

TDK-dolgozat

Hack János

Kód visszagenerálása nyelvfüggetlen AST alapján

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSI NYELVEK ÉS FORDÍTÓPROGRAMOK TANSZÉK



Szerző:

Hack János

Programtervező informatikus

IV. évfolyam

Témavezető:

Szűgyi Zsolt

egyetemi tanársegéd

2012

Tartalomjegyzék

1. Bevezetés	2
2. Irodalmi áttekintés	5
2.1. Absztrakt szintaxisfa (AST)	5
2.2. A Clang elemző	6
2.3. Szoftverkarbantartást segítő eszközök	7
2.3.1. PMD	7
2.3.2. RefactorErl	8
2.3.3. DMS	8
2.4. Következtetések levonása	9
3. A feldolgozáshoz szükséges leíró nyelv (ADL)	10
3.1. Az ADL megalkotását motiváló példa	10
3.2. A nyelv szerkezete és szabályai	13
3.2.1. Inicializáló rész	14
3.2.2. Definíciós rész	14
3.2.3. Rule kapcsolók	18
3.3. Az ADL feldolgozásához szükséges adatszerkezetek	21
3.4. Az ADL fájl feldolgozása	23
4. Az új reprezentációs modell	25
4.1. A modell szerkezete	25
4.2. A modell előállítás	26
4.3. A modell kirajzolása	28
5. Összefoglalás	30
A. A leíró nyelv EBNF leírása	34
B. Logikai kifejezés feldolgozó	36

1. fejezet

Bevezetés

Napjaink szoftverfejlesztésében egyre több időt és erőfeszítést igényel a nagy, ipari méretű programkódok megértése, karbantartása, továbbfejlesztése. Ezeknek a folyamatoknak segítésére készültek statikus elemző [1], refaktoráló [2] és különböző kód megértést támogató eszközök, illetve ezen eszközök továbbfejlesztése, hatékonyabbá tétele egy fontos kutatási terület. A forráskód mint szövegfájl önmagában kevésbé alkalmas arra, hogy számításokat, transzformációkat hajtsunk végre rajta, ezért szükséges, hogy vele ekvivalens, de algoritmikus szempontból könnyebben kezelhető formára hozzuk. Egy ilyen forma az absztrakt szintaxisfa (AST) [3]. Egy adott nyelven megírt forráskódból a szintaxisfát egy elemző program készíti el, az adott nyelv nyelvtani szabályai alapján [4]. Azonban egy ipari projekt esetében gyakori, hogy különböző modulokat, más-más, a feladatukhoz jobban illeszkedő programozási nyelven írnak meg, így ezen modulok AST-i is különböző szabályok alapján épülnek fel. Ha azonban a számításainkat, transzformációinkat nem modulonként külön-külön, hanem az egész projektre egységesen szeretnénk végrehajtani, akkor a projektünk szintaxisfáit általános, nyelvfüggetlen formára kell hozni. A nyelvfüggetlen szintaxisfa alatt nem egy univerzális fát értünk, amely bármely létező programozási nyelven írt kódot képes ábrázolni, hanem egy olyan fát, amely egy konkrét nyelvek halmazán (például egy adott projektben használt néhány nyelv) írt programokat képes egységesen reprezentálni.

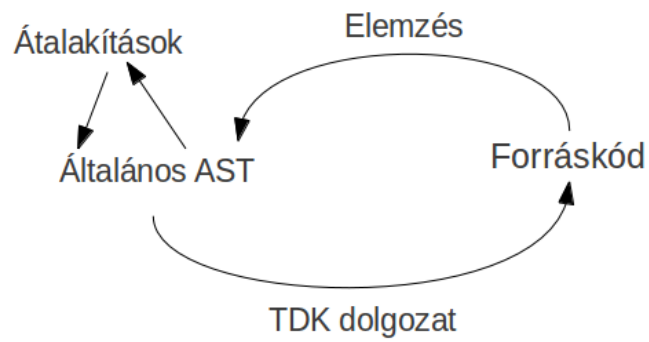
Ezen nyelvfüggetlen absztrakt szintaxisfák bemenetül szolgálnak a különböző szoftverkarbantartást segítő eszközöknek. A statikus elemzők elsősorban csak olvassák az AST-eket a kód összefüggéseinek feltérképezéséhez, metrikáinak kiszámolásához, potenciális programozói hibák felderítéséhez. Azonban léteznek olyan eszközök is – mint például a refaktorálást végző programok –,

amelyek különböző program-transzformációkat hajtanak végre, mely műveleteket az absztrakt szintaxisfán végzik. Ha az ilyen átalakított programot szeretnénk módosítani, továbbfejleszteni, akkor szükséges, hogy a transzformált AST-t visszaalakítsuk forráskóddá. Forráskód generálása az AST-ből nem triviális feladat, mert ugyan az AST egy ekvivalens ábrázolása a forráskódnak, de sok nyelvtani szabály csak implicit szerepel benne: például egy AST-ben nem tároljuk az utasítás utáni pontosvesszőt. Egy konkrét programozási nyelvhez dedikált eszköz esetében ez nem jelent olyan nagy problémát, hiszen az eszköz ismeri a nyelv nyelvtani szabályait, amelyek alapján el tudja végezni a visszaalakítást. Ha az adott nyelvben pontosvesszővel kell lezárni egy utasítást, akkor az eszköz ha egy utasítást reprezentáló AST részfához érkezik, tudja, hogy majd a szöveggé alakítás során, az utasítás végére hozzá kell fűznie azt. Egy olyan szoftverkarbantartást segítő eszköz esetében, amely nyelvfüggetlen szintaxisfákkal dolgozik ez lényegesen bonyolultabb feladat. Ebben az esetben fel kell készíteni az eszközt arra, hogy az összes olyan nyelvre amelyet támogat, felismerje és helyesen alkalmazza az AST-ben szereplő implicit szabályokat.

Dolgozatom célja egy rugalmas, egyszerűen kezelhető, könnyen kibővíthető, általános keretrendszer megalkotása, amely lehetővé teszi, hogy szintaktikusan helyes forráskódot generáljunk a nyelvfüggetlen AST-ből az eredeti programozási nyelvre. E keretrendszerhez definiálok egy leíró nyelvet amely segítségével minden általunk kezelni kívánt programozási nyelvhez meghatározhatjuk, hogy hogyan történjen a kód visszagenerálása. Ezáltal a keretrendszerünk egy új nyelvvel való kibővítéséhez, elég csak a hozzá tartozó leírófájlt definiálni, amit dinamikusan hozzáadhatunk. A rendszer képes a szintaxisfa mellett egyéb külső, számított eredmények kezelésére is, amelyeket valamilyen módon hozzákötöttünk az AST-hez. Az ilyen adatok megjelenítési módját szintén a leíró nyelvben határozhatjuk meg. Leggyakrabban használt külső adatok a magyarázó szövegek, kommentek. Ezek tipikusan nem kerülnek bele a szintaxisfába, és egy további elemzés során, külsőleg tudjuk hozzákötni őket a megfelelő AST-beli pontokhoz. Azonban a programozó számára fontos információt tartalmazhatnak, ezért a generált kódba is bele kell, hogy kerüljenek.

A dolgozatomban bemutatott keretrendszer terveink szerint beépül majd, az egyetemünkön fejlesztett kódmegértést támogató eszközbe a *Code Compass*-ba, de a moduláris felépítésének köszönhetően más termékek is alkalmazhatják.

Dolgozatom felépítése a következő: a 2. fejezetben ismertetem a dolgozat megértéséhez szükséges fogalmakat és a kapcsolódó irodalmat. A kódgene-



1.1. ábra. A probléma struktúrája

rálási szabályokat definiáló leírónyelvet a 3., a generáláshoz szükséges köztes adatszerkezet pedig a 4. fejezetben mutatom be. Végül az 5. fejezetben összefoglalom az eredményeimet.

2. fejezet

Irodalmi áttekintés

A probléma részletesebb megismerésének céljából először az AST-vel mint szerkezettel kell jobban megismerkednünk. Ezután az AST-k szerkezetét elemző eszközökkel foglalkozunk.

2.1. Absztrakt szintaxisfa (AST)

Az absztrakt szintaxisfát a forráskód szintaktikus szerkezetének ábrázolására használjuk. Mivel a szerkezete absztrakt, nem tartalmaz minden, a forráskódban megtalálható elemet. A kifejezések végét jelző tokeneket, valamint az utasításblokkokat jelzőket elhagyhatjuk, mivel a fa szerkezete alapján tudunk következtetni a helyükre. Ezért fontos különbséget tennünk az absztrakt szintaxisfa és a konkrét szintaxisfa között. A konkrét szintaxisfa vagy elemzőfa [5] egy köztes fa, amit a elemzők az AST felépítése előtt készítenek. Majd ebből további szemantikus információkkal kiegészítve létrehozzák az AST-t.

Általános AST alatt egy olyan szintaxis fát értünk, ami nyelvek egy halmazát képes ábrázolni és saját szerkezetet, speciális tulajdonságokkal is rendelkezhet.

A dolgozatban egy olyan AST-t használunk, aminek minden csúcsához tartozik egy típus, ami a kódrészlet tulajdonságaira utal. A csúcsoktól továbbá le lehet kérdezni a kódbeli szövegüket, valamint a csúcs gyerekeinek számát, típusát és szülőjének típusát. A dolgozatban leírtak ezen tulajdonságokat minimum elvárásnak tekintik ahhoz, hogy az AST-t vissza tudjuk alakítani forráskóddá. A csúcsok továbbá tartalmazhatnak más információkat is: például a kódrészlethez tartozó magyarázó szöveget, vagy egyéb speciális információt, mint például a csúcshoz tartozó, csak AST-ben jelen lévő megjegyzést. A

keretrendszerünk ezen információk megjelenítésére is képes.

Mintakép az általunk használt AST-ről:

Name	Type
▼ g	Function
▼ int (int)	FunctionProto
T int	Builtin
▼ i	ParmVar
T int	Builtin
0 IntegerLiteral	IntegerLiteral
▼ {} compound statement	CompoundStmt
▼ declaration	DeclStmt
▼ o	Var
class Obj	Record
▼ CXXConstructExpr	CXXConstructExpr
0 IntegerLiteral	IntegerLiteral
▼ call Obj::lol	CXXMemberCallExpr
▼ Obj::lol	MemberExpr
0 o	DeclRefExpr
▼ return	ReturnStmt
0 IntegerLiteral	IntegerLiteral
▼ f	Function
▼ int (void)	FunctionProto
T int	Builtin
▼ {} compound statement	CompoundStmt
▼ call g	CallExpr
▼ () implicit cast	ImplicitCastExpr
0 g	DeclRefExpr
CXXDefaultArgExpr	CXXDefaultArgExpr
▼ call g	CallExpr
▼ () implicit cast	ImplicitCastExpr
0 g	DeclRefExpr
▼ call f	CallExpr

2.1. ábra. AST részlet

2.2. A Clang elemző

A Clang [6] egy C, C++, Objective-C, Objective-C++ fordító ami az LLVM-et [7] használja backend-nek. A Clang kifejlesztésének célja, egy olyan program készítése amivel a forráskódok gyorsan lefordíthatóak kevés memóriaigénnyel, valamint a fordítás közben talált hibákra felhasználóbarát visszajelzést ad. A célok közé tartozik még a GCC-vel való [8] kompatibilitás és a nyílt forrású kódbázis.

A Clang statikus elemző moduljának segítségével képesek vagyunk egy programból többféle metrika előállítására ami a forráskód egy jellemzőjét mutatja. Ezen felül olyan magas szintű szolgáltatásokat is biztosít amely lehetővé

teszi többek között a halott kód detektálását vagy a programtervezési minták [9] felismerését. A program továbbá nyílt forráskódú, ennek köszönhetően tetszőlegesen testreszabhatjuk a saját projektjeinkben történő használat során.

A dolgozatban példaként bemutatott C++ AST-t a Clang elemző segítségével állítjuk elő. Az előállításnál erősen támaszkodunk ezen eszköz fa bejáró és forráskód elemző moduljára.

2.3. Szoftverkarbantartást segítő eszközök

A fejezet célja olyan, az iparban használt szoftvereszközök bemutatása amik a forráskód statikus elemzését végzik.

2.3.1. PMD

A PMD egy statikus Java elemző [10], ami szabályok alapján képes problémákat felderíteni a kódban mint például:

- Üres *switch*, *try*, *catch*, *finally* blokkok
- Nem használt változók
- Üres *if*- és *while*-feltételek
- Kód ismétlés

A PMD tartalmaz egy sor beépített szabályt az ellenőrizendő szabályok írására. A szabályokat írhatjuk az XPath segítségével, vagy Java-beli osztályokkal, amiket hozzá kell adni a szabálykészlet XML fájlhoz, hogy használni tudjuk azokat.

A PMD elsősorban kódolási szabályok megsértésének felfedezésére és egyéb statikus elemzésekre hívatott. Az elemzés során talált hibák nem fordítási hibák, de a program futási hatékonyságát befolyásolhatják. A PMD nem van egy kód másolást ellenőrző kiegészítője amit CPD-nek hívnak és a Rabin-Karp mintaillesztő algoritmust [11] használja a másolatok felderítésére. A CPD működik Java, JSP, C, C++, Fortran, PHP, valamint C# forráskóddal is. A PMD több népszerű fejlesztő környezet által is támogatott mint például: Eclipse, NetBeans/Sun Studio, JBuilder. A PMD az elemzéseket nem közvetlenül a forráskódon hanem a javaCC [12] által generált AST-n végzi.

A jelenlegi verziója az eszköznek csak egy forráskódon képes dolgozni egyszerre, így a fájlok közti összefüggések felfedezésére nem képes, valamint csak limitált típusú, szabály feldolgozására képes. Az eszközzel továbbá lehetséges a forráskód visszaalakítása, valamint másik nyelvű kóddá alakítására is, de ez a lehetőség is csak egy speciális AST-n, limitált körülmények között használható.

2.3.2. RefactorErl

A projekt célja egy olyan eszköz kifejlesztése ami az Erlang nyelven [13] írt programok statikus elemzéséhez és refaktoráláshoz nyújt segítséget. A programban jelenleg támogatott funkciók közül a fontosabbak:

- Felhasználó által definiált szabályok ellenőrzése
- Függőség vizsgálat
- Mellékhatásvizsgálat
- Adatfolyam és dinamikus függvényhívás vizsgálat
- Eclipse kiegészítő
- Többféle model tárolási lehetőség

2.3.3. DMS

A DMS egy sor olyan eszközkészletet biztosít ami forráskód automatikus elemzésére, módosításra szolgál [14]. A DMS segítségével implementálhatunk olyan statikus elemző eszközöket amik különböző programozási nyelveken írt programok támogatására is képesek. Az elemzések elvégzéséhez biztosít egy automatikusan épített AST-t, amit vissza is tud alakítani forráskóddá. Az eszközkészlet további lehetőségei közé tartozik a programkód valamilyen forrásnyelvről célnyelvre történő átalakítása, valamint nyelvi struktúrák ekvivalens átalakítása. Az ilyen átalakításhoz egy saját leírást használ, amivel meg tudjuk adni, hogy a forrásnyelv utasításait hogyan alakítsuk át célnyelv utasításkészletére, vagy hogyan helyettesítsük egy nyelvi kifejezést egy másikkal. Az alábbi példa bemutatja hogyan alakítsuk át a C nyelvi if utasítást ?: utasítássá.

```
rule simplify_conditional_assignment(v:left_hand_side,  
    e1:expression,e2:expression,e3:expression)
```

```

:statement->statement
=  " if (\e1) \v=\e2; else \v=e3; "
-> " \v=\e1:?\e2:\e3; "
if no_side_effects(v);

```

Ennek az eszköznek a forráskóddá alakító része már közel áll a dolgozatban célul kitűzött módszerhez, azonban bármennyire is rugalmas az eszköz csak a saját elemzője által épített AST-re, vagy egy másik AST ilyen formára hozott változatára képes ezeket az átalakításokat elvégezni. Az eszköz azonban nem nyílt forráskódú így nem tudjuk tetszésünk szerint testre szabni az AST visszaalakítását.

2.4. Következtetések levonása

Az iparban számos eszköz szolgál a forráskód statikus elemzésére valamilyen AST segítségével. Az eszközök egy része még az AST forráskódként történő átalakítására is képes. Azonban ezek az eszközök nagyon limitáltak, csak egy specifikus AST-vel képesek működni ami a saját szabályait szerint építettek fel. Hiány van olyan eszközből ami képes általános AST-t forráskóddá alakítani.

3. fejezet

A feldolgozáshoz szükséges leíró nyelv (ADL)

Mint azt a bevezetőben is említettük a forráskódból generált AST-k nem tartalmaznak minden szükséges információt ami a kód szövegének teljes visszaállításához szükséges. Ennek a problémának az áthidalására definiáltunk egy leíró nyelvet, az ADL-t (AST Description Language), amely meghatározza, hogy hogyan kell a fát helyes pozíciókkal forráskóddá visszaalakítani. Az ADL tervezésének legfőbb célja az volt, hogy hatékonyan és a fa szerkezetéhez közel álló módon tudjuk ezen szabályokat megfogalmazni. Ezáltal a meglevő AST, valamint a hozzá tartozó leírás segítségével már képesek leszünk a forráskód visszaállítására. Fontos azonban, hogy az AST amit leírunk erősen típusos legyen és egy típushoz csak egyféle kifejezés tartozzon. Ez a legtöbb fa esetén nem teljesül. *Például egy változó deklarációhoz tartozó csúcs sokszor csak implicit módon tartalmazza, hogy egy egyszerű változó vagy egy tömb deklarációjáról van szó.* Ehhez szükséges, hogy az AST bejárása közben az ilyen csúcsokat egyértelműsítsük, új típusok hozzárendelésével. Ezekkel az új típusokkal már könnyedén készíthetünk olyan leírást amivel a fa visszaalakítható. A leírónyelv EBNF [15] formában megadott szintaxisa megtalálható az A függelékben.

3.1. Az ADL megalkotását motiváló példa

Egy egyszerű példán keresztül szeretnénk bemutatni egy forráskódhoz tartozó AST szerkezetbeli hiányosságait, amik megnehezítik az átalakítást, illetve a szintaxisfa forráskóddá való visszaalakítását.

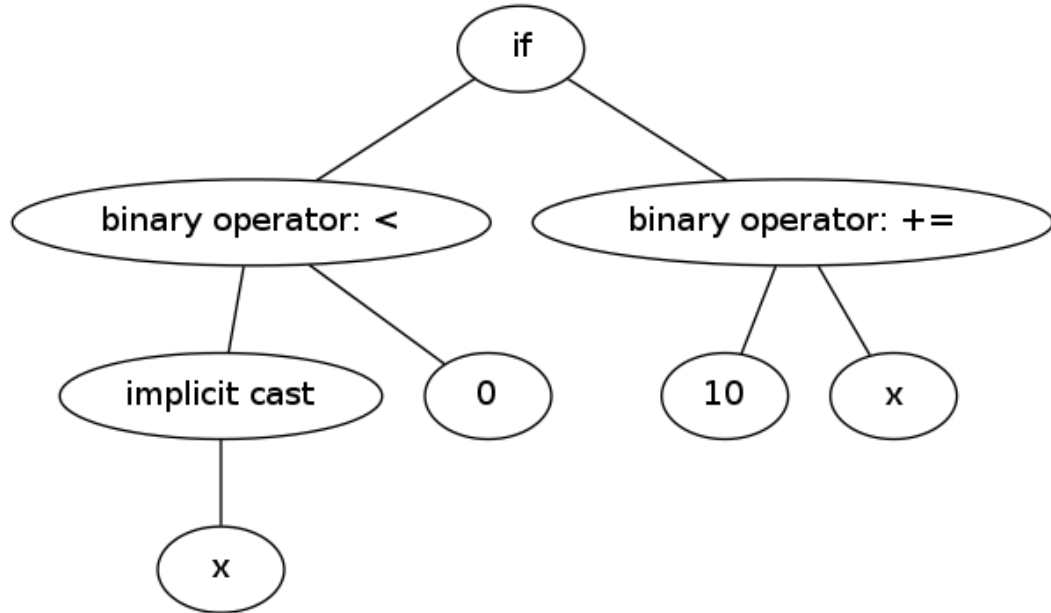
Tekintsük az alábbi C++ nyelven írt egyszerű kódrészletet:

```

0 | if (x > 0)
1 |   x += 10;

```

Ehhez a kódhoz tartozó AST a 3.1. ábrán látható.



3.1. ábra. A példakód AST reprezentációja

Az AST és a forráskód közti egyik legfeltűnőbb változás, hogy az AST-be bekerül egy `implicit cast` csúcs az `x` változó szülőjeként. Ez ugyan szükséges elem a fordításhoz, de mivel a forráskódban explicit nem szerepel, a visszaalakítás során figyelmen kívül kell hagynunk. Ha tovább keressük az eltéréseket észrevehetjük hogy a szeparátor karakterek `;`, `,`, `(`, `)` pedig ki-maradnak a szintaxisfából. Ezeket viszont a visszaalakítás során bele kell generálni a kódba.

Ezen különbségek kiküszöbölésére készíthetünk egy ADL leírást, amely alapján már az AST-ből szintaktikailag helyesen vissza tudjuk állítani az eredeti forráskódot. A példakód egy lehetséges ADL leírása:

```

%symbol endSymbol ";"
%rule ifRule ifStmt
%rule binOp BinaryOperator
%rule implCast ImplicitCastExpr

%%
ifRule:

```

```

    r. "(" r r. ")"
    d -r.endSymbol
;

binOp:
    l r
;

impCast|SKIP:
    l
;

```

A leírás két részből áll amelyet %% jelekkel választunk el egymástól. Az első részben szimbólumokat definiálhatunk és szabályokat deklarálhatunk. A szimbólumok olyan tokeneket definiálnak, amelyeket a forráskóddá alakításakor meg kell jelenítenünk. Ilyen szimbólum a példánkban az utasítást lezáró pontosvesszőt reprezentáló `endSymbol`. A szabályok deklarálásával kiíró szabályokat tudunk hozzárendelni az AST csúcsaihoz. Ezután ha a generáló algoritmus bejárja a fát, akkor minden csúcsra a hozzárendelt szabályt tudja alkalmazni. Jelen példában az elágazáshoz, a bináris operátorhoz és az implicit konvertáló kifejezéshez kötöttük hozzá sorrendben az `ifRule`, `binOp`, `implCast` szabályokat. Ezek a szabályok azt írják le, hogy az adott csúcsot hogyan alakítsuk vissza forráskóddá.

A szabályok definiálása a második részben történik. A szabály neve utáni sorban határozzuk meg, hogy a szabályhoz tartozó csúcshoz képest hogyan jelenítsük meg annak gyerekeit. A `binOp` szabály definíciójában levő `l r` leíró azt definiálja, hogy egy bináris operátornak két gyereke van, és úgy kell megjeleníteni őket, hogy először balra írjuk ki az első gyerekét, majd kiírjuk magát az operátort és utána jobbra, a második gyerekét. Lehetőségünk van különböző kapcsolókat is beállítani a szabályokon. Az `impCast` szabály `SKIP` kapcsolója azt írja le, hogy a csúcs nem kerül bele a második modellbe csak a gyerekei, már mennyiben megadtuk azok elhelyezkedését. A `SKIP` kapcsolós elem alatti pozíciók át lesznek alakítva a kihagyott elem pozíciójának függvényében. A legbonyolultabb szabály az `ifRule`, az elágazáshoz tartozó szabály. Az elágazás csúcsnak is két gyereke van. A kiírása pedig a következő módon történik:

a szabály az `r` leíróval kezdődik ezért először a csúcshoz tartozó szöveget, azaz az `if`-et írjuk ki, ezután tőle jobbra következik a nyitó zárójel, majd az elágazáshoz tartozó első gyerek, azaz az elágazás feltétele, és végül egy csukó zárójel. A `d` azt jelenti, hogy az elágazás második gyereket, azaz a törzsét, egy új sorba írjuk ki. A végére pedig pontosvesszőt kell tennünk. Ez utóbbit a `-r.endSymbol` definiálja. Ha egy leíró a mínusz jellel kezdődik, akkor az azt jelenti, hogy nem a szülőhöz, hanem az előző gyerekhez képest szeretnénk meghatározni az adott gyerek elhelyezkedését.

3.2. A nyelv szerkezete és szabályai

A leírás közben használt példák a C++ nyelv, valamint az abból a Clang elemző által generált AST alapján készültek.

Jelkészlet:

```
0 1 2 3 4 5 6 7 8 9
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
| _ [ ] ( ) : ; . , * - = > <
```

Foglalt szavak:

```
UFI, ULA, DFI, DLA, LFI, LLA, RFI, RLA, CHILD_TYPE, ITEM,
PARENT_TYPE, CHILD_COUNT, COND, %symbol, %rule, SKIP, skip,
LINEAR, V_DIRECTPOS, H_DIRECTPOS, CONTAINER, l, r, d, w,
and, or, not, normal, reverse
```

Operátorok és egyéb elemek:

```
!=, %, <=, >=, /*, */, //
```

3.2. ábra. Az ADL nyelv tokenjei

A leírás két részből kell, hogy álljon amit a `%%` operátorral választunk el. Az első részt inicializáló résznek a másodikat definíciós résznek nevezzük. A leírásban megengedett a C/C++ stílusú kommentek írása.

3.2.1. Inicializáló rész

Az inicializáló részben szimbólumokat definiálhatunk amikre később a szabályokban hivatkozni akarunk, valamint itt deklaráljuk ezeket a szabályokat is. A szabályok deklarálásánál azt mondjuk meg, hogy a fában milyen elemre alkalmazzuk azt. Ez egy szövegillesztés lesz a fában levő elem típusára. Ebben a részben vezethetünk be szimbólum neveket, melyek a szabályok törzsében szerepelhetnek. A szimbólum definíció egy névből és egy (" ") között megadott karaktersorozattól áll. A szimbólum nevek a modell építés során helyettesítésre kerülnek a (" ") között megadott karaktersorozattal. Példa szimbólum definícióra:

```
%symbol StatementEnd ";"
```

Továbbá itt szerepelhetnek még a szabály deklarációk, amik azt mondják meg, hogy melyik szabály alkalmazandó milyen típus esetén. A szabályokat a nevükkel és a hozzájuk rendelt típusok listájával adhatjuk meg. Példa egy szabály deklarációra:

```
%rule statementRule statement,ifStatement,...
```

A példában az `statementRule` nevű szabályt az egyszerű utasításokra és az elágazásokra alkalmazzuk.

3.2.2. Definíciós rész

Ebben a fejezetbe kerül a már deklarált szabályok definíciója. A szabályok kétféle szerkezetűek lehetnek egyszerűek vagy feltételesek. Az egyszerű szabályok törzse csak egyféle leírást tartalmaz a feltételes szabályok a feltételek függvényében több törzsszel rendelkezhetnek.

Egyszerű szabályok

Példa egy egyszerű szabályra:

```
ruleName|kapcsoló1|kapcsoló2:  
    itemdesc1 itemdesc2 ...  
;
```

A szabályok belsejében az `itemdesc`-ek sorrendje a fa bejárás sorrendjéhez kell, hogy illeszkedjen. Az `itemdesc`-ekben három fő tulajdonságát írhatjuk le egy elem elhelyezésének: pozíció, szöveg, ismétlés.

A pozíció az elem szülőhöz viszonyított elhelyezkedését definiálja. Négyféle pozíció megadása engedélyezett:

- d (down): az adott elem alá.
- u (up): az adott elem fölé.
- l (left): az adott elemtől balra.
- r (right): az adott elemtől jobbra.

A pozíciók előtt továbbá alkalmazható a kötőjel (–) ami, azt fejezi ki, hogy az elemet a mellette lévőhöz képest kell elhelyezni nem a szülő alapján. Ezen kívül létezik nyolc speciális pozíció ami mindig ugyanarra a helyre mutat:

1. UFI (upward first): az adott elemtől felfelé az első pozíció
2. ULA (upward last): az adott elemtől felfelé az utolsó pozíció
3. DFI (downward first): az adott elemtől lefelé az első pozíció
4. DLA (downward last): az adott elemtől lefelé az utolsó pozíció
5. LFI (left first): az adott elemtől balra az első pozíció
6. LLA (left last): az adott elemtől balra az utolsó pozíció
7. RFI (right first): az adott elemtől jobbra az első pozíció
8. RLA (right last): az adott elemtől jobbra az utolsó pozíció

Létezik még egy speciális elem a `skip` aminek használatáról a `LINEAR` kapcsolónál írunk. A pozíció megadásával akár be is fejezhetjük az aktuális elem leírását, vagy ha ez nem elég folytathatjuk az elem testreszabását az elem második részében.

A szöveg a pont (.) elválasztó után adható meg. Ebbe a részben használhatjuk az ADL leírás első felében definiált szimbólumokat, vagy írhatunk be konkrét szöveget (") között. Ebben a részbe továbbá megadható még egy speciális elem az `ITEM`, lásd később `V_DIRECTPOS` `H_DIRECTPOS` kapcsolók leírása.

Az ismétlés nem a fabeli elem ismétlését jelenti, hanem az elemleírás kerül újra felhasználásra a következő gyerek csúcsnál. Az ismétlés az elem után írható a következő formában:

```
[  
  <ismétlésszám>  
  |  
  <ismétlendő elemek elhelyezkedési iránya>  
]
```

Az ismétlésszám megadása kötelező, értéke a $\mathbb{N}^+$ halmazból kerülhet ki, vagy lehet csillag (*), ami azt jelenti, hogy az adott elem után minden gyerekre ez az elhelyezés alkalmazandó. Az irány opcionális és a következő két értéket veheti fel:

- normal: olyan irányban lesznek az elemek egymás után, ahogyan jöttek
- reverse: az elemek ellenkező sorrendben lesznek a bejárásbeli sorrendjükhöz képest

Az irány kihagyása esetén a normal érték az alapértelmezett.

C++ kód:

```
int a=5;
```

AST szerkezet:

```
var (a) -> builtin (int)  
        -> integerliteral (5)
```

ADL leírás:

```
varRule:  
  l r."=" r r."; "  
;
```

3.3. ábra. Példa egy egyszerű szabályra

A szabályokban alkalmazhatunk még blokkokat, azaz több item-et blokkba foglalhatunk és azokra alkalmazhatunk ismétlést. A blokkok törzsét kapcsos zárójelek közé írjuk. Az ismétlés ugyanúgy adható meg mint az `itemDescek`-nél a harmadik rész. A blokkok nem alkalmazhatók `V_DIRECTPOS`, illetve

H_DIRECTPOS kapcsoló használata esetén és a speciális pozíciók (lásd szabályok inicializáló része) sem alkalmazhatóak a törzsükben.

C++ kód:

```
int foo(int arg1, int arg2);
```

AST szerkezet:

```
function -> functionproto(foo)
          -> paramvar -> arg1 -> int
          -> paramvar -> arg2 -> int
          -> int
```

ADL leírás:

```
function|CONTAINER:
  1
;
functionproto:
  1 r."(" r {r.", " r}{[*] r.}" r.);" r.";"
;
```

3.4. ábra. Példa a blokkok használatára

Feltételes szabályok

A feltételes szabályok abban különböznek az egyszerűektől, hogy törzsükben feltételek sorozata választ el több részt és az a rész fog végrehajtódni aminek a feltétele teljesül. Ha több feltétel is teljesülne akkor az első illeszkedő fog végrehajtódni. A feltételes szabály a következőképpen néz ki:

```
rule1:
  COND (...)
  |
  COND (...)
  |
;
```

A '|' után megadhatunk újabb feltételt vagy írhatunk egy szabálytörzset ami azt a viselkedést határozza meg ami a feltétel nem teljesülése esetén fog végrehajtódni. Ha a '|' jel után újabb COND() '|' rész következik akkor a

fentebb leírt működés gyűrűzik tovább. A feltételes szabályok törzse a más nyelvekben megtalálható `if(...) else if(...) else ...` szerkezetnek felel meg. A `Cond` után a zárójelbe speciális kifejezéseket írhatunk. A kifejezésekben a megszokott logikai műveletek (`or`, `and`, `not`) valamint egyenlőség vizsgálat (`=`, `!=`) engedélyezett. A logikai műveleteken bal zárójelezés az alapértelmezett, de tetszőlegesen átzárójelezhetjük a kifejezést. Egy kifejezést szabályosnak tekintünk, ha az adott művelet mind a két oldalán egyező típusú kifejezés szerepel. Ilyen kifejezések a következőképpen írhatók: `name =`, `!= name`, `num !=`, `=`, `>=`, `<=`, `>`, `<` `CHILD_COUNT` és ezeknek a logikai műveletekkel kombinációi. A `CHILD_NUM` az elem gyerekeinek a számát adja vissza. A `name` bármi lehet, de ha olyan nevet adunk meg ami nem típusnév akkor a viselkedés a kirajzolásnál nem definiált. Használhatjuk még a következő két speciális változót: `CHILD_TYPE` ami az aktuális elem első bejárás szerinti gyerekének típusa, amire a szabályt alkalmazzuk, valamint `PARENT_TYPE` ami a szülő típusa.

3.2.3. Rule kapcsolók

A szabályokra az alábbi kapcsolókat alkalmazhatjuk:

- `CONTAINER`: Az elem bekerül a második modellbe mint befoglaló elem, de nem kerül kiírásra.
- `SKIP`: Az elem nem kerül bele a második modellbe. Ha vannak gyerekei, akkor azokon folytatódik a feldolgozás.

Példa ilyen elemre:

```
int a=5;
double b=a;
```

Ebben az utasításban az AST-be bekerül egy `(int)` `implicitCast` elem az értékadás jobb oldalára. A fenti utasítások visszaalakítása a következőképpen nézne ki:

```
int a=5;
double b=(int)a;
```

Az itt bekerült elemnek saját típusa van, amire írhatunk egy olyan szabályt ami az alatta levő fán tovább megy, de ezt az elemet nem fogja

C++ kód:

```
void proc();
int foo(int arg1, int arg2);
```

AST szerkezet:

```
function -> functionproto(foo) -> int
function -> functionproto(foo)
           -> paramvar -> arg1 -> int
           -> paramvar -> arg2 -> int
           -> int
```

ADL leírás:

```
function|CONTAINER:
  1
;
functionproto:
COND (CHILD_COUNT >= 1)
  1 r."(" r {r."", " r}{[*] r.}" r.);"
  |
  1 r."(" r.*)" r.);"
;
```

3.5. ábra. Példa a feltételes szabály definiálására

kirajzolni és nem is kerül be a második modellbe. A SKIP kapcsolós szabályok törzsét meg kell írni, ha tovább akarunk menni a fa alattuk levő részén. Ennek segítségével visszakaphatjuk az eredeti formáját a kifejezésnek, már amennyiben azt akarjuk.

- **LINEAR:**

Ezzel a kapcsolóval azt mondjuk meg, hogy az adott elem alatti részfat úgy tekintjük, mintha az összes elem ami benne szerepel elrendezéstől függetlenül az adott elem közvetlen gyereke lenne. Ebből az következik, hogy rájuk már nem alkalmazunk további szabályt. Jelen esetben azonban az alsó elemek is ki lesznek rajzolva csak a LINEAR kapcsolós szabálynál mondjuk meg hova kerülnek. Az ilyen elemek esetén van egy speciális pozíció a SKIP ami azt jelenti, hogy az adott elemet kihagyjuk. Ismétlés megadása ebben az esetben nem megengedett.

C++ kód:

```
int t[5];
```

AST szerkezet:

```
declStmt->var (t) -> constant array (int [5])
               -> builtin (int)
               -> integerliteral (5)
```

ADL leírás:

```
varRule|LINEAR:
    skip 1 r."[" r r."]" r.";"
;
```

3.6. ábra. Példa a LINEAR kapcsolóra

- V_DIRECTPOS és H_DIRECTPOS:

Ezzel a kapcsolóval direkt pozíciókat definiálunk az elemek elhelyezkedésére vonatkozóan: konkrét számokkal definiáljuk a kiírási sorrendjüket. Az ilyen szabályoknál kötelező szerepeltetnünk az `ITEM` változót ami az aktuális csúcsot jelképezi, amire a szabályt alkalmazzuk. Ennek az `ITEM` változónak a helyét a következő módon adhatjuk meg: `pos.ITEM`, ahol a `pos` eleme a $\mathbb{N}^+$ halmaznak. A pozícióknak az $1, 2, \dots, n+1$ sorozat egy permutációjának kell lenniük, ahol az n a gyerekek száma. Az elhelyezését irányát a kapcsoló határozza meg:

- V_DIRECTPOS (vertical directpositions): Az elemek elhelyezése függőlegesen értendő
- H_DIRECTPOS (horizontal directpositions): Az elemek elhelyezése vízszintesen értendő

Az itt leírt szabályok alapján készült a leírás és az AST segítségével már fel tudjuk építeni az általános modellt, ami alkalmas többféle megjelenítésre.

3.3. Az ADL feldolgozásához szükséges adatszerkezetek

Az ADL nyelv segítségével már tudunk készíteni olyan leírást, amivel az AST-ből hiányzó információkat pótoljuk. A feldolgozáshoz szükséges egy olyan program ami képes a leírásunkat algoritmusokban jól alkalmazható szerkezetű alakítani. Ennek a programnak létre kell hozni egy kapcsolótáblát amit az AST-bejárás közben tudunk alkalmazni az egyes csúcspontokra. A táblának legmegfelelőbb adatszerkezet az asszociatív tömb amiben a csúcspontok típusa képezi a kulcsértéket az adat rész pedig az alkalmazandó szabályból készült struktúra kell, hogy legyen. A tábla segítségével az új modell építő algoritmusunk a kifejezések típusa alapján ki tudja választani melyik szabályt kell rá alkalmazni. Ahhoz, hogy ezt a táblát elkészítsük szükségünk van egy a szabályokat ábrázoló rekord leírására, hogy hogyan kell egy ADL elemleírásból egy jól alkalmazható struktúrát készíteni. Ennek a rekordnak tartalmaznia kell a szabály után alkalmazott kapcsolók listáját, valamint a benne szereplő elemleírások listáját és hogy milyen feltétel mellett választjuk ki adott elemleírás listát. Az elemleírások ismételhetők és az ismétlésnek is több fajtája lehet. Tudnunk kell, hányszor kell az aktuális leírást egymás után alkalmaznunk és ismerünk kell az ismételt elemek egymás utáni elhelyezésének módját is. Tehát az ismétléseket is rekordként kell ábrázolnunk aminek két mezője az ismétlés száma, valamint az elemek elhelyezésének iránya. A szabályok leírásánál használt elemlista egy rekordpárt kell hogy tartalmazzon, az elem leírását magát és a hozzá tartozó ismétlésleíró. Az ADL elemek leírásában a következő információkat kell szerepeltetnünk:

- Pozíció: lehet szám vagy irány a szabályra alkalmazott kapcsolók függvényében.
- Pozíció módosító kapcsoló: A pozíció megadásánál használtuk-e a "-" kapcsolót az irány előtt.
- Új elem információk: Egy kapcsoló ami megmondja, hogy új-e az elem és ha igen akkor a szövegét is tárolnunk kell.
- Blokk információk: Egy kapcsoló ami megmondja, hogy elem blokk-e és ha igen akkor a benne szereplő elemek listáját is tárolnunk kell.

- Speciális elem kapcsoló: Megmondja, hogy az elem "skip" elem-e (lásd ADL leírás LINEAR kapcsoló).

A szabályok elemleírás listája is egy rekordpárokból épül fel. A rekord első mezője az ADL leírásban megadott feltételt kell tartalmaznia, a második mezőnek pedig az elemek listáját. Az imént leírtak egy lehetséges implementációja C++ nyelven:

```

0 struct AdlItem;
1
2 typedef std::pair<int, std::string> Repeat;
3 typedef std::pair<AdlItem, Repeat> AdlItemListElement;
4 typedef std::list<AdlItemListElement> AdlItemList;
5 typedef std::pair<std::string, AdlItemList> BodyElement;
6 typedef std::list<BodyElement> Body;
7
8 enum Options{noOption, skip, linear,
9             container, vdirectpos, hdirectpos};
10
11 struct AdlItem
12 {
13     int numPos;
14     std::string pos;
15     bool relative;
16     bool newItem;
17     std::string text;
18     bool isBlock;
19     AdlItemList contains;
20     bool skip;
21     AdlItem(std::string pos_ = "i",
22             bool relative_ = false,
23             bool newItem_ = false,
24             const std::string& text_ = std::string()):
25         numPos(-1), pos(pos_), relative(relative_),
26         newItem(newItem_), text(text_), isBlock(false),
27         skip(false) { }
28 };
29
30
31 struct RuleDesc
32 {
33     std::list<Options> options;
34     Body body;

```

```

35 |
36 | RuleDesc() {}
37 | RuleDesc(const std::list<Options>& options_,
38 |          const Body& body_)
39 |     : options(options_), body(body_)
40 | {}
41 | };

```

Az itt bemutatott táblát az ADL fájl feldolgozása közben kell felépíteni.

3.4. Az ADL fájl feldolgozása

Az iménti fejezetben leírt struktúrákat az ADL nyelvi fájl feldolgozása közben építjük. Ez a feladat többféleképpen is megvalósítható. Egy lehetséges stratégia az alulról felfelé haladó módszer, amire egy építési sorrend a következő:

1. Elkészítjük a leírás első felében szereplő típusról szabályra mutató asszociatív tömböt.
2. Felépítjük a szabályok törzsében szereplő elemeket, kitöltve a pozíciót és az esetleges szimbólumot.
3. Elkészítjük az elem mellé az ismétlést leíró rekordot, majd hozzávesszük az így elkészült adatszerkezetet az elemek listájához.
4. Elemezzük az esetleges feltételt ami a szabálytörzsének ezen részéhez tartozik és elhelyezzük a szabályleírást ábrázoló listában a hozzá tartozó elemlistával együtt.
5. Felépítjük a szabálynál alkalmazott kapcsolók listáját.
6. Elhelyezzük a szabályleírást, a szabályokat az elemek típusával azonosító asszociatív tömbben.

A szabályok leírásának beazonosítását végezhetjük két lépésben, először a szabály nevét azonosítjuk a típusról, majd a szabálynév alapján azonosíthatjuk a hozzá tartozó leírást. A kétlépéses azonosítás hivatkozás szerinti átadás hiányában szenvedő nyelvek esetén kevesebb memóriaigénnyel valósítható meg. A szimbólumok feldolgozásáról még nem mondtunk semmit az eddig leírtak során. A szimbólumok az elemleírások második részében szerepelhetnek, hiányzó elemek pótlására. A feldolgozás során a leggyorsabban egy asszociatív tömb

segítségével kezelhetőek, ahol a szimbólum nevével azonosítjuk a helyettesítendő szöveget. A szimbólumok tömbje gyorsan felépíthető, mivel a leírófájl elején található. Így a szimbólum nevének későbbi előfordulása esetén azonnal helyettesíthető a hozzá tartozó szöveggel. Az itt leírt szerkezetek egy lehetséges megvalósítása C++ nyelven:

```
0 class AdlBuilder
1 {
2 public:
3     void addTypeToRule(const std::string&, const std::string&);
4     void addRuleFeloldo(const std::string&, const RuleDesc&);
5
6     RuleDesc* ruleMapper(std::string);
7
8 private:
9     std::map<std::string, std::string> typeToRule;
10    std::map<std::string, RuleDesc> ruleToDesc;
11 };
```

Most már rendelkezésünkre áll az AST és egy jól alkalmazható leíró szerkezet, ami alapján megalkothatjuk a második modellt.

4. fejezet

Az új reprezentációs modell

4.1. A modell szerkezete

Az új modell egy fa, ami közel áll az eredeti szerkezetéhez, azonban már tartalmazza a kirajzoláshoz szükséges információkat. Ezen felül tetszőleges extra információ hordozására is képes. Az új AST lényege az, hogy használója definiálhatja, hogy milyen egyéb információt tartalmazzon egy csúcs, amit még ki akarunk rajzolni a szöveg mellé. Az ilyen információ lehet például egy csúcshoz tartózó megjegyzés, amit például egy ikonnal akarunk megjeleníteni a kirajzolás során a csúcs szövege mellett.

Az új fában az elemek sorrendje megváltozhat, hogy jobban illeszkedjen a kirajzoló algoritmushoz. A csúcsok amik ebben a modellben szerepelnek legalább a következő információkat kell hogy tartalmazzák a forráskóddá alakításhoz:

- A szöveg amit ki kell írni.
- Az elem típusa. A kirajzolás többféleképpen is történhet és a típus alapján döntjük el, hogy melyik sablont választjuk.
- Az elem pozíciója a szülőhöz képest.
- Pozíció kapcsoló: igaz érték esetén az elem pozícióját a szomszédjához képest kell értelmezni.
- Befoglalási kapcsoló: igaz érték esetén az elem csak befoglaló, nem kell kirajzolni a szövegét.
- Hivatkozás a szülőre.

- Hivatkozás a gyerekekre.

Egy csúcs lehetséges implementációja C++-ban:

```

0 struct ModelNodeBase
1 {
2     std::string text;
3     std::string itemType;
4     char pos;
5     bool container;
6     bool relative;
7     ModelNodeBase* parent;
8     std::list<ModelNodeBase*> childs;
9     ModelNodeBase(std::string text_ = "",
10                  std::string itemType_ = "",
11                  char pos_ = 'i' ):
12         text(text_), itemType(itemType_), container(false),
13         pos(pos_), relative(false), parent(0) { }
14 };

```

Az elemek elhelyezkedése a LINEAR kapcsolós szabályok esetén megváltozik. Az ilyen szabályoknál az aktuális elem alatti teljes részfa eltűnik, minden ott szereplő elem az aktuális közvetlen gyereke lesz. A többi szabály esetén a faszerkezet szülő-gyerek viszonyai nem változnak, csak az elemek sorrendjét rendezzük át.

4.2. A modell előállítás

Az új modell építéséhez be kell járnunk az általános AST-t és közben alkalmaznunk kell az ADL leírásból épített struktúrát. A régi AST bejárását egy bejáró osztály segítségével végezzük, ami képes többféle módon továbblépni az adott csúcsból. A többféle bejárési stratégiára az ADL-ben szereplő LINEAR kapcsoló speciális szerkezete miatt van szükség. Az ilyen csúcsoktól indítunk egy új balról jobbra szélességi bejárást [16] az alattuk levő részfa bejárására. A fa többi részét balról jobbra haladó szélességi bejárással járjuk be. Az építő algoritmus a bejáró osztály segítségével áll kapcsolatban a régi AST-vel, más módon nem látja azt. A bejárás során nem az általános AST csúcsait adjuk vissza, hanem egy interfész osztályt ami tartalmazza a szükséges lekérdezési műveleteket. Az építő algoritmus általánossága miatt van szükségünk az interfészekre, így ha később bővíteni akarjuk az új AST-t akkor csak az informá-

ciókat szolgáltató osztályokat kell átalakítanunk az algoritmus működéséhez. Az építő algoritmus két fő lépésből áll:

1. A szabályok alkalmazása már kész elemekre és a fa egy szintjének felépítése.
2. A szabályok törzsében szereplő elemek elkészítése és a szülő gyerek viszony megadása a helyes elhelyezéshez.

Az új modell építése a szabályok alkalmazásával kezdődik. Elkérjük az AST bejáró osztályunktól az első elemet szélességi bejárás szerint, elkészítjük belőle az új modellbeli alakját, majd elhelyezzük őt, mint gyökér elemet. Ezután megnézzük, hogy milyen kapcsolók tartoznak a szabályhoz, amit alkalmaztunk és átadjuk a már kész új AST-beli elemet a törzsét felépítő algoritmusnak. Ezen algoritmus egy listával tér vissza, ahol az elemek már helyes sorrendben szerepelnek a hozzájuk tartozó szülő hivatkozásával együtt. Az elemek listájának építése során ideiglenes változókat alkalmazunk a speciális pozíciókban elhelyezett elemekre (UFI, ...) és csak az építés végén helyezzük el őket. Figyelnünk kell továbbá a blokkok feldolgozására amik ismétlődhetnek. A blokkok belső szerkezete megegyezik a szabályok törzsével így alkalmazhatjuk rá ugyanazt az algoritmust, a szabályokhoz tartozó kapcsolók megadása nélkül. Az egyes elemek felépítése egy építő függvény segítségével történik, aminek a bemenő paraméterei az építendő elemhez tartozó lekérdező interfész és az adott elem ADL leírása, valamint egy logikai érték, ami igaz ha olyan elemet kell készítenünk ami nem szerepelt a fában. Ha olyan elemet adunk az építőnek, ami nem szerepelt az eredeti fában, úgy az első paraméter üresként is átadható. Az elemeket elkészítő algoritmusnak a következő információkat kell beállítania:

- Az elem pozíciója
- Az elem kirajzolható szövege
- Az elem típusa

A többi információt a szabályok törzsét építő algoritmus tölti ki a környezettől függően. A pozíciók kitöltésénél figyelnünk kell a direkt pozícióval megadott elemekre. Az ilyen elemeket nem tudjuk egyből elhelyezni az elem-listában, ezért külön tárolnunk kell őket. A tárolás egy módja ha asszociatív tömbben helyezzük el az ilyen elemeket a pozíciójukat használva kulcsként. A

szabály leírásának végeztével egy számlálós ciklussal egyszerűen kitölthetjük a visszatérési listát az asszociatív tömbünk segítségével. Fontos megemlítenünk, hogy a szülő kitöltése minden esetben egyszerű feladat, hiszen minden esetben az az elem lesz az amire a szabályt alkalmaztuk. A `LINEAR` kapcsolós elemek esetén a bejárás átváltásához előbb meg kell hívnunk a bejáró `initializeLinearIteration()` nevű függvényét, ami átáll az adott részfára. Paraméter értékre nincs szükségünk, mivel az átállás minden esetben az aktuális elem szülőjére történik. A bejárás során ügyelnünk kell, hogy minden elemet csak egyszer érintsünk. A feltételes szabályok törzsét az elemek listáját visszaadó algoritmusban választjuk ki. A feltételek szöveges formában történő tárolása esetén a feldolgozás menete a következő:

- Elkérjük a következő tokent a kifejezésből
- Változó esetén elhelyezzük egy ideiglenes változóban
- Művelet esetén a ideiglenes változókra és a következő tokenre alkalmazzuk azt és az eredményt a visszatérési változóban helyezzük el.
- Nyitó zárójel esetén rekurzívan meghívjuk a kiértékelő algoritmusunkat.
- Csukózárójel esetén visszatérünk az aktuális visszatérési értékkel.
- `not` token esetén rekurzívan meghívjuk a kiértékelőnket és negáljuk a visszatérési értéket majd megyünk tovább a feldolgozással
- `CHILD_TYPE` vagy `PARENT_TYPE` vagy `CHILD_NUM` kulcsszó esetén lekérdezzük az aktuális AST-beli elemtől ezeket az információkat és azzal dolgozunk tovább mint változóval.

A feltételes kifejezések lehetséges implementációja `C++` nyelven megtalálható a B függelékben. A függelékben lévő kód csak szintaktikusan helyes kifejezések esetén működik. Ezen algoritmusok segítségével elkészíthető a második AST, amit könnyen ki tudunk rajzolni egy megfelelő kirajzoló program segítségével.

4.3. A modell kirajzolása

Az új modellünk szerkezete már nagyon egyszerűen kirajzolható egy szintfolytonos bejárással, ahol az egyes csúcsok a pozíció alapján kerülnek kirajzolásra. Egy elem megjelenítési módját a típusa, valamint az alkalmazott speciális

kapcsolók határozzák meg. A kirajzolósi pozíciók kiszámítása a gyökér elemtől kezdődik, ami tetszőlegesen elhelyezhető, de a kódszerű megjelenítéshez a bal felső sarok koordinátáját javasoljuk. A pozíciókat listák segítségével a legegyszerűbb kiszámolni. A listákba sorrendben helyezzük el az egyes irányoknak megfelelő elemeket, majd a bal, jobb, fent, lent sorrendben számoljuk ki a kirajzolásbeli koordinátákat. A listákban szereplő elemek koordinátájának kiszámítása felcserélhető a kirajzoló függvényében. A modell továbbá könnyedén kirajzolható struktogram, vagy akár folyamatábra alakban is, az elemek típusa alapján. A kirajzolás már a probléma egy könnyebbik pontja és a módját, valamint a rajz komplexitását a kirajzoló könyvtár sokoldalúsága határozza meg.

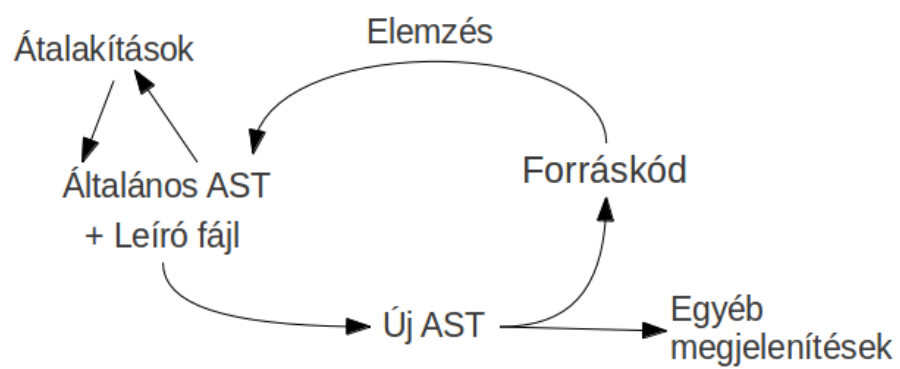
5. fejezet

Összefoglalás

A kapcsolódó eszközök vizsgálata során láthattuk, hogy egyre nagyobb az igény a különböző refaktoráló, statikus elemző és szoftvermegértést támogató eszközökre. Ezek az eszközök egy AST-n dolgoznak, majd a rajtuk végzett transzformációk után, ezt visszaalakítják forráskóddá. Azonban ezen eszközök vagy csak egy adott programozási nyelvre specializálódtak, vagy ha támogatnak több programozási nyelvet is, azt nagyon rugalmatlanul, sok megszorítás árán teszik.

A dolgozatomban erre a problémára készítettem egy módszert amely egy megoldást nyújt erre a problémára. Ez a módszer egy saját leíró nyelvre az ADL-re épül amely szintaxisfa struktúrájához illeszkedő módon írja le az elemek kiírásának módját, a kód visszagenerálása során.

A visszagenerálás során az ADL nyelvű leírófájl alapján sikerült a hiányzó elemek pótlását, valamint a forráskód szerkezeti rekonstrukcióját elérnem. Az általános AST-ből a leírt algoritmusok és struktúrák segítségével képes voltam egy új modellt előállítani, ami nem csak a forráskód teljes visszaalakítását tette lehetővé, de tetszőleges egyéb információ megjelenítését is támogatta a forráskód megjelenítése közben. A modell szerkezetéből adódóan a módszerhez később egyéb megjelenítés formákat is illeszthetünk, mint például struktogram-szerű megjelenítés, vagy folyamatábra, ezt jobban megfogalmazni vagy egyéb magasabb fokú grafikus megjelenítés. A dolgozat bevezetőjében levő 1.1. ábra az eredményeim alapján az 5.1 ábrán látható módon rajzolható újra.



5.1. ábra. Áttekintés

Irodalomjegyzék

- [1] F. Nielson, H. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 2004.
- [2] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, T. M., and M. Tóth, „Refactorerl - source code analysis and refactoring in erlang,” in *Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, (Tallin, Estonia), pp. 138–148, October 2011.
- [3] A. Salomaa, *Formal languages*. Academic Press Professional, Inc., 1987.
- [4] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
- [5] I. Chiswell and W. Hodges, *Mathematical logic*, vol. 3. OUP Oxford, 2007.
- [6] „Clang.” <http://clang.llvm.org/>.
- [7] „LLVM.” <http://llvm.org/>.
- [8] „GCC, the GNU Compiler Collection.” <http://gcc.gnu.org/>.
- [9] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.
- [10] „PMD.” <http://pmd.sourceforge.net/>.
- [11] R. Karp and M. Rabin, „Efficient randomized pattern-matching algorithms,” *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.
- [12] V. Kodaganallur, „Incorporating language processing into java applications: A javaCC tutorial,” *Software, IEEE*, vol. 21, no. 4, pp. 70–77, 2004.

- [13] F. Cesarini and S. Thompson, *Erlang Programming*. O'Reilly Media, 2009.
- [14] „DMS.” <http://www.semdesigns.com/Products/DMS/DMSToolkit.html>.
- [15] L. Garshol, „BNF and EBNF: What are they and how do they work.” <http://www.garshol.priv.no/download/text/bnf.html>, 2003.
- [16] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2001.

A. függelék

A leíró nyelv EBNF leírása

```
<startItems> ::= a-Z | _
<num> ::= { ? 0-9 ? } { ? 0-9 ? } *
<notNullNum> ::= { ? 1-9 ? } { ? 0-9 ? } *
<name> ::= <startItems> { <startItem> | <num> | } *
<string> ::= ' ' ?tetszőleges karakterek sorozata? ' '

<symbol> ::= "%symbol" <name> <string>
<ruleDecl> ::= "%rule" <name> <name> { ",", <name> } *

<switch> ::= "V\_DIRECTPOS" | "H\_DIRECTPOS" |
            "LINEAR" | "SKIP" | "CONTAINER"

<direction> ::= "l" | "r" | "d" | "u"
<specialDirection> ::= "LLA" | "LFI" | "RLA" | "RFI" |
                       "DLA" | "DFI" | "ULA" | "UFI"
<loopDirection> ::= "normal" | "reverse"
<itemFirstPart> ::= <direction> | <specialDirection> |
                   <notNullNum> | "skip"
<itemSecondPart> ::= "." (<name> | <string> | "ITEM")
<itemThirdPart> ::= [ "[" (<notNullNum> | "*" )
                     [ "|" <loopDirection> ] "]" ]

<item> ::= <block> | <itemFirstPart>
          [ "." <itemSecondPart> ] [ <itemThirdPart> ]
<block> ::= "{" { <item> } * "}" [ <itemThirdPart> ]
```

```

<specialCondItem> ::= "PARENT\_TYPE" |
                      "CHILD\_TYPE" | "CHILD\_COUNT"
<operatorSide> ::= <name> | <specialCondItem> | <num>

<equality> ::= <operatorSide> "=" <operatorSide>
<notequality> ::= <operatorSide> "!=" <operatorSide>
<and> ::= <condition> "and" <condition>
<or> ::= <condition> "or" <condition>
<not> ::= "not" <condition>
<condition> ::= "(" <condition> ")"
               { <condition> }* | <and> | <or> |
               <not> | <equality> | <notequality>
<cond> ::= "COND" "(" <condition> ")"

<simpleBody> ::= { <item> }*
<complexBody> ::= <cond> <simpleBody> "|"
                 <simpleBody> {complexBody}*

<ruleBody> ::= <simpleBody> | <complexBody>

<ruleDef> ::= <name> [ <switch> [ { "|" <switch> }* ] ]
               ":" <ruleBody> ";"

```

B. függelék

Logikai kifejezés feldolgozó

```
0 bool logicExprEvaluate(const string& expr_,const string&
  childType_,const string& parentType_,int childNum_,int&
  startInd_)
1 {
2   bool result=true;
3   string variable1;
4   for(;startInd_<expr_.size() && startInd_!=string::npos ;++
      startInd_)
5   {
6     string keyword=getNextKeyword(expr_,startInd_);
7     if(keyword==Not)
8     {
9       result=!logicExprFelolddo(expr_,childType_,parentType_,
        startInd_);
10      break;
11    }
12    else if(keyword==Child_Type)
13    {
14      variable1=childType_;
15    }
16    else if(keyword==Parent_Type)
17    {
18      variable1=parentType_;
19    }
20    else if(keyword==Child_Num)
21    {
22      variable1=itoa(childNum_,10);
23    }
24    else if(keyword=="(")
25    {
```

```

26         result=logicaExprFeloldo(expr_,childType_,parentType_,++
27             startInd_);
28     }
29     else if(keyword=="")
30     {
31         break;
32     }
33     else if(keyword==Equel || keyword==NotEquel)
34     {
35         string nextKeyword=getNextKeyword(expr_,++startInd_);
36         string variable2=nextKeyword;
37         if(nextKeyword==Parent_Type)
38         {
39             variable2=parentType_;
40         }
41         else if(nextKeyword==Child_Type)
42         {
43             variable2=childType_;
44         }
45         if(keyword==Equel)
46         {
47             if(variable1==variable2)
48             {
49                 result=true;
50             }
51             else
52             {
53                 result=false;
54             }
55         }
56         else//keyword==NotEquel
57         {
58             if(variable1!=variable2)
59             {
60                 result=true;
61             }
62             else
63             {
64                 result=false;
65             }
66         }
67     }

```

```

68     else if(keyword==More || keyword==Less || keyword==LessEqual
        || keyword==MoreEqual)
69     {
70         string nextKeyword=getNextKeyword(expr_,++startInd_);
71         int variable2;
72         if(nextKeyword==Child_Num)
73         {
74             variable2=childNum_;
75         }
76         else
77         {
78             variable2=atoi(nextKeyword.c_str());
79         }
80
81         if(keyword==Less)
82         {
83             if(atoi(variable1.c_str()) < variable2)
84             {
85                 result=true;
86             }
87             else
88             {
89                 result=false;
90             }
91         }
92         else if(keyword==More)
93         {
94             if(atoi(variable1.c_str()) > variable2)
95             {
96                 result=true;
97             }
98             else
99             {
100                 result=false;
101             }
102         }
103         else if(keyword==LessEqual)
104         {
105             if(atoi(variable1.c_str()) <= variable2)
106             {
107                 result=true;
108             }
109             else

```

```

110         {
111             result=false;
112         }
113     }
114     else if(keyword== MoreEqual)
115     {
116         if(atoi(variable1.c_str()) >= variable2)
117         {
118             result=true;
119         }
120         else
121         {
122             result=false;
123         }
124     }
125 }
126 else if(keyword==And)
127 {
128     result=result & logicExprFeloldo(expr_, childType_,
129                                     parentType_, ++startInd_);
129 }
130 else if(keyword==Or)
131 {
132     result=result | logicExprFeloldo(expr_, childType_,
133                                     parentType_, ++startInd_);
133 }
134 }
135
136 if(startInd_>=expr_.size()-1 || startInd_==string::npos)
137 {
138     _startIndex=0;
139 }
140
141 return result;
142 }
143
144 string getNextKeyword(const string& expr_, int & startInd_)
145 {
146     string keyword;
147     int length=0;
148     if(startInd_!=string::npos && startInd_<expr_.size())
149     {
150         int endPos=expr_.find(CustomLogicSep, startInd_);

```

```
151     if (endPos!=string::npos)
152     {
153         length=endPos-startInd_;
154         keyword=expr_.substr(startInd_, length);
155     }
156     startInd_+=length;
157 }
158
159 return keyword;
160 }
```