívan egymásba ágyazott *if ... else* elágazásokkal reprezentáljuk a *clause*-ok szerinti esetszétválasztásokat. Erre a reprezentációra az egyes ágak sorrendjének megtartása miatt van szükség. Az első ágot az *if* tartalmazza, a többi ágra pedig az *else* ág törzsében létrehozuk a következő *if ... else* elágazást. Ezt a stratégiát mindaddig alkalmazzuk, amíg vannak további ágak az eredeti függvénydefinícióban. Az utolsó ágba, amennyiben sem a paraméterbeli mintaillesztések, sem az őrfeltételek nem indokolnak szétválasztást, nem hozunk létre újabb elágazást az *else* törzsében. Azon függvények esetében amelyek csak egy *clause*-ból állnak, és nincsenek belépési feltételek sem, nincs szükség elágazásra, és a definíció törzsébe az eredeti törzs képe kerül. Használjuk az alábbi jelöléseket a  $s \in SG_{func}$  gráf egyes részeinek jelölésére:

- $s_i \in SG_{clause} : root_S(s_i) \in children_S(s) (i \in [1..|children_S(s)|], clause \in C_S), s_i$  az  $i$ . *clause*-t leíró részgráf

- $n \in \text{children}_S(s_i); j \in \mathbb{N}_0; \text{sname}_{ij}, \text{spattern}_{ij}, \text{sguard}_{ij}, \text{sbody}_{ij} \in SG_{Expr}$   
(Expr tetszőleges kifejezés):

- $n = \psi_S(s)(\text{root}_S(s_i), \text{name}, j) \Rightarrow n = \text{root}_S(\text{sname}_{ij})$ , azaz  $n$  csúcs a függvény nevét leíró részgráf gyökércsúcsa. Minden *clause*-hoz tartozik egy nevet tartalmazó részgráf, és ezek tartalma megegyezik, ezért a leképezésnél csak az elsőt vesszük figyelembe, tehát  $i$  index elhagyható. Mivel függvénynek csak egy neve van,  $j$  értéke mindig 1 lesz, ezért ezt az indexet is elhagyhatjuk az egyszerűség kedvéért. A továbbiakban  $\text{sname}_{ij}$  helyett az  $\text{sname}$  jelölést használjuk erre a részgráfra.
- $n = \psi_S(s)(\text{root}_S(s_i), \text{pattern}, j) \Rightarrow n = \text{root}_S(\text{spattern}_{ij})$ , azaz  $n$  az  $i$ . *clause* formális paraméterlistájából a  $j$ . mintát reprezentáló részgráf gyökere.
- $n = \psi_S(s)(\text{root}_S(s_i), \text{guard}, j) \Rightarrow n = \text{root}_S(\text{sguard}_{ij})$ , azaz  $n$  az  $i$ . ág őrfeltételét ábrázoló részgráf gyökércsúcsa. Ebben az esetben  $j = 1$  (az őrfeltétel egy logikai állítás), ezért az egyszerűség kedvéért  $\text{sguard}_{ij}$  helyett az  $\text{sguard}_i$  jelölést használjuk.
- $n = \psi_S(s)(\text{root}_S(s_i), \text{body}, j) \Rightarrow n = \text{root}_S(\text{sbody}_{ij})$ , tehát  $n$  az  $i$ . *clause* törzsének  $j$ . kifejezését leíró részgráf gyökere.

A következő jelölések segítségével írjuk le  $e \in ECST_{FUNCTION\_DECL}$  fa megfelelő részeit:

- $\text{emod} \in ECST_{MODIFIER\_LIST}$
- $\text{etyp} \in ECST_{TYPE}$
- $\text{enam} \in ECST_{NAME}$
- $\text{efplist} \in ECST_{FORMAL\_PARAM\_LIST}$
- $\text{eblock} \in ECST_{BLOCK\_SCOPE}$ .

A  $\text{convert}_{func}(s) = e$  esetén:

**4.2.2.1. Szabály.**  $\text{children}_E(e) =$

$$\{\text{root}_E(\text{emod}), \text{root}_E(\text{etyp}), \text{root}_E(\text{enam}), \text{root}_E(\text{efplist}), \text{root}_E(\text{eblock})\}.$$

A definíció további részeit a fenti részfák szerinti csoportosításban fogalmazzuk meg:

#### Módosítók listája ( $emod \in ECST_{MODIFIER\_LIST}$ )

Imperatív nyelvek esetén ez a részfa írja le egy adott függvény láthatóságát. Egy *Erlang* függvény akkor publikus, ha valamelyik *export* form listájában szerepel. Ha nem szerepel egyik ilyen listában sem, akkor privátnak tekintendő. Ennek megfelelően:

**4.2.2.2. Szabály.**  $children_E(emod) = \{p \mid isleaf_E(p)\}$ , valamint:

- ha  $s$  exportált  $\Rightarrow \varphi_E(e)(p, text) = "public"$
- ha  $s$  nem exportált  $\Rightarrow \varphi_E(e)(p, text) = "private"$ .

#### A visszatérési érték típusa ( $etyp \in ECST_{TYPE}$ )

Egy *Erlang* függvény visszatérési értéke mindig az utolsó kifejezésének az értéke, ami bármilyen típusú lehet (Dinamikusan típusos nyelvről lévén szó). Több ágú függvények esetén az is előfordulhat, hogy az egyes ágak különböző típussal térnek vissza. Ezen okokból a visszatérési érték típusául minden esetben az *"any"*-t, azaz a legbővebb típust választjuk.

**4.2.2.3. Szabály.** Az *etyp* fában rögzítjük, hogy a visszatérési érték típusa *"any"* lesz. Formálisan:

$$children_E(etyp) = \{a \mid isleaf_E(a) \wedge \varphi_E(e)(a, text) = "any"\}.$$

#### A függvény neve ( $enam \in ECST_{NAME}$ )

A függvény nevét leíró fa szerkezetében megegyezik a moduloknál látottal.

**4.2.2.4. Szabály.** Az *enam* gráf írja le a függvény nevét, ami a *sname* gyökércsúcsának értéke. Formálisan:

$$children_E(enam) = \{nn \mid isleaf_E(nn) \wedge \varphi_E(e)(nn, text) = \varphi_S(s)(root_S(sname), value)\}.$$

#### Formális paraméterlista ( $efplist \in ECST_{FORMAL\_PARAM\_LIST}$ )

Az *Erlangban* egy függvény paraméterlistája kettős funkciót lát el, mert amellet, hogy felsorolja a paramétereket (amiből a definícióhoz tartozó összes *clause*-nak ugyanannyi van), mintaillesztés révén elágazási feltételeket is megfogalmazhat. Ezt

a két funkciót a leképezés során külön kell választanunk, ugyanis az *eCST* által támogatott imperatív nyelvekben a paraméterlista nem alkalmas feltételek megfogalmazására. A funkciók különválasztásához paramétereknek  $Var_j$  neveket adunk ( $j \in \mathbb{N}_0$ ), és az *eCST*-ben megszokott módon deklaráljuk őket, és a későbbiek során a mintaillesztések által definiált feltételek megfogalmazását ezek segítségével valósítjuk meg. Vezessük be az alábbi jelöléseket:

- $epar_j \in ECST_{PARAMETER\_DECL}$ ,
- $epm_j \in ECST_{MODIFIER\_LIST}$ ,
- $ept_j \in ECST_{TYPE}$  és
- $epn_j \in ECST_{NAME}$ .

A fentiek ismeretében az *efplist* felépítéséhez a következő szabályokat alkalmazzuk:

**4.2.2.5. Szabály.** *Az efplist fa második szintjén az egyes paramétereket leíró rész-fák helyezkednek el. Ezek darabszáma megegyezik a  $root_S(s_i)$  csúcsból kiinduló pattern címkéjű élek számával. Formálisan:*

$$children_E(efplist) = \{root_E(epar_j) \mid P = children_S(s_i)|_{nodes_S(s_i) \times pattern \times \mathbb{N}_0} \wedge j \in [1..|P|]\}.$$

**4.2.2.6. Szabály.** *Az  $epar_j$  paraméter deklarációnak három kötelező része van: módosítók listája, típus és név. Az imperatív nyelvekben ismert paraméter módosítók nem értelmezettek Erlang programok esetében, abba az ágba semmi sem fog szerepelni. Minden paraméter típusául az "any" legbővebb típust adjuk meg, és a  $j$ . paraméter neve a  $Var_j$  lesz. Formálisan:*

- $children_E(epar_j) = \{root_E(epm_j), root_E(ept_j), root_E(epn_j)\},$
- $children_E(epm_j) = \emptyset,$
- $children_E(ept_j) = \{a_j \mid isleaf_E(a_j) \wedge \varphi_E(e)(a_j, text) = "any"\}$  és
- $children_E(epn_j) = \{nn_j \mid isleaf_E(nn_j) \wedge \varphi_E(e)(nn_j, text) = "Var_j"\}.$

## A függvény törzse ( $eblock \in ECST_{BLOCK\_SCOPE}$ )

A függvény definíció törzsének felépítését két részre bontjuk, ugyanis eltérő szabályok vonatkoznak a következő esetekre:

- Az eredeti definíció csupán egy *clause*-ból áll, a paraméterlista nem fogalmaz meg belépési feltételeket, és őrfeltételek sincsenek. Ebben az esetben nincs szükség elágazásra.
- Belépési feltételekkel ellátott *clause*, vagy *clause*-ok alkotják a definíciót, és ekkor a definíciót elágazásokba szervezzük.

**Elágazásmentes definíció** Az elágazásmentes eset a legegyszerűbb, hiszen az eredeti definíció törzsében szereplő kifejezéseket egymás után képezzük le. Ennek megfelelően, a korábban bevezetett jelölésekkel *eblock* felépítését a következő szabály írja le.

**4.2.2.7. Szabály.** *A clause (ami általánosan az  $i$ . clause, de ebben a konkrét esetben ez az egyetlen, ezért  $i = 1$ ) törzsének utasításait sorrendben képezzük egy blokk szerkezetbe, amely blokkot a "{" és "}" szimbólumok határolnak. Formálisan:*

$$\begin{aligned} children_E(eblock) = & \{bsopen \in nodes_E(eblock) \mid \varphi_E(eblock)(bsopen, text) = "\{"\} \\ & \cup \{root_E(convert_c(sbody_{ij})) \in nodes_E(eblock) \mid \\ & Body = children_S(s_i)|_{nodes_S(s) \times body \times \mathbb{N}_0} \wedge i = 1 \wedge j \in [1..|Body|] \wedge \\ & N_{C_S}(root_S(sbody_{ij})) = c\} \\ & \cup \{bsclose \in nodes_E(eblock) \mid \varphi_E(eblock)(bsclose, text) = "\}"\}. \end{aligned}$$

**4.2.2.1. Példa.** *Tekintsük az alábbi függvényt (a gráfban  $start_s \in SG_{function}$ ), tegyük fel, hogy nem exportált, és keressük  $e^{(1)} \in ECST_{FUNCTION\_DECL}$  fát, amire  $convert_{function}(start_s) = e^{(1)}$  (4.2. ábra).*

```
start() ->
    started.
```

A 4.2.2.1 szabály alapján:

- $children_E(e^{(1)}) = \{2, 3, 4, 5, 6\}$ .

A 4.2.2.2 szabály alapján:

- $children_E(e^{(2)}) = \{7\}$ .

A 4.2.2.3 szabályt alkalmazva:

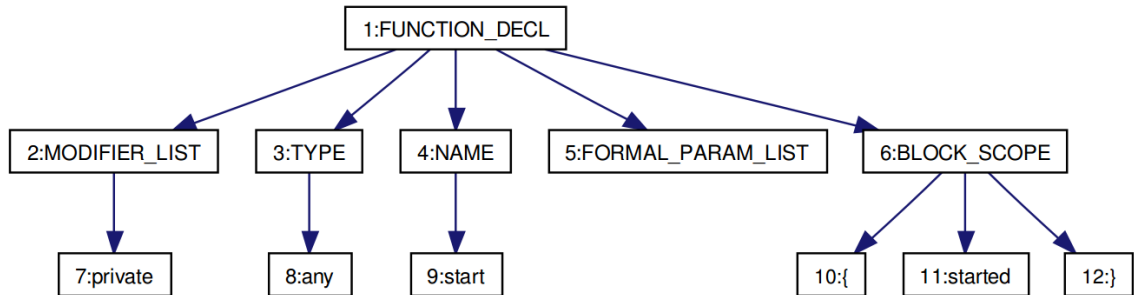
- $children_E(e^{(3)}) = \{8\}$ .

A 4.2.2.4 szabály miatt:

- $children_E(e^{(4)}) = \{9\}$ .

Mivel nincsenek formális paraméterek,  $isleaf_E(5)$ . Végül pedig a 4.2.2.7 szabály miatt:

- $children_E(e^{(6)}) = \{10, 11, 12\}$ .



4.2. ábra.  $e^{(1)} \in ECST_{FUNCTION\_DECL}$

**Elágazásos definíció** Az  $eCST$ -beli rekurzív elágazást az  $i$ . szint leírásával adjuk meg, ehhez vezessük be az alábbi jelöléseket:

- $ebrstate_i \in ECST_{BRANCH\_STATEMENT}$ ,
- $ebrnam_i \in ECST_{NAME}$ ,
- $ebri f_i, ebrlse_i \in ECST_{BRANCH}$ ,
- $eif_i \in ECST_{if}$ ,

- $eelse_i \in ECST_{else}$ ,
- $eifcond_i \in ECST_{CONDITION}$ ,
- $eifblock_i, eelseblock_i \in ECST_{BLOCK\_SCOPE}$ .

A leképezés megvalósítását az alábbi szabályok írják le.

**4.2.2.8. Szabály.** Az elágazás struktúrát az eblock blokkba ágyazzuk, amelyet a "{" illetve a "}" szimbólumok határolnak. Formálisan:

$$\begin{aligned} children_E(eblock) &= \{bsopen \in nodes_E(eblock) \mid \varphi_E(eblock)(bsopen, text) = "\{"\} \\ &\cup \{root_E(ebrstate_1)\} \\ &\cup \{bsclose \in nodes_E(eblock) \mid \varphi_E(eblock)(bsclose, text) = "\}"\}. \end{aligned}$$

**4.2.2.9. Szabály.** Az  $ebrstate_i$  fa második szintjén az elágazás nevét ("if"), valamint az if és az else ágakat leíró részfák gyökerei helyezkednek el (utóbbi csak akkor, ha van  $i + 1$ . clause is). Formálisan:

- $children_E(ebrstate_i) = \{root_E(ebrnam_i), root_E(ebri f_i), root_E(ebrelse_i)\}$ ,
- $children_E(ebrnam_i) = \{n \mid isleaf_E(n) \wedge \varphi_E(ebrnam_i)(n, text) = "if"\}$ ,
- $children_E(ebri f_i) = \{root_E(eif_i)\}$  és
- $children_E(ebrelse_i) = \{root_E(eelse_i)\}$ .

**4.2.2.10. Szabály.** Az  $eif_i$  fa gyökeréből pontosan két csúcsba vezet el. Egyik ilyen csúcs az elágazás feltételét tartalmazó részfa ( $eifcond_i$ ) gyökércsúcsa, a másik pedig az ág törzsét leíró részfa ( $eifblock_i$ ) gyökere. Az  $eelse_i$  esetében csak a törzs részfája ( $eelseblock_i$ ) található a második szinten.

Formálisan:

- $children_E(eif_i) = \{root_E(eifcond_i), root_E(eifblock_i)\}$  és
- $children_E(eelse_i) = \{root_E(eelseblock_i)\}$ .

Az első összetettebb részfeladat, az elágazás feltételének leképezése, ugyanis itt a paraméterlista mintákból és az őrfeltételekből egyetlen logikai állítást kell kapnunk. A probléma megoldásához külön képezzük a mintaillesztések feltételeit illetve az őrfeltételeket, majd ezek konjunkciója adja az eredményt.

A paraméterlistában többféle feltétel megfogalmazódhat, melyeknek teljesülniük kell a belépéshez. A paramétereket három fő csoportba sorolhatjuk, amely csoportok esetén különböző módon kell eljárunk a leképezés megvalósításához.

**Változó.** Ez a legegyszerűbb eset, ilyenkor csupán egy értékkötés történik a függvény hívásakor. Ebben az esetben az adott változót minden későbbi előfordulásakor a korábban létrehozott  $Var_j$  változó fogja helyettesíteni az  $eCST$ -ben. Ez a minta nem ad semmilyen belépési feltételt, ezért a leképezés során ezt nem tüntetjük fel az elágazás feltételében.

**Konstans literál (szám, atom)** esetén a függvény hívásakor a megfelelő argumentum értékének kell megegyeznie a megadott konstans értékkel. Az ebből a mintából adódó feltételt egy egyenlőségvizsgálattá képezzük, és kimenetként egy  $Var_j == konstans\ érték$  logikai kifejezést leíró részfát kapunk.

**Összetett kifejezések** esetén a 4.1.5 részben definiált függvények megfelelő alkalmazásával fogalmazzuk meg a megkötéseket, mindig a megfelelő  $Var_j$  változót használva. Ha az  $i$ . *clause*-ban egy paraméteren belül új változó kerül bevezetésre, akkor az adott változó gráfcúcsából ( $varnode$ ), és  $i$ -ből a  $rename(varnode, i)$  új változónevet generáljuk számára, és egy kezdeti értékadással egybekötött változó deklarációt képezzük, amelyet beillesztünk a feltételbe. A  $spattern_{ij}$  paraméterekből álló  $spattern_i$  lista feldolgozása során kapott konjunkció ( $spattern_i$ ) ugyanolyan feltétel lesz, mint egy őrfeltétel, ezért átírhatjuk a korábbi őrfeltételt egy olyan  $sguard_i^*$ -re, amely  $spattern_i$  és  $guard_i$  konjunkciója.

**4.2.2.11. Szabály.** Az  $eifcond_i$  fa felépítéséhez  $spattern_i$ -ből és  $sguard_i$ -ből képezzük  $sguard_i^* \in SG_{infix\_expr}$ -t. Ebben a konjunkció egy infix operátoros kifejezés, és leképezését a 4.2.13 szabály szerint valósítjuk meg, majd ennek eredménye kerül az  $eifcond_i$  második szintjére. Formálisan:

$$children_E(eifcond_i) = \{root_E(convert_{infix\_expr}(sguard_i^*))\}.$$

**4.2.2.12. Szabály.** Az  $i$ . szint törzsét az  $eifblock_i$  blokkba ágyazzuk, amelyet a "{" illetve a "}" szimbólumok határolnak. Magát a törzset a szinthez tartozó  $sbody_{ij}$  gráfok leképezéséből kapjuk. Formálisan:

$$\begin{aligned}
& children_E(eifblock_i) = \\
& \{ebsopen \in nodes_E(eifblock_i) \mid \varphi_E(eifblock_i)(ebsopen, text) = "\{\}"\} \\
& \cup \{root_E(convert_c(sbody_{ij})) \in nodes_E(eifblock_i) \mid \\
& Body = children_S(s_i)|_{nodes_S(s) \times body \times \mathbb{N}_0} \wedge j \in [1..|Body|] \wedge N_{C_S}(root_S(sbody_{ij})) = c\} \\
& \cup \{ebsclose \in nodes_E(eifblock_i) \mid \varphi_E(eifblock_i)(ebsclose, text) = "\}\}"\}.
\end{aligned}$$

Az *else* ág felépítése előtt meg kell vizsgálnunk a definíciót az *i*. szinten. Miután az *i*. *clause* törzse bekerült az *if* ágba, az alábbi két szabály közül kell kiválasztanunk a megfelelőt.

**4.2.2.13. Szabály.** Az *i*. az utolsó előtti *clause*, és az utolsónak nincsenek belépési feltételei. Ebben az esetben az utolsó ág törzsét az *eelseblock<sub>i</sub>* blokkba ágyazzuk, amit a "*{*" és a "*}*" szimbólumok határolnak. Formálisan:

$$\begin{aligned}
& children_E(eelseblock_i) = \\
& \{ebsopen \in nodes_E(eelseblock_i) \mid \varphi_E(eelseblock_i)(ebsopen, text) = "\{\}"\} \\
& \cup \{root_E(convert_c(sbody_{(i+1)j})) \in nodes_E(eelseblock_i) \mid \\
& Body = children_S(s_{(i+1)})|_{nodes_S(s) \times body \times \mathbb{N}_0} \wedge j \in [1..|Body|] \wedge \\
& N_{C_S}(root_S(sbody_{(i+1)j})) = c\} \\
& \cup \{ebsclose \in nodes_E(eelseblock_i) \mid \varphi_E(eelseblock_i)(ebsclose, text) = "\}\}"\}.
\end{aligned}$$

**4.2.2.14. Szabály.** Az *i*. *clause*-t egy olyan *clause* követi, aminek vannak belépési feltételei, akkor *ebrstate<sub>(i+1)</sub>* elágazás struktúráját az *eelseblock<sub>i</sub>* blokkba ágyazzuk, amit a "*{*" és a "*}*" szimbólumok határolnak. Formálisan:

$$\begin{aligned}
& children_E(eelseblock_i) = \\
& \{ebsopen \in nodes_E(eelseblock_i) \mid \varphi_E(eelseblock_i)(ebsopen, text) = "\{\}"\} \\
& \cup \{root_E(ebrstate_{(i+1)})\} \\
& \cup \{ebsclose \in nodes_E(eelseblock_i) \mid \varphi_E(eelseblock_i)(ebsclose, text) = "\}\}"\}.
\end{aligned}$$

**4.2.2.2. Példa.** A 4.2.2.1 példa függvényét írjuk át az alábbi módon:

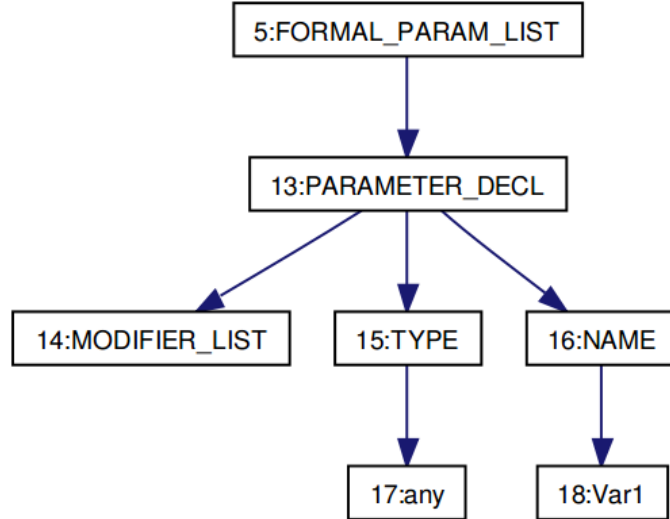
```

start({1,A}) ->
    A;
start(_) ->
    started.

```

Keressük  $start_s \in SG_{function}$  eCST-beli megfelelőjét! Az előző példához képest megváltozik a formális paraméterlistát és a törzset leíró részfa, a többi változatlan marad. A paraméterlista a 4.2.2.5 és 4.2.2.6 szabályok alapján (4.3 ábra):

- $children_E(e^{(5)}) = \{13\}$ ,
- $children_E(e^{(13)}) = \{14, 15, 16\}$ ,
- $children_E(e^{(14)}) = \emptyset$ ,
- $children_E(e^{(15)}) = \{17\}$  és
- $children_E(e^{(16)}) = \{18\}$ .



4.3. ábra.  $ECST_{PARAMETER\_DECL}$

### 4.2.3. Névtelen függvények ( $SG_{fun\_expr}$ )

A névtelen függvények, vagy függvénykifejezések leképezése a függvényeknél látott módon történik annyi módosítással, hogy az eCST-beli részfa gyökércsúcsa  $FUNCTION\_EXPR$  lesz, az első szintjén pedig nem szerepel módosító lista, sem név.

#### 4.2.4. Függvényhívás, applikáció ( $SG_{application}$ )

A függvényhívás, vagy applikáció a hivatkozott függvény nevét (névtelen függvény esetén a változó nevét, amiben tároljuk), és az argumentumait tartalmazza. A  $s \in SG_{application}$  gráf leírására vezessük be az alábbi jelöléseket:

- $s' \in SG_{nameexpr}$ ,
- $s'' \in SG_{arglist}$ ,
- $children_S(s) = \{root_S(s'), root_S(s'')\}$ .

Legyen  $e \in ECST_{FUNCTION\_CALL}$ , és a leírására használjuk a következő jelöléseket:

- $en \in ECST_{NAME}$ ;
- $eargs \in ECST_{ARGUMENT\_LIST}$  és

A fenti jelölésekkel a  $convert_{application}(s) = e$  leképezés szabálya a következő:

**4.2.4.1. Szabály.** Az  $e$  fában tárolódik a hivatkozott függvény neve, illetve argumentumlistája.

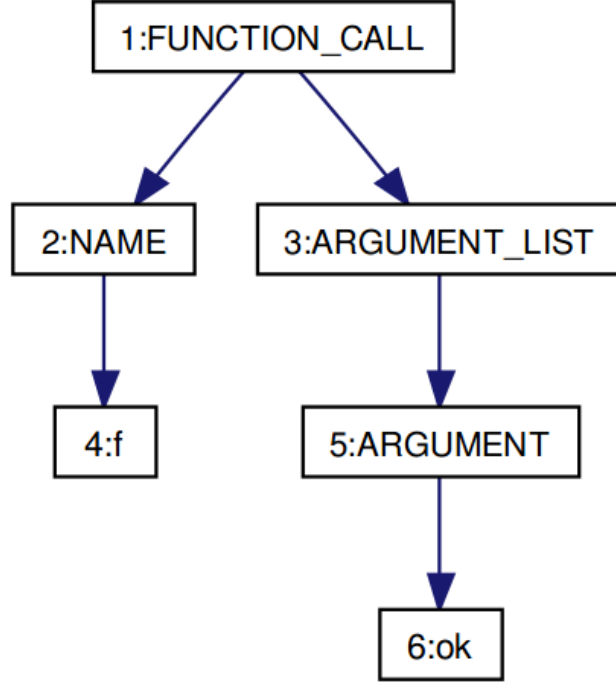
- $children_E(e) = \{root_E(en), root_E(eargs)\}$ ,
- $children_E(en) = \{n'\}$ , ahol  $n'$  a hivatkozott függvény nevét tartalmazó csúcs,
- $\forall a \in children_E(eargs): ea \in ECST_{ARGUMENT} \wedge children_E(ea) = \{root_E(convert_c(s''')) \in nodes_E(ea) | root_S(s''') \in children_S(s'') \wedge N_{C_S}(root_S(s''')) = c\}$ .

**4.2.4.1. Példa.** Legyen  $f_s \in SG_{application}$  az alábbi függvényhívást leíró gráf:

$f(ok)$ .

Keressük  $e^{(1)} \in ECST_{FUNCTION\_CALL}$  fát (4.4 ábra), amelyre  $convert_{application}(f_s) = e^{(1)}$ . A 4.2.4.1 szabály alkalmazásával:

- $children_E(e^{(1)}) = \{2, 3\}$ ,
- $children_E(e^{(2)}) = \{4\}$ ,
- $children_E(e^{(3)}) = \{5\}$ ,
- $children_E(e^{(5)}) = \{6\}$ .



4.4. ábra. Függvényhívás

#### 4.2.5. Értékadás kifejezés ( $SG_{match\_expr}$ )

Egy értékadás kifejezés egyszerre több változót vezethet be, amelyeket különböző összetettségű struktúrák elemeiként használhat mintaillesztésre. Ezeket az újonnan bevezetett változókat a leképezés során be kell vezetnünk az  $eCST$ -ben. Erre a célra kezdeti értékadásos változódeklarációkat ábrázoló részfákat kell létrehozni minden változó számára.

A változók kezdeti értékének kifejezésére a 4.1.5 segédfüggvényeket használjuk. Legyen  $s \in SG_{match\_expr}$  és  $\forall i \in [1..n] : e_i \in ECST_{VAR\_DECL}$ , ahol  $n$  az  $s$ -ben bevezetett új változók száma, és tudjuk az alábbiakat:

- $sln, srn \in SG_{Expr}$ ,
- $ln, rn \in children_S(s)$ ,
- $ln = root_S(sln)$  és
- $rn = root_S(srn)$ .

Vezessük be az alábbi jelöléseket az  $n$  darab deklaráció kifejezésére ( $i \in [1..n]$ ):

- $emod_i \in ECST_{MODIFIER\_LIST}$ ,

- $etyp_i \in ECST_{TYPE}$ ,
- $enam_i \in ECST_{NAME}$  és
- $eval_i \in ECST_{VALUE}$ .

Az  $convert_{match\_expr}(s) = \{e_i | \forall i \in [1..n]\}$  gráf egy deklarációs "erdő", amelyikben az egyes fák második szintjén sorrendben a következő részfák találhatók: módosító listák, típus, név és érték. Hasonlóan a függvénydefiníció paramétereire, a módosító lista itt is üres marad, az új változó típusa az "any" lesz, a nevét pedig a 4.1.6-ben ismertetett függvények segítségével képezzük. A kezdeti értéket (jobb-érték) az  $s$ -beli jobb-érték és a 4.1.5-beli függvények segítségével fogalmazzuk meg. A fa felépítését az alábbi szabályok alkalmazásával valósíthatjuk meg:

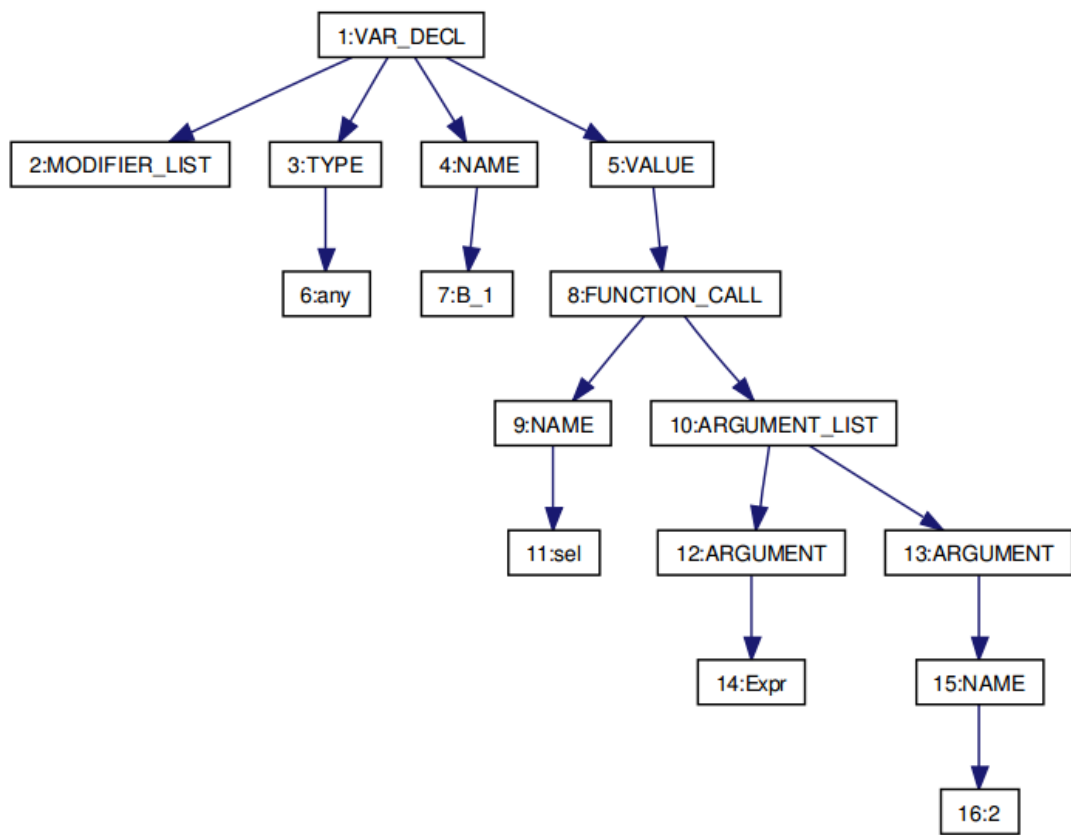
**4.2.5.1. Szabály.** *Az  $i$ . deklaráció kezdeti értékét a 4.1.5-beli  $sel$  függvény megfelelő számú egymásba ágyazása és  $srn$  segítségével kifejezzük, és nevezzük  $val_i^*$ -nak. Ezt a deklarációt leíró gráf szabálya formálisan:*

- $children_E(e_i) = \{root_E(emod_i), root_E(etyp_i), root_E(enam_i), root_E(eval_i)\},$
- $children_E(emod_i) = \emptyset,$
- $children_E(etyp_i) = \{a \mid isleaf_E(a) \wedge \varphi_E(e_i)(a, text) = "any"\},$
- $children_E(enam_i) = \{nn \mid isleaf_E(nn) \wedge var = getvar(ln, i) \wedge \varphi_E(e_i)(nn, text) = rename(var, cnum(var))\}$  és
- $children_E(eval_i) = \{root_E(convert_{application}(val_i^*))\}.$

**4.2.5.1. Példa.** *Jelölje  $varB \in SG_{variable}$  az alábbi kifejezés  $B$  változóját leíró gráfot, és írjuk fel a deklarációját megadó eCST fát, ha tudjuk, hogy  $cnum(varB) = 1$ :*

$\{A, B\} = Expr.$

*A 4.2.5.1 szabályt alkalmazva a 4.5 ábrán látható fát kapjuk eredményül.*



4.5. ábra.  $ECST_{VAR\_DECL}$

#### 4.2.6. Elágazás kifejezések ( $SG_{case\_expr}$ , $SG_{if\_expr}$ )

*Erlang* programok esetében a 4.2.2-ban megismert elágazáson kívül léteznek a függvény törzsön belül használható elágazások is. Ezek a *case ... of* illetve az *if* kifejezések. Egy *case\_expr* kifejezésnek van egy fejeleme, aminek az értéke alapján az egyes ágak mintái és őrfeltételei döntenek el, melyik ágra fut a vezérlés. Egy *if\_expr* kifejezés az előbbtől annyiban tér el, hogy nincs fejeleme, és az ágak feltételeit csak őrfeltételek alkotják, minták nélkül. A két szerkezet *eCST*-be történő képzése hasonló szabályokon alapul, ráadásul az eredmény szerkezete ugyanolyan, mint a 4.2.2 alfejezetben látott elágazásé.

**4.2.6.1. Szabály.** *Egy  $sc \in SG_{case\_expr}$  kifejtése során a fejelem ( $shead \in SG_{Expr}$ ,  $roots(shead) \in children_S(sc)$ ) kifejezést egy fiktív változónak adjuk értékül, majd ezzel a változóval fejezzük ki a feltételeket. Ezért ebben az esetben a kimeneten az  $e \in ECST_{BRANCH\_STATEMENT}$  részgráfot egy deklarációs  $v \in ECST_{VAR\_DECL}$  részgráf fogja megelőzni. A  $v$  gráf felépítése megegyezik a 4.2.5-ben tárgyaltával. Ebben a gráfban a változó neve egy olyan  $Var_{num}$ , ahol a  $num \in \mathbb{N}_0$  a  $roots(sc)$  csúcs azonosítója. A  $v$  gráfban a kezdeti érték helyére a  $convert_{Expr}(shead)$  kerül.*

A továbbiakban az  $sc$  és  $si \in SG_{if\_expr}$  gráfokat az *eCST*-ben reprezentáló rekurzív elágazásokat az *i.* szint leírásával adjuk meg, ehhez vezessük be az alábbi jelöléseket:

- $ebrstate_i \in ECST_{BRANCH\_STATEMENT}$ ,
- $ebrnam_i \in ECST_{NAME}$ ,
- $ebri f_i, ebre lse_i \in ECST_{BRANCH}$ ,
- $eif_i \in ECST_{if}$ ,
- $eelse_i \in ECST_{else}$ ,
- $eifcond_i \in ECST_{CONDITION}$  és
- $eifblock_i, eelseblock_i \in ECST_{BLOCK\_SCOPE}$ .

A fenti jelölésekkel az alábbi szabályok segítségével építhetjük fel az elágazást reprezentáló részfákat:

- 4.2.2.9,
- 4.2.2.10,
- 4.2.2.11,
- 4.2.2.12,
- 4.2.2.13,
- 4.2.2.14.

#### 4.2.7. Try kifejezések ( $SG_{try\_expr}$ )

Az *Erlang* programok elágazás kifejezései közé tartoznak a *try\_expr* kifejezések is, amelyekhez hasonlóval imperatív nyelvekben is találkozhatunk. Ez az elágazás mindig legalább kettő ágú, az első ágban szerepel a végrehajtandó alprogram, ami kivételt válthat ki. Amennyiben kivétel keletkezik, a vezérlés a megfelelő *catch* ágra kerül, majd legvégül, függetlenül attól, hogy keletkezett-e kivétel, vagy sem, a vezérlés az *after* ágra kerül, ha van ilyen ág. *Erlangban* a *try* kifejezéshez elágazások is tartozhatnak, melyekben a fejelem értékét próbáljuk illeszteni az egyes ágak mintáira, majd őrfeltételeire. Legyen:

- $s \in SG_{try\_expr}$ ,
- $e \in ECST_{BRANCH\_STATEMENT}$ ,
- $ebrname \in ECST_{NAME}$ ,
- $ebrtry, ebrcatch_i, ebrafter \in ECST_{BRANCH}$ ,
- $etry \in ECST_{try}$ ,
- $ecatch_i \in ECST_{catch}$ ,
- $eafter \in ECST_{after}$ ,
- $etryblock, ecatchblock_i, eafterblock \in ECST_{BLOCK\_SCOPE}$ .

A leképezés:  $convert_{try\_expr}(s) = e$ , ahol  $e$  felépítésének szabályai a következők:

**4.2.7.1. Szabály.** Az  $e$  fa második szintjén az elágazás nevét ("try"), valamint az try, az  $n$  darab catch és az after ágakat leíró részfaák gyökerei helyezkednek el. Formálisan:

- $children_E(e) =$   
 $\{root_E(ebrname), root_E(ebrtry), root_E(ebrcatch_1), \dots, root_E(ebrcatch_n),$   
 $root_E(ebrafter)\},$
- $children_E(ebrname) =$   
 $\{nam \mid isleaf_E(nam) \wedge \varphi_E(ebrname)(nam, text) = "try"\},$
- $children_E(ebrtry) = \{root_E(etry)\},$
- $\forall i \in [1..n] : children_E(ebrcatch_i) = \{root_E(ecatch_i)\}$  és
- $children_E(ebrafter) = \{root_E(eafter)\}.$

**4.2.7.2. Szabály.** A fejelemet  $shead \in SG_{Expr}$  ( $root_S(shead) \in children_S(s)$ ) képezzük a "try" ág törzsébe. Jelölje  $sbranches \in SG_{Expr}$  az  $s$  gráfban a try kifejezésbeli elágazást, amennyiben az létezik. Ekkor  $convert_{Expr}(sbranches)$  egy olyan  $ebranches \in ECST_{BRANCH\_STATEMENT}$  részfa lesz, mint 5.8-ben a  $case\_expr$  kifejezésből képzett fa. A fejelem fogja betölteni a  $case\_expr$  fejelemének szerepét. Ezért a fejelem, majd az elágazás képe együtt fogja alkotni a "try" törzsét.

- $children_E(etry) = \{root_E(etryblock)\},$
- $children_E(etryblock) =$   
 $\{bsopen \in nodes_E(etryblock) \mid \varphi_E(etryblock)(bsopen, text) = "\{"\}$   
 $\cup \{root_E(convert_{Expr}(shead)), root_E(ebranches)\}$   
 $\cup \{bsclose \in nodes_E(etryblock) \mid \varphi_E(etryblock)(bsclose, text) = "\}"\}.$

**4.2.7.3. Szabály.** Az  $ecatch_i$  részfaának második szintjén helyezkedik el az elkapott kivétel típusa ( $extype \in ECST$ ), illetve az ág törzse, amin belül az eredeti catch ág törzsének ( $catchbody_i \in SG_{Expr}$ ) képe helyezkedik el.

- $children_E(ecatch_i) =$   
 $\{root_E(extype) \mid is\_leaf(extype)\} \cup \{root_E(ecatchblock_i)\},$

- $children_E(ecatchblock_i) =$   
 $\{bsopen \in nodes_E(ecatchblock_i) \mid \varphi_E(ecatchblock_i)(bsopen, text) = "\{\}"\}$   
 $\cup \{root_E(convert_{Expr}(catchbody_i))\}$   
 $\cup \{bsclose \in nodes_E(ecatchblock_i) \mid \varphi_E(ecatchblock_i)(bsclose, text) = "\}\}"\}.$

**4.2.7.4. Szabály.** Az *after* részfa második szintjére az eredeti *after* clause törzsének ( $safterbody \in SG_{Expr}$ ) képe kerül.

- $children_E(after) = \{root_E(afterblock)\},$
- $children_E(afterblock) =$   
 $\{bsopen \in nodes_E(afterblock) \mid \varphi_E(afterblock)(bsopen, text) = "\{\}"\}$   
 $\cup \{root_E(convert_{Expr}(safterbody))\}$   
 $\cup \{bsclose \in nodes_E(afterblock) \mid \varphi_E(afterblock)(bsclose, text) = "\}\}"\}.$

#### 4.2.8. Listagenerátor kifejezés ( $SG_{list\_comp}$ )

*Erlang* programokban a listagenerátorok segítségével rendkívül tömören tudunk ciklusokat definiálni. Egy ilyen generátoros kifejezés alkalmas arra, hogy egy megadott lista elemeit megszűrje bizonyos feltételek szerint, majd a kiválasztott elemeket felhasználva kiértékelje a generátor fej részében megadott kifejezést. A listagenerátor egy fejelemből, generátorokból és a hozzájuk tartozó szűrőfeltételekből állnak. Az egyes listaelemeknek meg kell felelniük a generátor mintaillesztésének illetve a hozzá tartozó szűrőfeltételeknek. Egy kifejezésben szereplő generátorok az imperatív értelmezésben egymásba ágyazott ciklusokat reprezentálnak. A leképezés definíciójához a *C#* és *Java* nyelvek eszközkészletének *foreach* ciklus szerkezetét vesszük alapul. Egy  $n$  darab generátort, valamint minden  $i$ . generátorhoz  $n_i$  darab szűrőfeltételt tartalmazó listagenerátor kifejezés Szemantikus Program Gráfbeli reprezentációjának leírásához vezessük be az alábbi jelöléseket:

- $s \in SG_{list\_comp},$
- $shead \in SG_{hexpr},$  a fejelem részgráfja,
- $scompr \in SG_{compr},$  a generátorok és szűrők részgráfja,
- $slistgen_i \in SG_{list\_gen},$  az  $i$ . generátor,

- $sfilter_{ij} \in SG_{filter}$ , a  $slistgen_i$ -hez tartozó  $j$ . feltétel részgráfja, valamint
- $slistexpr_i, spattern_i \in SG_{Expr}$ , az  $i$ . generátor által feldolgozott listakifejezés és minta.

A fenti elemek viszonyát az alábbi összefüggések írják le:

- $children_S(s) = \{root_S(shead), root_S(scompr)\},$
- $children_S(scompr) = \{root_S(slistgen_i) | \forall i \in [1..n]\} \cup \{root_S(sfilter_{ij}) | \forall i \in [1..n] \wedge j \in [0..n_i]\},$  és
- $children_S(slistgen_i) = \{root_S(slistexpr_i), root_S(spattern_i)\}.$

Az  $eCST$  reprezentáció leírásához vezessük be az alábbi jelöléseket:

- $\forall i \in [1..n] : e_i \in ECST_{LOOP\_STATEMENT}, ename_i \in ECST_{NAME},$
- $elooopcond_i \in ECST_{CONDITION},$
- $\forall k \in [0..n] : evardecl_k \in ECST_{VAR\_DECL},$
- $ebscope_i \in ECST_{BLOCK\_SCOPE}$  és
- $ebrstate_i \in ECST_{BRANCH\_STATEMENT}.$

A  $convert_{list\_comp}(s) = e$  leképezés szabályai a következők:

**4.2.8.1. Szabály.** Az egész kifejezésre egy listaként tekintve, az azonosítására bevezetünk egy fiktív változót üres listát adva kezdeti értéként. Majd minden lépés eredményével bővítjük ezt a változót a "push" művelet felhasználásával. Ily módon az  $e$  gráf valójában két részfából fog állni, a fenti változó deklarációjából ( $evardecl_0$ ) és az  $e_1$  ciklusból Formálisan:  $e = \{evardecl_0, e_1\}.$

**4.2.8.2. Szabály.** Az  $evardecl_0$  deklaráció kezdeti értéke tehát az üres lista lesz, és a deklarációt leíró gráf szabálya formálisan:

- $children_E(evardecl_0) = \{root_E(emod_0), root_E(etyp_0), root_E(enam_0), root_E(eval_0)\},$
- $children_E(emod_0) = \emptyset,$

- $children_E(etyp_0) = \{a \mid isleaf_E(a) \wedge \varphi_E(e_0)(a, text) = "any"\}$ ,
- $children_E(enam_0) = \{nn \mid isleaf_E(nn)\}$  és
- $children_E(eval_0) = \{vv \mid isleaf_E(vv) \wedge varphi_E(evardecl_0)(vv, text) = "[]"\}$ .

**4.2.8.3. Szabály.** Az egymásba ágyazott ciklusok közül a legkülső lesz az első, és a legbelső az  $n$ ., amelynek a magjába kerül a fejelem képe.  $\forall i \in [1..n]$  ciklus, azaz  $e_i$  felépítése a következő:

- $children_E(e_i) = \{root_E(ename_i), root_E(eloopcond_i), root_E(ebscope_i)\}$ ,
- $children_E(ename_i) = \{nn \mid isleaf_E(nn) \wedge varphi_E(e_i)(nn, text) = "foreach"\}$ .

**4.2.8.4. Szabály.** Az  $i$ . ciklus feltételében deklarálunk egy változót ( $evardecl_i$ ), amivel szimuláljuk az  $evardecl_{i-1}$  által jelölt lista elemein történő iterációt. Minden  $evardecl_i$  változóval az eredeti gráf  $slistexpr_i$  kifejezését azonosítjuk. Ennek megfelelően a  $eloopcond_i$  részfa felépítése a következő:

- $children_E(eloopcond_i) = \{root_E(evardecl_i)\} \cup \{einnode \in nodes_E(eloopcond_i) \mid isleaf_E(einnode) \wedge \varphi_E(eloopcond_i)(einnode, text) = "in"\} \cup \{root_E(evardecl_{i-1})\}$ .

**4.2.8.5. Szabály.** Az  $i$ . ciklus törzsébe egy elágazást ágyazunk, amelynek csupán egy if ága van, és feltételét a 4.1.5-ben ismertetett függvények segítségével a  $spattern_i$  mintából és a  $\forall j \in n_i : sfilter_{ij}$  szűrőfeltételekből képezzük. Az elágazás felépítési szabálya megegyezik a 5.8 fejezetben ismertetettel. Az egyetlen ág törzsébe kerül az  $i + 1$ . ciklust leíró  $e_{i+1}$  fa, amennyiben  $i \in [1..n - 1]$ . Ha  $i = n$ , akkor a törzsbe  $convert_{hexpr}(shead)$  kerül, aminek értékét  $vardecl_0$  változójához adjuk a "push" művelettel, ezt pedig  $ECST_{FUNCTION\_CALL}$  faként ábrázoljuk.

## 4.2.9. Tuple kifejezés ( $SG_{tuple}$ )

Erlangban a rendezett  $n$ -esek típusát *tuple*-öknek nevezzük, melyeknek eleme bármilyen kifejezés lehet. Az *eCST* eszközkészlete nem tartalmaz olyan elemeket, amikkel

egy *tuple* leírható, ezért a leképezés megvalósításához egy *TUPLE* típusú csúccsal bővítjük  $N_E$ -t. Ekkor  $s \in SG_{tuple}$  és  $e \in ECST_{TUPLE}$  esetén  $convert_{tuple}(s) = e$  szabálya:

**4.2.9.1. Szabály.**  $children_E(e) = \{root_E(convert_{cl}(s')) \mid root_S(s') \in children_S(s) \wedge N_{C_S}(root_S(s')) = cl\}$ .

#### 4.2.10. Lista kifejezés ( $SG_{cons}$ )

A *tuple*-höz hasonlóan a  $[E, \dots, E|E]$  ( $E$  itt tetszőleges kifejezést jelöl) alakú lista kifejezések ábrázolására sincs eszköze az *eCST*-nek. Ahhoz, hogy  $N_E$  halmazt ki tudjuk egészíteni, előbb nézzük meg hogyan épül fel  $s \in SG_{cons}$  gráf. Ha tudjuk, hogy  $E$  egy tetszőleges kifejezést jelöl, akkor  $children_S(s) = \{root_S(s') \mid s' \in SG_{list}\} \cup \{root_S(s'') \mid s'' \in SG_E\}$ .  $SG_{cons}$  és  $SG_{list}$  gráfoknak kell megadnunk a megfelelőjét, tehát  $N_E$ -t az alábbi két csúcstípussal bővítjük:

- $f(cons) = LIST\_EXPR$
- $f(list) = LIST$

Vezessük be az alábbi jelöléseket:

- $s \in SG_{cons}$ ,
- $e \in ECST_{LIST\_EXPR}$ ,
- $l \in nodes_E(e)$ ,
- $N_{C_E}(l) = LIST$ .

Ekkor  $convert_{cons}(s) = e$  szabálya:

**4.2.10.1. Szabály.** A listakifejezés képe formálisan:

- $children_E(e) = \{l\} \cup \{root_E(convert_a(tail)) \in nodes_E(e) \mid tail \in children_S(s) \wedge N_{C_S}(tail) = a\}$ ,
- $children_E(l) = \{root_E(convert_a(expr)) \in nodes_E(l) \mid expr \in children_S(s') \wedge s' \in SG_{list} \wedge root_S(s') \in children_S(s) \wedge N_{C_S}(expr) = a\}$ .

#### 4.2.11. Zárójeles kifejezések ( $SG_{parenthesis}$ )

Az  $(E)$  alakú kifejezések szerkezete mindkét gráfban azonos, tehát  $s \in SG_{parenthesis}$ ;  $e \in ECST_{PARENTHESIZED\_EXPR}$  és  $convert_{parenthesis}(s) = e$  esetén:

- $children_E(e) = \{open \in nodes_E(e) \mid$   
 $\varphi_E(e)(open, text) = "("\}$   
 $\cup \{root_E(convert_c(s')) \in nodes_E(e) \mid s' \in children_S(s) \wedge \exists n \in \mathbb{N}_0 :$   
 $\psi_S(s)(root_S(s), esub, n) = root_S(s') \wedge N_{C_S}(root_S(s)) = c\}$   
 $\cup \{close \in nodes_E(e) \mid$   
 $\varphi_E(e)(close, text) = ")"\}$ , vagyis a zárójeles kifejezést a nyitó és csukó záró-  
jelek határolják, és a kettő között pedig a kifejezés belsejében lévő kifejezések  
képei helyezkednek el a megfelelő sorrendben.

#### 4.2.12. Blokk kifejezések ( $SG_{block\_expr}$ )

A *begin* és *end* kulcsszavak által közrezárt blokkok megfelelnek az *eCST*-ben a *BLOCK\_SCOPE* csúcsban gyökerező részfa által leírt programrésznek. Legyen  $s \in SG_{block\_expr}$ ;  $e \in ECST_{BLOCK\_SCOPE}$  és  $convert_{block\_expr}(s) = e$ , ekkor:

- $children_E(e) = \{bsopen \in nodes_E(e) \mid$   
 $\varphi_E(e)(bsopen, text) = "{"\}$   
 $\cup \{root_E(convert_c(s')) \in nodes_E(e) \mid root_S(s') \in children_S(s) \wedge e' =$   
 $convert_c(s') \wedge \varphi_E(e)(root_E(e'), text) \notin \{'', '\}'\} \wedge N_{C_S}(root_S(s')) = c\}$   
 $\cup \{bsclose \in nodes_E(e) \mid$   
 $\varphi_E(e)(bsclose, text) = "}\}$ , azaz egy "{" és "}" szimbólumok által határolt  
blokkon belül leképezzük  $s$  kifejezéseit a megfelelő sorrendben.

#### 4.2.13. Konstansok és operátorok ( $SG_{infix\_expr}, SG_{prefix\_expr}$ )

Az  $E \circ E$ ,  $E!E$ ,  $\circ E$  alakú kifejezések és konstansok esetében a legegyszerűbb a leképezés definíciója, ugyanis mindkét struktúrában ugyanúgy néznek ki az ezeket leíró részgráfok.

### 4.3. A tárolt információ

Az *eCST* egy élcímkezetlen fa, ami csupán a kódban szereplő lexikális elemek sor- és oszlopindexét tartalmazza, minden más a számításokhoz szükséges információt az általános csúcsok gráfbeli elhelyezkedéséből nyer ki a *SSQSA*. Ezzel szemben a *Szemantikus Program Gráf* élei címkézettek, az egyes elemek jóval több információt tárolnak a forráskódról, ami rekonstruálhatóvá teszi azt. A két gráf közötti leképezés során rengeteg információt elvetünk, ugyanis az szükségtelen az *eCST* reprezentáció megvalósításához. A rengeteg információ eldobása azt eredményezi, hogy a leképezés megfordításával, nem az eredeti Erlang programot kapjuk vissza, hanem csupán egy azzal szemantikailag megegyezőt.

## 5. fejezet

# Megvalósítás

Folyamatban van egy a dolgozatban tárgyalt leképezést megvalósító program fejlesztése a *RefactorErl* eszköz részeként. A fejlesztés során folyamatosan bővitem a program funkcióit, így egyre több kifejezés átalakítását képes elvégezni. Jelen állapotában az átalakító az alábbi funkciókat biztosítja:

- *Modulok leképezése:* Minden modulhoz készül egy *eCST* gráf, amiben a "COMPIRATION\_UNIT"  $\rightarrow$  "PACKAGE\_DECL"  $\rightarrow$  "CONCRETE\_UNIT\_DECL" hármassal kezdődik a fa. Az említett csúcsokból az utolsóhoz kapcsolódik a modul nevét leíró részfa.
- *Függvények leképezése:* Minden függvénydefiníciót egy törzsből álló definícióvá alakít, és ahol szükséges létrehozza a fiktív elágazást a törzsben. A függvény paraméterbeli mintaillesztéseiből, és az őrfeltételekből felépíti az egyes ágak feltételeit, az ágak törzsébe pedig az egyes *clause*-ok törzse kerül.
- *Elágazások leképezése:* A függvények esetén a fiktív elágazást létrehozó részei a programnak alkalmasak a tényleges elágazások imperatív képének megalkotására is.
- *Értékadások leképezése:* Az értékadásokat változó deklarációként képezi le a program.
- *Mintaillesztések leképezése:* A mintaillesztések által definiált feltételekkel ekvivalens imperatív kifejezéseket generál a program, ezen kívül az összetett mintákban szereplő változók kezdeti értékadással egybekötött deklarációját is

megfogalmazza, ahol a kezdeti érték a minta azonosítására létrehozott fiktív változóra meghívott függvény formájában jelenik meg. Ez a függvényhívás a változó mintabeli szerepét írja le (pl. lista n. eleme).

- *Névtelen függvény kifejezések leképezése:* Szerkezetükben a függvény definícióhoz hasonlítanak, azzal a különbséggel, hogy itt nincs név. Az *eCST*-ben a névtelen függvények részfáit egy "FUNCTION\_EXPR" csúcs vezeti be.
- *Makrók leképezése:* A program a makrók kifejtése utáni programkód által leírt szerkezeteket képezi le.
- *Változók leképezése:* Minden változó helyett bevezetésre kerül egy új változó, ami által egyértelmű lesz a leképezés után is az egyes változók láthatósága. A *Szemantikus Program Gráf* változókat leíró szemantikus csúcsainak segítségével határozza meg a program, hogy mikor melyik bevezetett változónév kerüljön az *eCST*-be.
- *Egyszerű kifejezések leképezése:* A legegyszerűbb kifejezések esetében (pl. infix operátorok, konstansok) nem kell semmilyen szerkezeti átalakítást végrehajtania a programnak, hiszen ezek az imperatív struktúrában is ugyanúgy jelennek meg.

## 5.1. Az átalakító program használata

A program a *RefactorErl* szerves részét képezi, így a használatához előbb ezt kell elindítanunk. Ha elindítottuk a *RefactorErlt*, és vannak betöltött modulok az adatbázisában, akkor ezekből készíthetünk *eCST* gráfot az *eCST-konverter* segítségével. Három opcióból választhatunk:

- Az összes modult konvertáljuk
- Egy kiválasztott modult konvertálunk
- Egy kiválasztott modul valamelyik függvényét konvertáljuk

A választásunktól függően a `ri:eCST/0`, a `ri:eCST/1`, vagy a `ri:eCST/3` függvényt kell meghívunk. A függvény nevét követő szám az aritást, vagyis a paraméterek

számát jelenti, így a teljes adatbázis leképezéséhez nem kell megadnunk semmit paraméterben, egy modul esetén annak nevét, míg egy kiválasztott függvény esetén az azt tartalmazó modul nevét, a függvény nevét, és annak aritását kell megadnunk.

A kimenet minden esetben egy lista, aminek minden eleme egy átalakított modulhoz tartozik. Egy listaelem egy három tagú *Tuple*, ahol az első tag minden esetben a "Writing file" szöveg, a második tag a kimeneti fájl elérési útvonala, míg a harmadik tag a fájlba írás sikerességétől függően egy *ok*, vagy egy *error* atom. A kimeneti fájlok neve az átalakított modulok fájlnevéből, az utolsó módosítás időpontjából és csak függvény konverziónál az átalakított függvény nevéből épül fel.

## 5.2. Tesztelési esettanulmány

**Egyszerű függvény 1** Az alábbi példakódhoz generált gráfot a 5.1 ábrán láthatjuk:

```
-module(f2).  
g(A)-> A.
```

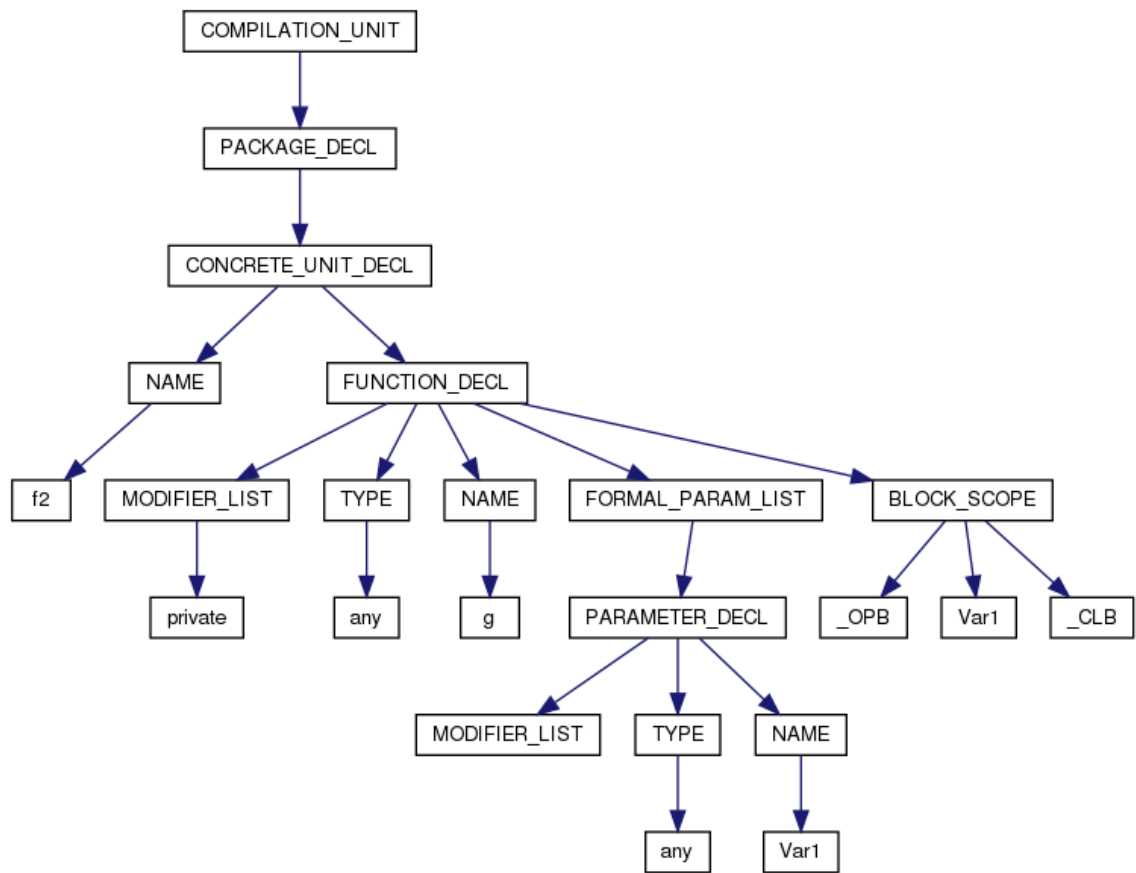
A gráf tartalmazza a kötelező "COMPILATION\_UNIT" és "PACKAGE\_DECL" csúcsokat. A következő szinten a modult reprezentáló "CONCRETE\_UNIT\_DECL" csúcs is jelen van, aminek egyik gyereke a modul nevét írja le, a másik pedig a "FUNC\_DECL" gyökerű részfa. A függvény konstrukció is tartalmazza az összes kötelező elemet.

A további tesztesetekben a kódok bizonyos részletét, és a hozzá tartozó részfákat mutatom be.

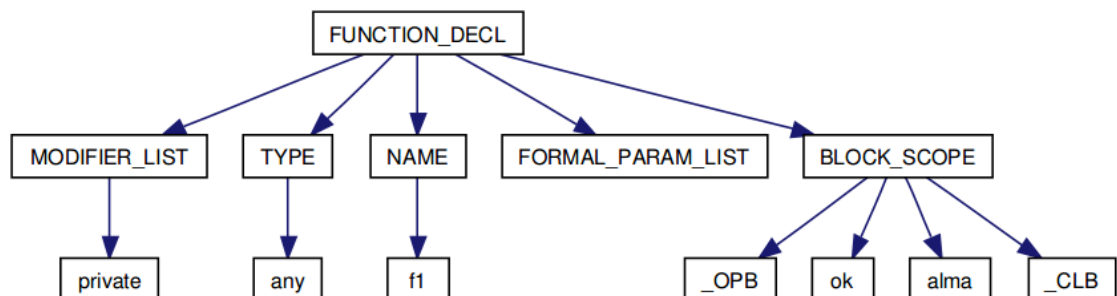
### Egyszerű függvény 2

```
-module(f3).  
f1()->  
    ok,  
    alma.
```

A 5.2 ábrán jól látható, hogy a kötelező függvény elemek megvannak, és a törzsben megjelenik a két különböző kifejezés is.



5.1. ábra. Egyszerű függvény 1 - teszt



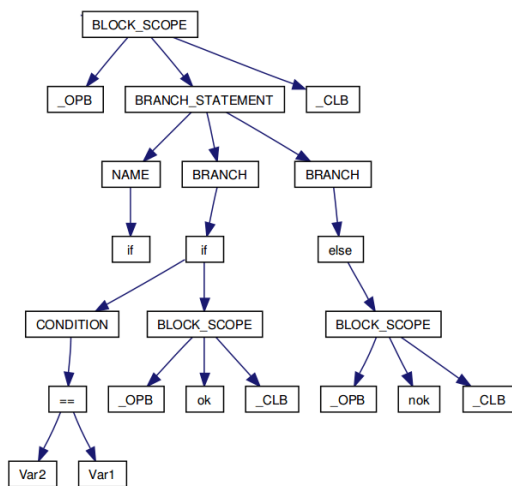
5.2. ábra. Egyszerű függvény 2 - teszt

**Több clause-os függvény 1** Az alábbi két modul  $f/2$  függvénye ugyanazt a feladatot oldja meg, eldönti két értékről, hogy megegyeznek-e:

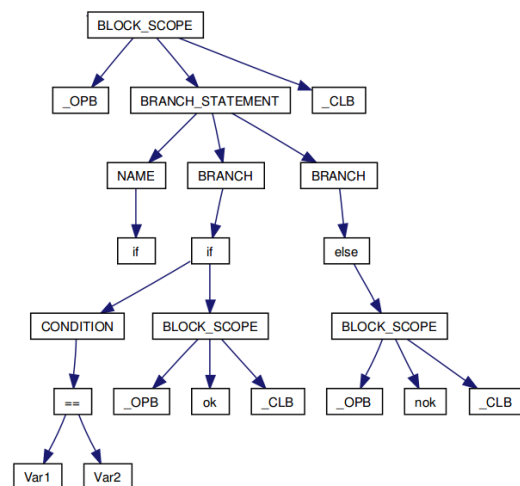
```
-module(f11).  
f(A, A) -> ok;  
f(_, _) -> nok.
```

```
-module(f12).  
f(A, B) when A == B -> ok;  
f(_, _) -> nok.
```

A két függvény *eCST*-jének hasonlítsuk össze a törzset reprezentáló részét a 5.3 és 5.4 ábrákon!



5.3. ábra. f11:f/2 - teszt

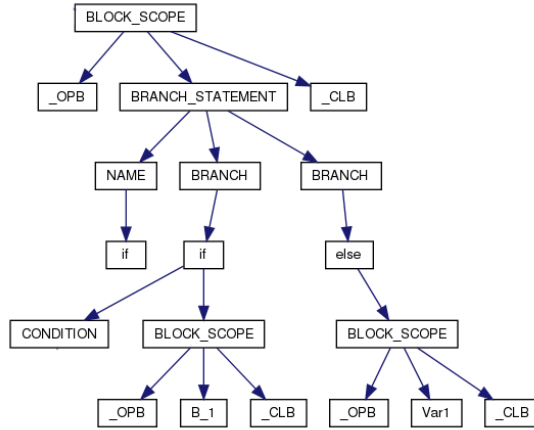


5.4. ábra. f12:f/2 - teszt

**Több clause-os függvény 2** Az alábbi függvény néhány komponensén mutatom be az eredményt, ugyanis a teljes gráf már túl nagy ahhoz, hogy megjeleníthessem:

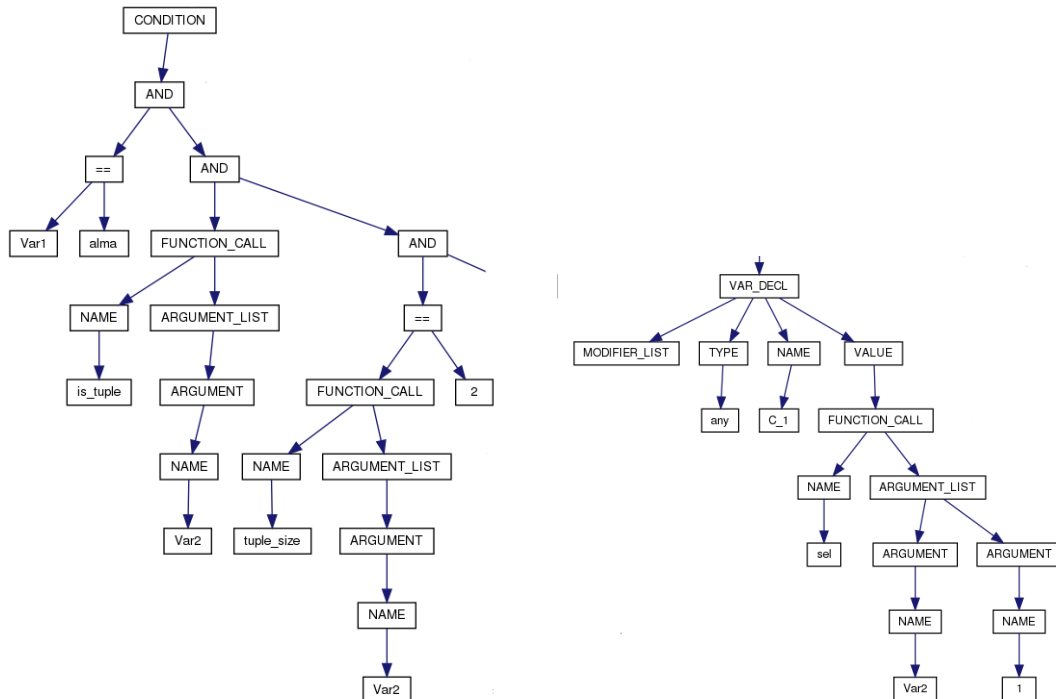
```
-module(f9).  
h(alma, {C, {B,D}}) when C == korte; B == alma -> B;  
h(A,B)-> A.
```

A 5.5 ábrán láthatjuk a függvényhez létrehozott fiktív elágazás szerkezetét. Az elágazás feltételének reprezentációja túl nagy méretű, ezért egy-egy kiragadott pél-



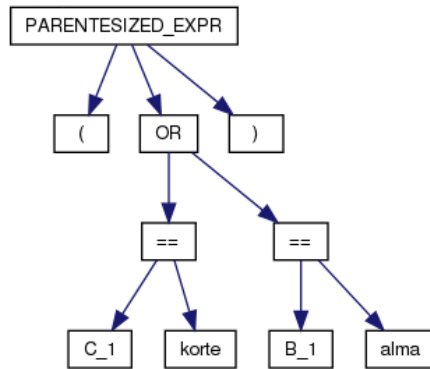
5.5. ábra. Fiktív elágazás szerkezete - teszt

dában szemléltetem az eredményt. A 5.6 ábra bal oldalán a minta kód két paraméterére a mintaillesztés által meghatározott feltételek egy részét láthatjuk. A második változóra további megkötések vonatkoznak, hiszen egyik eleme szintén összetett kifejezés. Az ábra jobb oldalán a mintában szereplő C változó deklarálását láthatjuk, aminek kezdeti értéke leírja a minta kifejezésbeli szerepét.



5.6. ábra. Paraméterek mintaillesztéseiből képzett feltételek és változódeklarálás

A 5.7 ábrán az őrfeltételek *eCST* reprezentációját láthatjuk, ami jelen esetben a vagy kapcsolat miatt egy zárójeles kifejezés lett.

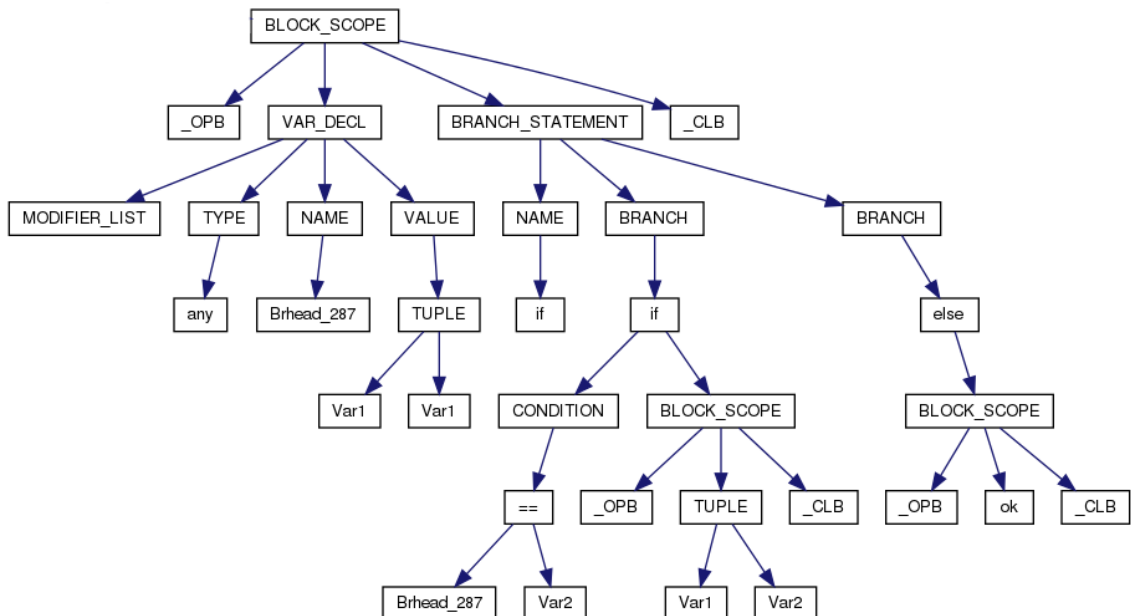


5.7. ábra. Örfeltételből generált struktúra - teszt

**Case kifejezés** Az alábbi mintakódon mutatom be a *case* kifejezés konverzióját:

```

-module(c3).
g(A, B) ->
  case {A, A} of
    B -> {A, B};
    _ -> ok
  end.
  
```



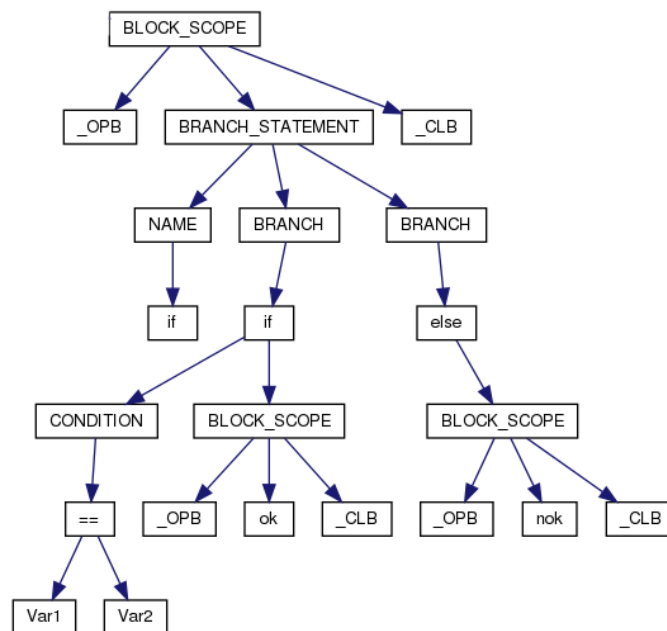
5.8. ábra. Case kifejezés - teszt

A 5.8 ábrán láthatjuk, hogy program a kifejezés fejelemét egy fiktív változónak adja értékül, és majd ezen a változón keresztül hivatkozik rá a feltételek kifejtése során. A fiktív változó deklarálása megelőzi az elágazást.

**If kifejezés** Az alábbi mintakódon mutatom be a *if* kifejezés konverzióját:

```
-module(if1).
f(A, B) ->
    if
        A == B -> ok;
        true -> nok
    end.
```

A 5.9 ábrán jól látható, hogy a generált gráf szerkezete nagyon hasonlít azon függvények grájára, amelyekben a mintaillesztés alapján ellenőriztük, a két paraméter egyenlőségét, illetve az őrfeltételek révén történt meg az ellenőrzés.



5.9. ábra. If kifejezés - teszt

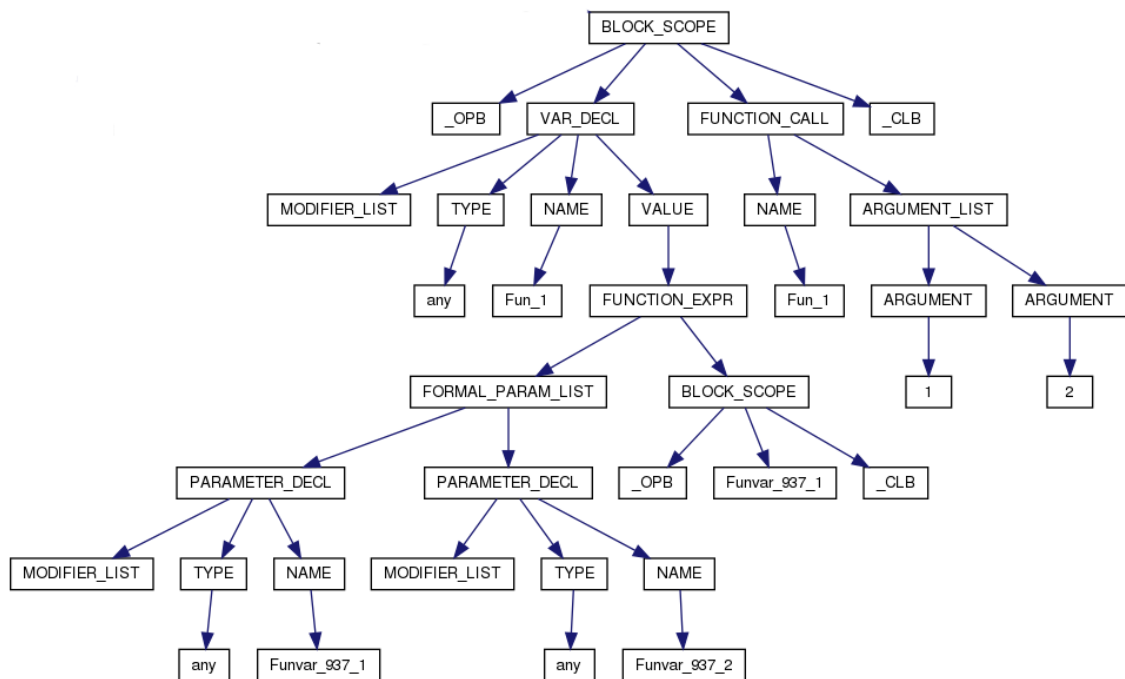
**Függvény kifejezés** Az alábbi mintakódon mutatom be a névtelen függvény kifejezés konverzióját:

```

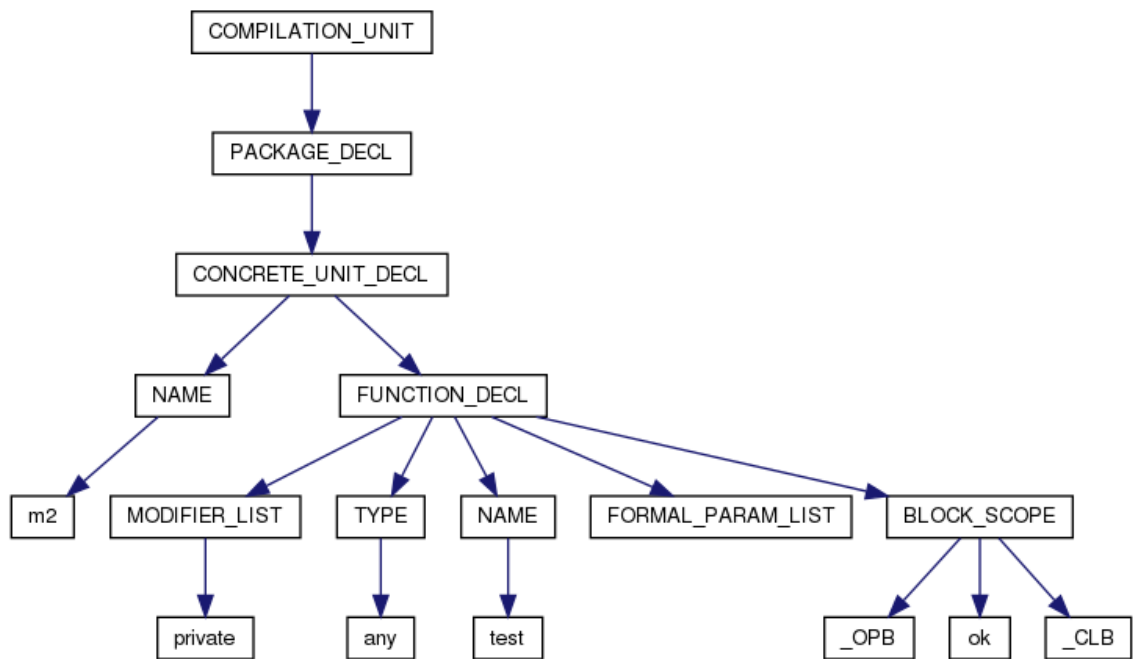
-module(fu1).
f() ->
    Fun = fun(A, B) ->
        A
    end,
    Fun(1, 2).

```

A 5.10 ábrán láthatjuk, értékadással egybekötött változó deklarálásként konvertálja a kifejezést, és a kezdeti érték lesz a névtelen függvény, aminek részfáját a "FUNCTION\_EXPR" csúcs vezeti be. A részfa felépítése ugyanolyan, mint a nevesített függvények esetében. A **Fun** változóba mentett függvény meghívásának részfája ugyanolyan szerkezetű, mint a nevesített függvények hívásainál.



5.10. ábra. Függvény kifejezés - teszt



5.11. ábra. Makró kifejezés - teszt

**Makró kifejezés** Az alábbi mintakódon mutatom be a makrók konverzióját:

```

-module(m2).
-define(Part1, test()).
-define(Part2, ) -> ok).

?Part1?Part2.

```

A fenti kódban a két makró összeillesztése egy függvényt ad, és a konvertertől elvárjuk, hogy az átalakítás során ennek a függvénynek a fáját építse meg. Az eredményt a 5.11 ábrán láthatjuk.

## 6. fejezet

# Összegzés

### 6.1. Áttekintés

Napjainkra a szoftverfejlesztés szerves részévé vált a program megértést támogató, refaktoráló szoftverek használata, amik jelentősen megkönnyítik a programozók munkáját. Ezek egy csoportját alkotják a statikus elemzők, melyeknek csupán a vizsgált program forráskódjára van szükségük az elemzés elvégzéséhez. A legtöbb ilyen eszköz egy bizonyos nyelvre specializálódott, mint például a *RefactorErl*. Ez egy nyílt forráskódú, *Erlang* programok megértését támogató statikus elemző eszköz, ami az elemzéseire *Szemantikus Program Gráf* formájában tárolja a forráskódból kinyert információt. A *SSQSA* ezzel szemben egy több nyelvet támogató eszköz, és a különböző nyelven írt programok egységes reprezentációjára alkalmas, XML-ben tárolt *eCST*-n végzi elemzéseit.

A dolgozatomban megvizsgáltam a két eszköz által használt gráf matematikai reprezentációját, valamint definiáltam egy leképezést, amely során megvizsgáltam azokat a kérdéseket, hogy hogyan lehet a jóval több információt tároló funkcionális *Szemantikus Program Gráf* elemeit megjeleníteni egy imperatív programokhoz igazított szintaxisfa (*eCST*) leíró eszközei segítségével. Bemutattam a leképezést elvégző program már implementált funkcióit a tesztelési esettanulmányon keresztül.

## 6.2. Továbbfejlesztési lehetőségek

A leképezés megvalósításával elérhetővé tehető, hogy a RefactorErl-lel elemzett programokra az SSQSA metrikáit ki lehessen számolni. A dolgozatomban ismertetett leképezés definícióját az optimálisabb eredmény elérésének érdekében bizonyos konkrét kifejezések esetében kiegészíthetjük plusz információkkal, ezáltal finomítva, pontosítva azt. Minél részletesebb, és finomabb a definíció, annál megbízhatóbb eredményt adhatnak majd a *SSQSA* elemzései.

# Ábrák jegyzéke

|                                                                              |    |
|------------------------------------------------------------------------------|----|
| 2.1. Szemantikus Program Gráf - példa . . . . .                              | 7  |
| 3.1. eCST - példa . . . . .                                                  | 12 |
| 4.1. $e \in ECST_{COMPILATION\_UNIT}$ . . . . .                              | 18 |
| 4.2. $e^{(1)} \in ECST_{FUNCTION\_DECL}$ . . . . .                           | 23 |
| 4.3. $ECST_{PARAMETER\_DECL}$ . . . . .                                      | 27 |
| 4.4. Függvényhívás . . . . .                                                 | 29 |
| 4.5. $ECST_{VAR\_DECL}$ . . . . .                                            | 31 |
| 5.1. Egyszerű függvény 1 - teszt . . . . .                                   | 44 |
| 5.2. Egyszerű függvény 2 - teszt . . . . .                                   | 44 |
| 5.3. f11:f/2 - teszt . . . . .                                               | 45 |
| 5.4. f12:f/2 - teszt . . . . .                                               | 45 |
| 5.5. Fiktív elágazás szerkezete - teszt . . . . .                            | 46 |
| 5.6. Paraméterek mintaillesztéseiből képzett feltételek és változódeklarálás | 46 |
| 5.7. Őrfeltételből generált struktúra - teszt . . . . .                      | 47 |
| 5.8. Case kifejezés - teszt . . . . .                                        | 47 |
| 5.9. If kifejezés - teszt . . . . .                                          | 48 |
| 5.10. Függvény kifejezés - teszt . . . . .                                   | 49 |
| 5.11. Makró kifejezés - teszt . . . . .                                      | 50 |

# Irodalomjegyzék

- [1] Alan J. Perlis, Frederick G. Sayward, Mary Shaw:  
*Software Metrics: An Analysis and Evaluation*, MIT Press, 1981, [404],  
ISBN-9780262160834
- [2] *Official website of RefactorErl tool*,  
<https://plc.inf.elte.hu/erlang/>, 2013.05.02.
- [3] *Erlang Programming Language website*,  
<http://www.erlang.org/>, 2013.05.02.
- [4] Francesco Cesarini, Simon Thompson: *Erlang Programming*, O'Reilly Media,  
2009, [498], ISBN-978-0-596-51818-9.
- [5] *Set of Software Quality Static Analyzers (SSQSA) - Google Project Hosting*,  
<https://code.google.com/p/ssqsa/>, 2013.05.05.
- [6] Akmal B. Chaudhri, Awais Rashid, Roberto Zicari: *XML Data Management: Native XML and XML-Enabled Database Systems*, Addison-Wesley Professional, 2003, [688], ISBN-0201844524.
- [7] *Erlang Syntax\_ Tools Reference Manual*,  
[http://www.erlang.org/doc/man/erl\\_syntax.html](http://www.erlang.org/doc/man/erl_syntax.html), 2013.05.09.
- [8] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, Roland Király: *Modeling semantic knowledge in Erlang for refactoring*, in *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, Vol. 54*, page 7-16, KEPT 2009, [10], (2009).
- [9] Németh András: *Data access optimization*, TDK dolgozat, ELTE IK, 2012.

- [10] Tóth, M. and Bozó, I.: *Static Analysis of Complex Software Systems Implemented in Erlang*, in *Central European Functional Programming Summer School – Fourth Summer School*, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743, 2012.
- [11] Zoran Budimac, Gordana Rakic, Milos Savic: *SSQSA architecture*, in *Balkan Conference in Informatics*, page 287-290, ACM 2012, [4], ISBN 978-1-4503-1240-0.
- [12] Gerlec, Č., Rakić, G., Budimac, Z., Heričko, M.: *A Programming Language Independent Framework for Metrics-based Software Evolution and Analysis*. *Computer Science and Information Systems*, Vol. 9, No. 3, 1155-1186. (2012)
- [13] *Parse tree - Wikipedia*,  
[http://en.wikipedia.org/wiki/Parse\\_tree](http://en.wikipedia.org/wiki/Parse_tree), 2013.05.06.