Követelmények

A dolgozat keretein belül egy DSL nyelvnek kell elkészülnie és egy hozzá tartozó kódgenerátornak. A DSL-ben meg lehessen adni karakterkódolás-párokat, illetve, hogy melyik bájtsorozatot melyik másik bájtsorozattal kell helyettesíteni, hogy a két kódolás között konverziót lehessen végezni. A DSL-hez készüljön modell, ami az Xtext Eclipse plug-inben legyen létrehozva, és egy tetszőleges kiegészítő funkciót is meg kell valósítania.

A kódgenerálás a DSL nyelvben megírt programokat dolgozza fel és azokból készítsen használható C forráskódot. A generált kódnak csupán a DSL-ben leírt információkat, vagyis az egyes bájtkonverziókat kell tárolnia, nem egy működő, teljes értékű alkalmazást kell generálni. Azonban készülnie kell egy C nyelvű keretprogramnak is, mely a generált kódot használja fel, és a kettő együtt az iconv The Open Groupe szabványt valósítják meg, mely egy függvénycsoportot definiál karakterkódolások közötti konverziókhoz.

Tervezés

A DSL nyelvben alapvetően több karakterkódolás közötti konverziót is meg lehet adni struktúrált formában: megadjuk mi a kiinduló (source) kódolásunk, mi a cél (target) kódolásunk, majd leírjuk az egyes kódpontokat. Az egyszerűség érdekében a kódpontokat nem számmal, hanem hexadecimális kóddal írjuk le.

Itt látható egy kezdeti vázlat a DSL nyelv szintaxisához, melyben még nem szerepel a célkódolás megjelölése:

source someEncodingName {

alias anAliasName

// illegalChar 0xffee

// this is a comment

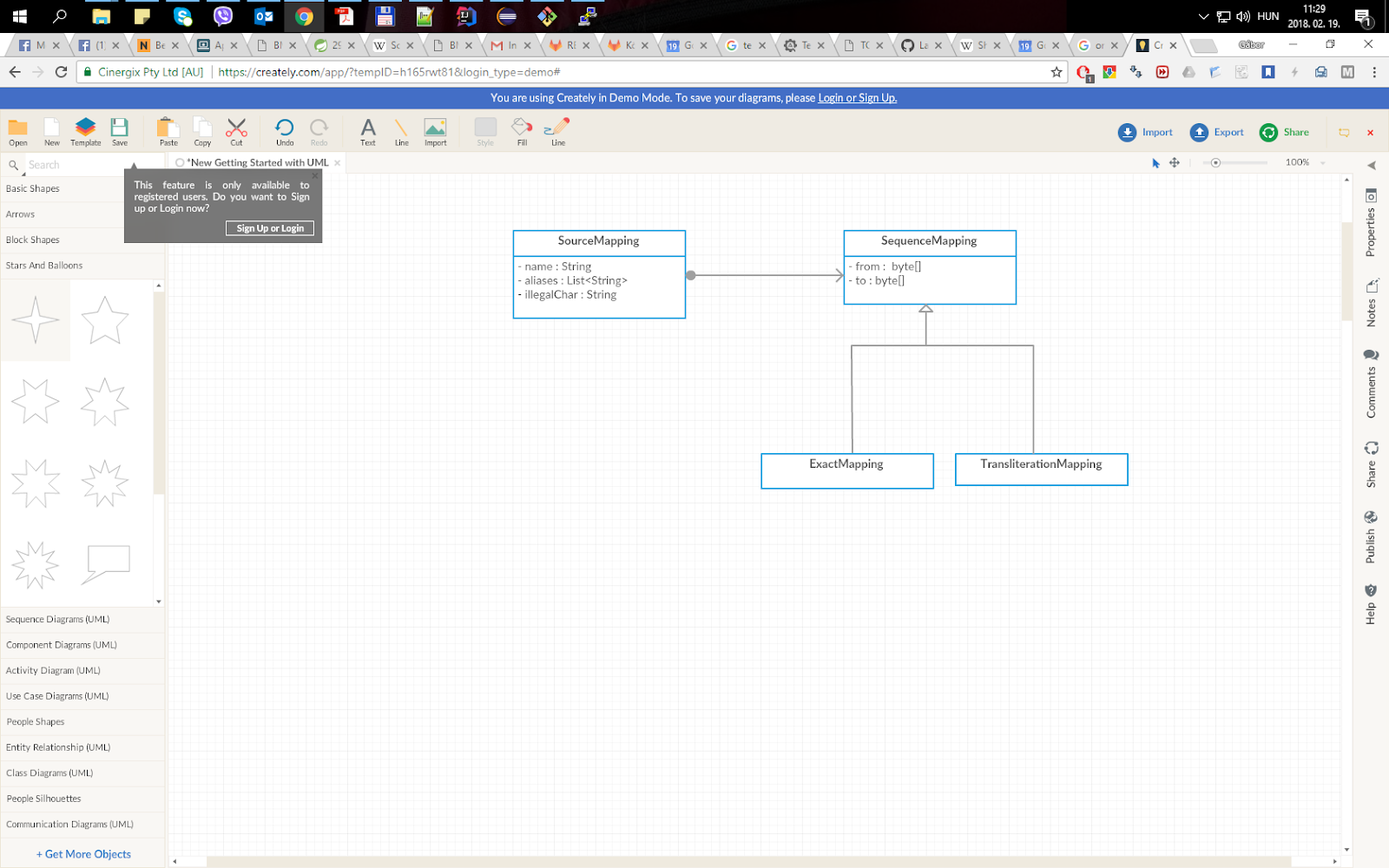
0x0001 = 0xff01 // this will be an exact mapping

0x0002 ~ 0xff02 // this will be a transliteration mapping

// ...

}

A DSL nyelvnek, mint metamodellnek itt látható egy kezdeti diagramja:



A generált C kód függvényeket tartalmazna, amellyel a konverziók adatait lehet lekérdezni. A függények nagy mértékben támaszkodnak Kövesdán Gábor hashtáblák C nyelvű implementációjára.[[1]](#footnote-1) Az egyes függvények hashtáblákat adnának vissza, melyben elérhetőek az aliasok, a nyelvekben használt kódegységeket, és a mappingok.

Fejlesztés

A nyelvtan elkészítése:

A tervezés után a saját DSL nyelvemet kellett elkészítenem. Ennek a műveletnek a lényege, hogy Xtext-ben objektumokat/metamodell elemeket hozok létre, melyek megfeleltethetőek a tervezett nyelv egy-egy részének. Egy új Xtext projekt létrehozása után az EncodingLang.xtext fájlban írtam le a DSL-em nyelvtanát.

Mint minden Xtextes DSL-ben, így a sajátomban is először egy olyan objektumot kellett létrehoznom, amelyben a nyelv többi elemét tudjuk tárolni. Ez az objektum hagyományosan a Modell nevet kapja. Látható, hogy a Model objektum egy elements nevű listában tárolja a SourceMapping nevű objektumokat, a csillag jelzi, hogy legalább egy darabot.

grammar org.xtext.example.EncodingLang with org.eclipse.xtext.common.Terminals

generate encodingLang "http://www.xtext.org/example/EncodingLang"

Model:

(elements+=SourceMapping)\*;

A SourceMapping objektum megadja, hogy egy karakterátkódolásnál, mely kódolásból indulunk ki, annak milyen más elnevezései vannak, és hogyan kell konvertálni egy tetszőleges célkódolásba. A DSL-ben egy SourceMapping írásakor a source kulcsszót kell írni előbb, melyet az idézőjelek jeleznek. Majd egy ID-t ami, a SourceMapping name attribútumába lesz tárolva. Az ID egy ún. terminál (lásd később). Ezután felsorolhatjuk az Aliasokat, és a Conversionöket, a Modellnél látott leírással.

SourceMapping:

"source" name=ID "{"

(aliases += Alias)\*

(conversions += Conversion)\*

"}"

;

Egy Conversionben megadható, hogy mi lesz a kívánt célkódolás, és adott kiinduló kódolás esetén, az egyes kódpontok, hogy változnak meg. A name feltüntetése után írhatjuk le a konkrét konverziókat, melyek objektumainak neve Mappings és a mappings nevű listában tárolódnak el.

Conversion:

"target" name=ID "{"

(mappings += Mapping)\*

"}"

;

Egy Mappings egy konkrét kódpontokat párosít össze, megadva, hogy a kiinduló karakterkódolásban egy kódpont célkódolásban melyik kódpontnak felel meg. Kétféle Mappingot különböztetek meg. Az egyik az ExactMapping, melynek írásakor = karakter, a másik a TransliterationMapping, mely esetben ~ karakter használatos. A két Mapping közötti különbség a jelentésben van: az ExactMapping 1-1 kapcsolatban álló kódpontokat kapcsol össze, míg a TransliteraionMapping a problémásabb párosításoknál használatos. Például ha egy karakternek nincs benne a célkódolásban, így azt egy hozzá hasonló karakterhez kell párosítani. Programozási szempontból nem kezelem másként a két Mappingot, azonban a későbbi bővítés érdekében mégis két objektumot hoztam létre. A kódban lévő INPUTCHAR, az ID-hoz hasonló terminál (lásd később).

Mapping:

ExactMapping | TransliterationMapping;

ExactMapping:

from=INPUTCHAR "=" to=INPUTCHAR;

TransliterationMapping:

from=INPUTCHAR "~" to=INPUTCHAR;

A SourceMapping egyik attribútuma az Aliasok listája. Egy Alias objektum a nyelvben egy alias kulcsszóból és egy ID terminálból áll.

Alias:

"alias" name=ID;

Egy nyelvtan készítésekor terminálok használata legtöbbször elengedhetetlen. A DSL nyelvünkben megírt program szöveges inputot jelent egy parser számára. A parser a kódban szereplő karakterláncoknak és sztringeknek próbál valamilyen logikai jelentést adni. A fenti objektumoknál ez a feladat nem volt nehéz a kulcsszavak miatt. De egyéb sztringeket már nem lehet objektumokhoz rendeli, mert annyira függetlenek tőlük. Ilyen például egy komment a programban. Ezeket a karakterláncokat termináloknak nevezzük, és ezeket a terminálokat lehet hozzárendelni az objektumok attribútumaihoz. Egy terminál implementálásakor meg kell határozni, hogy mely konkrét karakterek tartoznak hozzá. Mivel a legtöbb terminál igen gyakori, az Xtextben ezek már implementálva vannak, például az ID, az INT, a STRING, a DOUBLE, SL\_COMMENT (single line comment), ML\_COMMENT (multi-line comment) vagy a WS (white space). Ezen beépített Xtext terminálok automatikusan importálódnak a nyelvtanba. Kis gyakorlást igényel, hogy egy fejlesztő meg tudja határozni egy nyelv készítésekor, hogy mely nyelvi elemekhez kell a fentiekhez hasonló objektumot vagy terminált létrehoznia. Én úgy tudtam értelmezni, hogy az objektumok információkat tárolnak változókban és listákban, a terminálok pedig maguk az információk: egy integer vagy egy sztring.

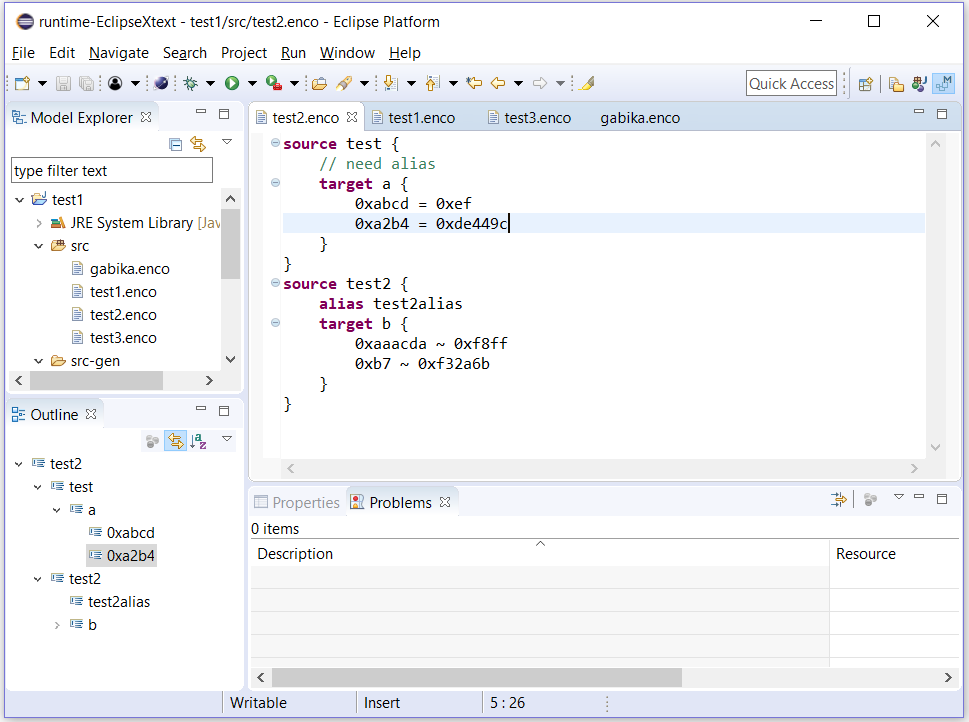
Az én DSL nyelvemben egy új terminált is létre kellett hoznom, melynek neve INPUTCHAR lett. A terminál segítségével a karakterkódolásokban használt kódpontokat lehet hexadecimális formában leírni. Egy ilyen karakterlánc egy nullás karakterrel kezdődik, x-szel folytatódik, majd utána páros számú hexadecimális számrendszerbeli karakter következik, legalább két darab, kis és nagybetű között nem teszünk különbséget. Például az ASCII hetvenötös kódpontú ’K’ karaktere 0x4B, vagy akár 0X4b formában is leírható.

terminal INPUTCHAR:

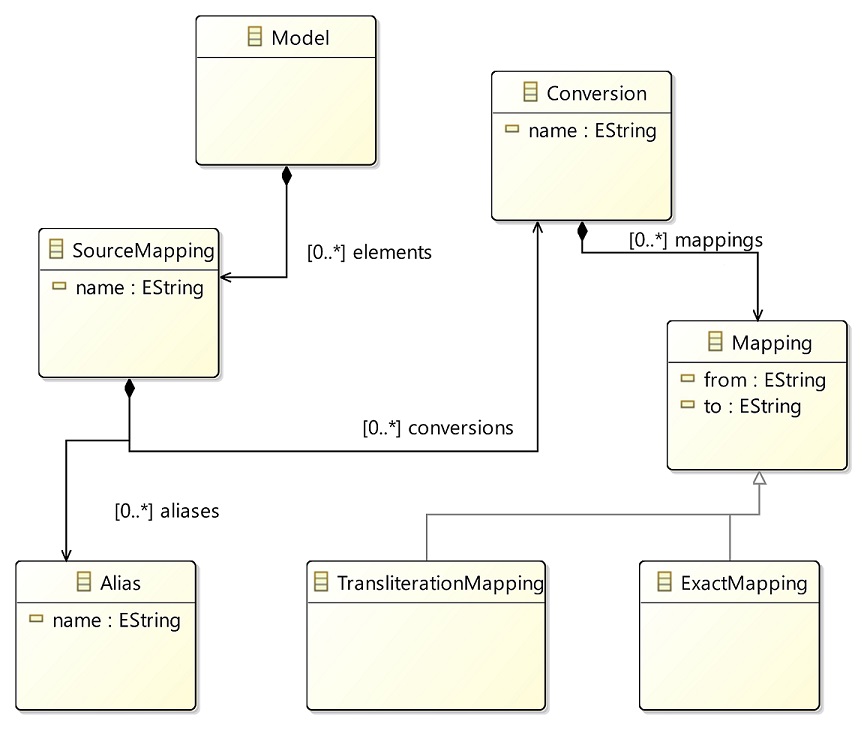
'0'('x'|'X')(('0'..'9'|'a'..'f'|'A'..'F')('0'..'9'|'a'..'f'|'A'..'F'))+

;

Ezzel elkészült a DSL nyelv nyelvtana. Az Xtext a nyelvtanból képes egy Eclipse plug-int generláni, melyet futtatva lehetőségünk van a saját DSL nyelvünkben kódokat írni. Az Xtext egyéb funkciókat is biztosít a plug-inhoz, mely funkciók egy átlagos programozási nyelvtől vagy programozási nyelv szerkesztőtől elvárható, például a szintaxis kiemelése vagy a forráskód outlineja. A nyelv neve EncodingLang lett, a forráskódok fájlok kiterjesztése pedig .enco.

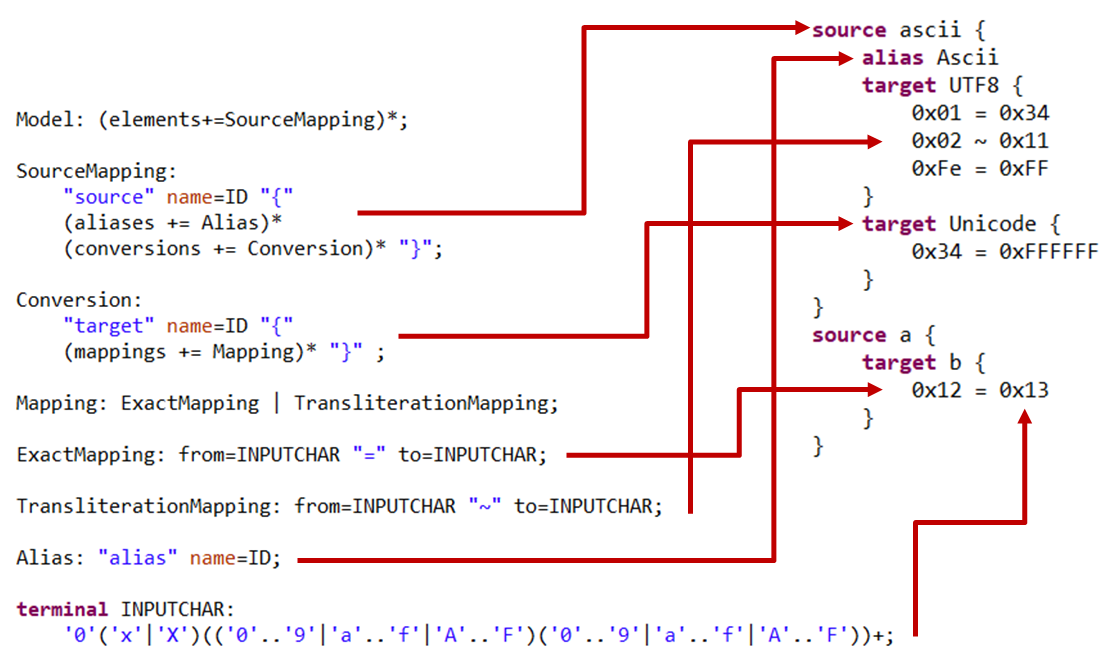


A nyelvtan elkészítéséhez használt programnyelv az Xtext saját nyelve.[[2]](#footnote-2) Ebből a nyelvtani leírásból az Xtext képes EMF modellt generálni, és sok jelentős Java osztályt készíteni. Az EMF-beli Ecore modellhez volt lehetőségem egy nézetet készíteni, melyben grafikusan látható a modell diagramja.



A diagram tartalmazza az összes implementált nyelvtani objektumot, és a hozzájuk tartozó attribútumokat. Ezek már ténylegesen elérhető és használható Java osztályok. Fontos megjegyezni a Mapping osztály. Itt ugyanis nem volt deklarálva semmilyen attribútum, de mivel egy Mapping objektum vagy ExactMapping vagy TransliterationMapping lehet, és mivel mindkét leszármazott tartalmazta és from és to attribútumokat, ezek az attribútumok ki lettek emelve az ősosztályba. Ezt, és a többi tervezői döntést az Xtext önállóan, emberi beavatkozás nélkül hozta meg.

Az Xtext objektumok és a nyelvtani elemek a következőképp állíthatóak kapcsolatba:



A nyelv tesztelése

A nyelv elkészítésével az Xtext lehetőséget biztosít a nyelv tesztelésére Java osztályokon keresztül. Kétféle szempont szerint lehet tesztelni. Az első a szintaxis ellenőrzése, vagyis, hogy a parser egy megírt programban az elvárásoknak megfelelően ismeri fel az egyes nyelvtani elemeket. Egy ilyen teszt teszteseteinek működése abból áll, hogy a parser bemeneteként megadunk egy helyes vagy helytelen szintaxissal rendelkező programkódot, és megvizsgáljuk, hogy a parser elfogadja vagy visszautasítja a kódot, és ebből következtetünk a parser megfelelő működésére.

A következőkben néhány tesztesetet szeretnék bemutatni példaként. Ezek a tesztek Xtend nyelven íródtak meg, ezért idegennek tűnhetnek, de értelmezésük nem okozhat problémát. Az Xtend nyelvet a kódgenerátor implementálásakor fogom bővebben ismertetni.

Az első teszteset a legelső tesztesetem, melyben csupán egyetlen source objektumot hozok létre. A parseHelper a parser egy példánya, ennek a parse() metódusa dolgozza fel a forráskódot. Az Assert osztály segítségével megvizsgálhatjuk a függvény visszatérési értékét és forráskódban lévő hibákat.

@Test

**def** **void** simpleSource() {

**val** result = parseHelper.parse('''

source test{}

''')

Assert.*assertNotNull*(result)

Assert.*assertTrue*(result.eResource.errors.isEmpty)

}

Ebben a tesztesetben a kommentezés terminálját vizsgáltam, és annak integritását a nyelvtanommal. A teszt elve ugyanolyan, mint az előző tesztesetnél.

@Test

**def** **void** comment() {

**val** result = parseHelper.parse('''

source test{ //comment1

alias testAlias1 //comment2

/\*commentline1

commentline2\*/

alias TestAlias2

}

''')

Assert.*assertNotNull*(result)

Assert.*assertTrue*(result.eResource.errors.isEmpty)

}

A következő teszteset szándékosan egy hibás forráskóddal használja a parsert, és az elvárt eredmény is a parser hibajelzése.

@Test

**def** **void** mixed\_2Bytes\_NumOnly() {

**val** result = parseHelper.parse('''

source test{

alias TestAlias1

0x1468=0x3487

alias TestAlias2

0x3537~0x5367

0x8648~0x2946

}

''')

Assert.*assertNotNull*(result)

Assert.*assertFalse*(result.eResource.errors.isEmpty)

}

Az utolsó teszteset, amit be szeretnék mutatni, egy olyan teszteset, melyben a INPUTCHAR terminálomat vizsgálom meg.

@Test

**def** **void** multipleMapping\_2Bytes() {

**val** result = parseHelper.parse('''

source test {

target a {

0xabcd=0xefef

0xa2b4=0xde9c

0xaaaa~0xffff

0xb78c~0xf32a

}

}

''')

Assert.*assertNotNull*(result)

Assert.*assertTrue*(result.eResource.errors.isEmpty)

}

A nyelvtan és a hozzá tartozó szintaxis tesztek implementálása alatt a Test Driven Development (TDD) fejlesztési módszert próbáltam ki. Ennek lényege három különböző lépés, vagyis szaksz ismételt használata. Először teszteket készítek a program egy minimális funkciójáról, ez a fejlesztés piros szakasza. A következő a zöld szakasz, amikor ezt a funkciót implementálom, hogy működőképes legyen és átmenjen a teszten. Végül a kék szakasz jön, mikor is a funkciót refaktorálom, hogy a forráskód szebb és praktikusabb legyen, miközben a teszteket továbbra is futtatjuk. Ezt a három szakaszt az implementálás befejeztéig folyamatosan iterálom.[[3]](#footnote-3)

Számomra hasznosnak bizonyult ez a fejlesztési módszer, ugyanis adott pillanatban céltudatosan tudtam, hogy a nyelv melyik részét szeretném elkészíteni. Továbbá, a fejlesztés végeztével, a nyelvtan elkészültével jelentős mennyiségű teszteset állt rendelkezésemre. A fent bemutatott tesztesetek a TDD különböző iterációiban készültek el: például a legelső teszt a fejlesztés legelején, az utolsó pedig a legvégén.

A másik nézőpont, mely alapján lehetőségünk van az elkészített Xtext nyelvtant tesztelni, az Xtext által generált modellt ellenőrzi. Ha DSL nyelvtant metamodellként kezeljük, akkor a nyelvben megírt programok a metamodell példányosításának felel meg. Ez a modell a nyelvtanban definiált objektumokból épült fel, és az Xtext generálja az programkód feldolgozásával. Azonban ellenőrizni kell, hogy a modellben lévő objektumok tulajdonságai és kapcsolatai megfelelnek a programozó szándékának.

Ennek a tesztnek az alapgondolata, hogy egy használható, teljes értékű, hibamentes programkódot megírunk, átadjuk az Xtext-nek, hogy feldolgozza, majd a generált modellt több szempontból megvizsgáljuk. A vizsgálatban az objektumok attribútumait, listáit, listáinak méreteit, stb... kérjük le, és összevetjük a program írójának igényeivel, és remélhetőleg a modell megfelel az elvárásoknak, vagyis a nyelvtan is helyes.

Ez a teszt is Xtend nyelvben készült és a parser egy példányát használja. A programkódra meghívjuk a parse() metódust, ami visszatér egy modell változóval. Majd az Assert osztály segítségével ellenőrizhetjük a generált modellt.

@Test

**def** modelTest(){

**val** model = '''

source ascii {

alias Ascii

target UTF8 {

0x01 = 0x34

0x02 ~ 0x11

0xFe = 0xFF

}

target Unicode {

}

}

source a {

target b {

0x12 = 0x13

}

}

source c{}

'''.parse

Assert.*assertNotNull*(model);

Assert.*assertTrue*(model.eResource.errors.isEmpty)

Assert.*assertEquals*(model.elements.size, 3);

// …

Assert.*assertEquals*(model.elements.get(1).name, "a")

Assert.*assertEquals*(model.elements.get(1).conversions.size, 1)

Assert.*assertEquals*(model.elements.get(1).conversions.get(0).name, "b")

Assert.*assertEquals*(model.elements.get(1).conversions.get(0).mappings.size, 1)

Assert.*assertEquals*(model.elements.get(1).conversions.get(0).mappings.get(0).from, "0x12")

Assert.*assertEquals*(model.elements.get(1).conversions.get(0).mappings.get(0).to, "0x13")

Assert.*assertEquals*(model.elements.get(2).name, "c")

Assert.*assertEquals*(model.elements.get(2).conversions.size, 0)

}

Ezzel a kétféle teszttel biztosra mehetünk, hogy a parser meg tudja különböztetni a szintaktikailag helyes és helytelen programkódokat, illetve, hogy a megfelelő modellt készíti el adott forráskód feldolgozásakor. Ha szükséges, további teszteket is lehet készíteni a modell egyéb kritériumaihoz, például függőségek esetén megvizsgálható, hogy nincs-e objektumok között körkörös függőség.

Egyéb programozási funkció

Szakdolgozatom részeként meg akartam ismerni azon kiegészítő funkciókat, amelyekkel bővíteni tudom a DSL nyelvemet vagy a nyelv szerkesztőjét. Ezen funkciók megkönnyítik a nyelvben való programozást, de egyéb előnyeik is lehetnek. Ilyen funkció például a content assist, scoping, sorosítás, quick fixes vagy az Xtext által már valamilyen mértékben kidolgozott szintaxiskiemelés. Ezen kiegészítések implementálásához az Xtext Java osztályokat szolgáltat és a plug-inbe való integrációját is megoldja.

Számomra a nyelv formázása tűnt a legérdekesebbnek. A formázás egységesíti a forráskódban lévő whitespaceek használatát, javítja a forráskód elrendezését és olvashatóbbá, átláthatóbbá teszi a kódot. Sőt, akár a programozási hibák számát is csökkentheti, azáltal, hogy az olvashatóbb kódban könnyebben tudjuk a hibákat felfedezni. Továbbá az egységesített kódok miatt, több kódban lévő eltéréseket könnyebben meg lehet találni (git diff). A szerkesztőfelületek általában segítenek, hogy automatikusan lehessen formázni a programot, például billentyűkombinációval. Azért választottam a formázás funkcióját, mert nem tudtam elképzelni, mennyire egyszerű vagy monoton feladat egy nyelv kinézetét meghatározni, de szerencsére nem volt hosszú, unalmas tevékenység.

Az Xtext ehhez a funkcióhoz is generált Java osztályt, melyben Xtend nyelven kellett programozni. A formázás implementálásakor a nyelvben található ún. kulcsszavakból kellett kiindulnom. Ezek a kulcsszavak a nyelvtanban található sztringek (nem a STRING terminál és nem terminálok), melyeket program írása során bele kell írni a kódba. Az én DSL nyelvemben nincs túl sok kulcsszó: source, target, alias, {, }, =, ~. Formázás során bemenetként megkapom a program modelljét, vagyis a DSL nyelvtan példányosítását, és az ebben található objektumokhoz vagy objektumok kulcsszavaihoz rendelhetek whitespaceket. Többnyire az elemek elé vagy után tuduk beszúrni whitespacet, de például beállítható, hogy egy nyitó és záró kapcsos zárójel közötti sorok egy tabulátornyi hellyel bentebb kezdődjenek. Formázás során végig kell iterálni az összes objektumon és beállítani az egyes whitespaceket.

Az első formázó függvényt az Xtext generálta, bár a függvény üres volt. A függvény paraméterként megkapja az éppen formázni kívánt elemet, jelen esetben a Modell objektumot. Mivel a Modell csak tárolja a nyelv többi elemét, lekérjük tőle a tárolt SourceMappingokat, és mindre meghívjuk a format függvényt. Az Xtend nyelv sajátossága, hogy itt nem a sourceMapping format függvényét hívjuk meg, hanem a format függvényt, melynek egyik paramétere a sourceMapping objektum.

**def** **dispatch** **void** format(Model model, **extension** IFormattableDocument document) {

**for** (SourceMapping sourceMapping : model.getElements()) {

sourceMapping.format;

}

}

A következő függvény a SourceMapping formázását végzi el. A használt append függvények az elem után helyez ez whitespace-t, az első sorban például a source kulcsszó után egy szóközt helyez el. Míg a prepend függvény az elem elé helyez whitespacet. Az interior függvény a kapcsoz zárójelek között sorokat határozza meg, jelen esetben azt, hogy minden sor egy tabulátorral kezdődjön. Ezután az Aliasokra és a Conversionökre hívjuk meg a format függvényt.

**def** **dispatch** **void** format(SourceMapping s, **extension** IFormattableDocument document) {

s.regionFor.keyword("source").append[oneSpace]

**var** open = s.regionFor.keyword("{")

open.prepend[oneSpace].append[newLine]

**var** close = s.regionFor.keyword("}")

close.append[newLine]

interior(open, close)[indent]

**for** (Alias alias : s.getAliases()) {

alias.format

}

**for** (Conversion conversion : s.getConversions()) {

conversion.format

}

}

Az Aliasok formázása nem bonyolult. Minden Alias után új sor következik, illetve az alias kulcsszó előtt egy tabulátor áll, utána pedig egy szóközt helyez el.

**def** **dispatch** **void** format(Alias a, **extension** IFormattableDocument document) {

a.append[newLine]

a.regionFor.keyword("alias").prepend[indent].append[oneSpace]

}

Egy Conversion formázása megegyezik a SourceMappingéval. Mivel a Conversion tárolja a hozzá tartozó Mappingokat, ezért a Mappingok lisáját végig kell iterálni, és mindegyikre meghívni az illeszkedő format függvényt.

**def** **dispatch** **void** format(Conversion c, **extension** IFormattableDocument document) {

c.regionFor.keyword("target").prepend[newLine].append[oneSpace]

**var** convopen = c.regionFor.keyword("{")

convopen.prepend[oneSpace].append[newLine]

**var** convclose = c.regionFor.keyword("}")

convclose.append[newLine]

interior(convopen, convclose)[indent]

**for** (Mapping mapping : c.getMappings()) {

mapping.format

}

}

A Mappingok formázásánál gondom akadt, ugyanis egy adott Mapping esetén nem tudtam, hogy melyik kulcsszóhoz tudok whitespacet illeszteni. Le kellett volna valahogy kérdeznem a Mappingtól, hogy melyik fajta Mapping, hogy eldönthessem, melyik kulcsszót kell használnom. Azonban tudtam, hogy ez objektum orientált programozás szempontjából kerülendő megoldás, így ezt elvetettem. Kutatás után a megoldás egyszerűbb lett, mint gondoltam. Amikor egy kulcsszót szeretnénk megtalálni az objektum programkódbeli sorai között, nem generálódik hibajelzés, ha olyan kulcsszót keresünk, amely nincs a kódban. Így meg tudtam adni mindkét féle kulcsszónak a környezetét, és amelyik szerepel a forráskódban, az fog érvényesülni.

**def** **dispatch** **void** format(Mapping m, **extension** IFormattableDocument document) {

m.append[newLine]

m.regionFor.keyword("=").surround[oneSpace]

m.regionFor.keyword("~").surround[oneSpace]

}

A formázás elkészülte és kipróbálása után egy tesztet is írtam hozzá, természetesen egy Xtext által készített osztályban. A formázás tesztelése elég egyértelmű feladat. Össze kellett hasonlítanom egy automatikusan és egy kézzel formázott forráskódot.

A teszt végrehajtásához a FormatterTestHelper osztályt biztosította az Xtext.

@Inject **extension** FormatterTestHelper formatterTester

Az összehasonlítást a FormatterTestHelper asserFormatter metódusa végzi, nekünk csak a bemeneti forráskódokat kell előállítanunk. Először egy formázatlan kódot adtunk meg, erre automatikusan meg lesz hívva a formázó függvény.

@Test

**def** **void** formatting() {

assertFormatted[

toBeFormatted = '''

source test {

// need alias

target a{

0xabcd = 0xef

0xa2b4 = 0xde449c

}

}

source test2 {

alias test2alias target b {

0xaaacda ~ 0xf8ff

0xb7 ~ 0xf32a6b

}}

'''

Másodszor, a formázás elvárt végeredményét kellett megadni. A formázást helyesen készítettem el, ez lett a végeredmény és a teszt is sikeres volt.

expectation = '''

source test {

// need alias

target a {

0xabcd = 0xef

0xa2b4 = 0xde449c

}

}

source test2 {

alias test2alias

target b {

0xaaacda ~ 0xf8ff

0xb7 ~ 0xf32a6b

}

}

'''

]

}

Kódgenerátor

A nyelvtan elkészültével az Xtext generál egy Eclipse plug-int, melyben a saját nyelvünkben lehet programozni. Az Xtext továbbá egy parsert is készít, amely a DSL nyelvben megírt programokat dolgozza fel, és Java osztályokat hoz létre a DSL program modelljének megfelelően. A Java osztályokat tetszőlegesen fel lehet dolgozni, például statisztikák készítése, de a kódgenerálás is egy lehetőség.

Az Xtext által készült osztályok egyike a kódgenerátor. A generált osztályban eleinte csak minta függvényeket készít az Xtext. Megmutatja, hogyan lehet fájlokat létrehozni, hogyan lehet Hello World programot kiírni a fájlba. Ez is olyan osztály, amely Xtend nyelven íródott.

Az Xtend nyelv a kódgenerálásnál mutatja meg erősségeit. Kiemelendő a multiline sztring használata, mivel sokkal átláthatóbbá teszi a forróskódot, amelyet generálunk. Továbbá lehetőség van kilépni a sztringből, úgy hogy azt ne szakítsa meg. Erre a « és a » karakterek adnak lehetőséget, amelyek nincsenek benne a magyar billentyűzetekben. Két ilyen karakter között számításokat végezhetünk és függvényeket hívhatunk meg, és ezek eredményei kerülnek bele a multiline sztringbe.

Az osztály legfőbb függvénye a doGenerate függvény, ez lesz meghívva az Xtext által. Bemeneti változói egy Resource objektum, mely a lekódolt modellt tartalmazza, egy IFileSystemAccess2, mely a fájlrendszerben képes módosításokat végezni, és egy IGeneratorContext, melyet használatára nem votl szükségem. A fsa objektum generateFile metódusával lehet a kódgenerálást elindítani. Megadjuk, hogy melyik fájlba szeretnénk kódot írni, majd azt a metódust, amely generálja a kód sztringjét. Látható, hogy két fájlt generálok, egyet a C függvényeknek, egyet a header fájlnak. A calcFileName egy saját függvény, mely a resource fájlból kiszámítja a DSL programfájl nevét, annak érdekében, hogy a generált fájlnak is ugyanaz lehessen a neve.

**override** **void** doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {

fsa.generateFile(resource.calcFileName+"\_gen.txt", resource.generate )

fsa.generateFile(resource.calcFileName+"\_gen\_header.txt", resource.generateHeader )

}

Először a könnyebbik függvényt, header fájl függvényét szeretném bemutatni. Az egész függvény, a többi generáló függvényhez hasonlóan, csupán egy multiline sztring, melyet a három nyitó és három záró aposztróf és a sötétebb háttér jelöl. Az Xtend úgy értelmezi ezt a függvényt, hogy a sztring a visszatérési érték, és az fsa.generateFile ezt a sztringet fogja fájlba írni. Így próbálja az Xtend a kódgenerálást átláthatóbbá, praktikusabbá tenni, hogy a generálandó kódot helyezi fókuszba. Továbbá megfigyelhető, a sztringből való kilépés használata is, mikor is függvényeket hívok a fájlnév kiszámítására. A generált kódban három C függvény lesz elérhető, egy az aliasoknak, egy a kódegységeknek és egy a mappingoknak.

**def** generateHeader(Resource r)'''

//Generated by HIM

#ifndef «r.calcFileName.toUpperCase»\_GEN\_H

#define «r.calcFileName.toUpperCase»\_GEN\_H

#include <stdint.h>

#include <assert.h>

#include "hashtable.h"

hashtable\* aliases\_hashbtable();

hashtable\* unit\_lenghts();

hashtable\* mappings();

#endif

'''

Ezután a C függvények implementációját generáló metódusokat mutatom be. Először egy kisebb függvény hívódik meg, mely további függvényeket hív az egyes C függvényekhez. Ezzel kicsit tagoltabb és bővíthetőbb lesz a kód.

**def** generate(Resource r) '''

//Generated by HIM

#include "«r.calcFileName».h"

«r.generateAliases»

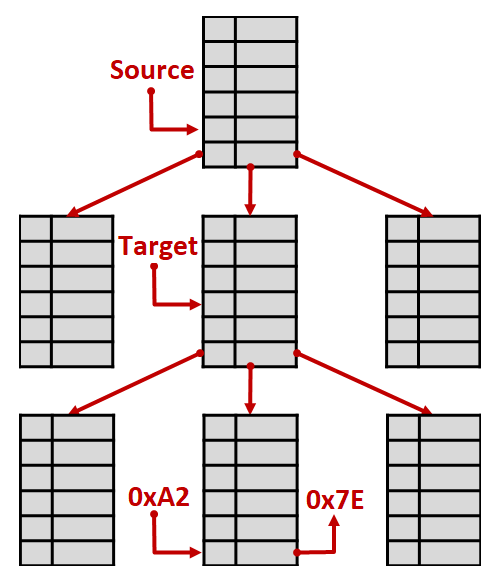
«r.generateUnitLengths»

«r.generateMappings»

'''

A három függvény közül a mappingokhoz generáló függvényét ismertetem először. Majd a másik két függvényt, melyek komplexitása kisebb, de az elvük és a megvalósításuk h

A mappingokhoz tartozó C függvényben hashtáblák hierarchiáját kell megvalósítani. A függvény visszatérési értéke egy hashtábla lesz. Ez a hashtábla további hashtáblákat tárol, melyekben a kiinduló kódolás szerinti konverziók találhatóak. Egy kiinduló kódolás hashtáblájában ugyancsak hastblákat tárolunk a cél kódolás szerint. A cél kódolás táblájában viszont már a tényleges bitkódok tárolódnak, melyek leírják az adott kiinduló kódolásból a célkódolásba átvivő konverziókat. Egy vázlatos ábra a hastáblák hierarchiájához:



A kódgenerálás ezen részének az elve, hogy Xtext által előállított Java osztályokon végigiterálunk, melyeket a DSL forráskódból származtatott le, és az egyes objektumok attribútumait leképezzük C kódra. A függvényben először is inicializálok egy hashtáblát, mely minden egyéb hashtáblát tárolni fog. Inicializáláskor meg kell adni, hogy milyen értékeket szeretnénk tárolni benne, jelen esetben sztringekhez rendelünk hashtáblákat.

**def** generateMappings(Resource r)'''

hashtable\* mappings(){

// hashtable for everything

hashtable\* mappings = hashtable\_init(128, sizeof(char[10]), sizeof(hashtable\*));

Következőnek kezdődhet az elemeken való iteráció. Ehhez valamilyen ciklus lenne a legelőnyösebb, amihez az Xtend készségesen biztosít lehetőséget. Egy ciklus létrehozásához a «FOR» … «ENDFOR» művelet használható, mellyel kilépünk az multiline sztringből és például végigiterálhatunk egy listán. Ezzel rengeteg időt spórolunk meg, és mondhatni ez a «FOR» ciklus az, ami miatt a kódgenerálás folyamata előnyösebb, mint a kézzel való kódolás. Segítségével egyszerűbb és gyorsabb lesz adott kód előállítása. A resource objektumnak, ami a modellünket biztosítja, létezik egy függvénye mellyel ki lehet szűrni adott típusú objektumokat, jelen esetben a SourceMapping elemeket. A hashtable\_put függvény helyez el kulcs-érték párokat egy hashtáblában: ebben az esetben a fő hashtáblába helyezzük el a SourceMapping nevét, illetve a hozzá tartozó hashtáblát. A felhasznált calcString függvény egységesíti a bemeneti sztringek hosszát, hogy könnyebb legyen a táblákban való tárolásuk.

«**FOR** s:r.allContents.*toIterable*.*filter*(SourceMapping)»

// hashtable for «s.name»

const char\* encoding\_«s.name» = "«calcString(s.name)»";

hashtable\* mappings\_from\_«s.name» = hashtable\_init(128, sizeof(char[10]), sizeof(hashtable\*));

hashtable\_put(mappings, encoding\_«s.name», &mappings\_from\_«s.name»);

Értelemszerűen, ezután az adott SourceMappinghoz tartozó Conversionök következnek. Itt az Xtend egy másik nyelvi elemét, az «IF» … «ENDIF» műveletet is használom. Ennek az elágazásnak a működése abból áll, hogy, ha az IF mellé true érték kerül be, akkor a «IF» … «ENDIF» közötti rész belekerül a sztringbe és így a generált kódba.

«**FOR** c:s.conversions»

«**IF** !c.mappings.isEmpty»

// hash for «s.name»->«c.name»

const char\* encoding\_«s.name»\_to\_«c.name» = "«calcString(c.name)»";

A kódnak ezen a részén, már foglalkozni kell a konverziókhoz tartozó bájt-párokkal, ugyanis ezek méretéhez kell igazítani a hashtáblák méretét. Ezek vizsgálatára szolgál a getLengthOfHash és a getIntType függvény.

hashtable\* mappings\_from\_«s.name»\_to\_«c.name» = hashtable\_init(

«getLengthOfHash(c.mappings.get(0).from.length-2)»,

sizeof(«getIntType(c.mappings.get(0).from.length-2)»),

sizeof(«getIntType(c.mappings.get(0).to.length-2)»));

hashtable\_put(mappings\_from\_«s.name», encoding\_«s.name»\_to\_«c.name», &mappings\_from\_«s.name»\_to\_«c.name»);

Itt is és a generálás többi részén is figyelni kellett arra, hogy az egyes változók nevei ne egyezhessenek meg, mert az C kód fordítása hibát eredményezne. Ennek érdekében kellett minden változóhoz belevenni a hozzá tartozó modellbeli objektum nevét.

Most már az egyes bájtsorozatokat is el tudjuk tárolni. Az egyszerűbb tárolás érdekében, nem a hexadecimális kódot tároljuk, hanem a tényleges bájtokat számok formájában. Ehhez el kellett készítenem a HexToDec osztályt Java nyelven, melynek hex2decimal függvénye ezt az átalakítást végzi el.

//filling in «s.name»->«c.name»

«getIntType(c.mappings.get(0).from.length-2)» from\_value\_«s.name»\_to\_«c.name»;

«getIntType(c.mappings.get(0).to.length-2)» to\_value\_«s.name»\_to\_«c.name»;

«**FOR** m:c.mappings»

from\_value\_«s.name»\_to\_«c.name» = «HexToDec.*hex2decimal*(m.from)»;

to\_value\_«s.name»\_to\_«c.name» = «HexToDec.*hex2decimal*(m.to)»;

hashtable\_put(

mappings\_from\_«s.name»\_to\_«c.name»,

&from\_value\_«s.name»\_to\_«c.name»,

&to\_value\_«s.name»\_to\_«c.name»);

«**ENDFOR**»

A modell bejárásával az összes hashtábla feltöltésre került. A ciklusok és elágazások lezárásával, a generált függvény visszatér a mappings nevő fő hashtáblával, a generáló függvény pedig lezárja a multiline sztringet.

«**ENDIF**»

«**ENDFOR**»

«**ENDFOR**»

return mappings;

}

'''

A függvények, melyek a másik két C függvényt generálják, a fentihez nagyon hasonlóan működnek. A különbség, hogy azok csak egy-egy hashtáblát készítenek és töltenek fel. Az aliasok hashtáblájához tartozó függvény a SourceMapping objektumokon iterál végig, melyet megint a resource objektumtól kér el. A hashtáblába a SourceMapping neve is bekerül, mely kulcshoz tartozó érték szintén a neve. Így ha a SourceMapping nevét keresnénk a listába, akkor is lesz eredmény.

**def** generateAliases(Resource r)'''

hashtable\* aliases\_hashbtable(){

hashtable\* aliases = hashtable\_init(256, sizeof(char[10]), sizeof(char[10]));

«**FOR** s: r.allContents.*toIterable*.*filter*(SourceMapping)»

//aliases for «s.name»

const char\* encoding\_«s.name» = "«calcString(s.name)»";

hashtable\_put(aliases,encoding\_«s.name»,encoding\_«s.name»);

Ezután az egyes SourceMappingok aliases listáját bejárva elmenti az Aliasok neveit a hashtáblába. Ezzel miden lényegi adatot elmentettünk, visszatérhetünk a hashtáblával és a generáló függvényt lezárhatjuk. Itt megjegyezném, hogy az üres sorok és tabulátorok (pl a két «ENDFOR» közötti szürke háttér), illetve a generált kommentek fontos szerepet játszanak a generált kód olvashatóságában.

«**FOR** a: s.aliases»

//put to «s.name» alias «a.name»

const char\* alias\_«a.name»\_for\_«s.name» = "«calcString(a.name)»";

hashtable\_put(aliases,alias\_«a.name»\_for\_«s.name»,encoding\_«s.name»);

«**ENDFOR**»

«**ENDFOR**»

return aliases;

}

'''

Az utolsó függvényben az egyes karakterkódolásokhoz tartozó kódhosszakat kell eltárolnunk, vagyis hány bájton tárolunk egy karaktert az adott kódolásban. Ehhez egy darab hashtáblát használok, melyben sztring kulcsokhoz rendelek egész számokat. A függvényt itt is egy hashtábláa inicializálásával kezdem, majd a SourceMappingok iterálást kezdem el. Az egyes karakterkódolásokhoz tartozó kódhosszakat az első megadott karakterkonverzióból számítom ki. SourceMappingok esetén azonban egy «IF» feltétellel meg kell vizsgálni, hogy tartozik-e hozzá konverzió és abban van-e megadva leképezés. Ha ugyanis nincs, de mégis hivatkozunk az első elemekre, a generáló hibába fog ütközni.

**def** generateUnitLengths(Resource r)'''

hashtable\* unit\_lenghts(){

hashtable\* unit\_lengths = hashtable\_init(128, sizeof(char[10]), sizeof(uint8\_t));

«**FOR** s: r.allContents.*toIterable*.*filter*(SourceMapping)»

«**IF** !s.conversions.isEmpty && !s.conversions.get(0).mappings.isEmpty»

//unit legnths for «s.name»

const char\* encoding\_«s.name» = "«calcString(s.name)»";

uint8\_t length\_«s.name» = «(s.conversions.get(0).mappings.get(0).from.length-2)/2»;

hashtable\_put(unit\_lengths, encoding\_«s.name», &length\_«s.name»);

Majd a SourceMappinghoz tartozó Conversionöket nézem végig és a célkódolás kódhosszát kiszámítom, majd eltárolom. Itt is meg kell vizsgálni, hogy a Conversionhöz tartozik-e konverzió, ugyanis a nyelv definíciója szerint egyet sem kötelező megadni.

«**FOR** c:s.conversions»

«**IF** !c.mappings.isEmpty»

//unit length for «c.name»

const char\* encoding\_«s.name»\_to\_«c.name» = "«calcString(c.name)»";

uint8\_t length\_«s.name»\_to\_«c.name» =  
 «(c.mappings.get(0).to.length-2)/2»;

hashtable\_put(unit\_lengths, encoding\_«s.name»\_to\_«c.name», &length\_«s.name»\_to\_«c.name»);

Végül lezárjuk a ciklusokat, visszatérünk a hashtáblával, a generáló függvényt pedig lezárjuk.

«**ENDIF**»

«**ENDFOR**»

«**ENDIF**»

«**ENDFOR**»

return unit\_lengths;

}

'''

A kódgenerátor használata

A kódgenerátor használatához az Xtext által készített plugint kell futtatnunk.  
A plug-inban már ismertetett módon van lehetőségünk a DSL nyelvnek megfelelő forráskódot szerkeszteni. A kódgenerálást úgy tudjuk elindítani, ha egy forráskódot elmentünk, feltételezve, hogy a program helyes. A mentés művelete tüzeli el a kódgenerálás eseményét (trigger), mikor is meghívódik a generátor doGenerate függvénye, ami két szövegfájl hoz létre melyek a generált C kódokat tartalmazzák.

Egy rövid példán keresztül szemléltetni a dolgozatom lényegi részét, a tényleges kódgenerálást. Vegyük az alábbi DSL programot:

**source** test2 {

**alias** TesT

**target** a {

0xab = 0x45ef

}

}

A fájl neve test2.enco. Mentés után létrejönnek a test2\_gen\_header.txt és a test2\_gen.txt fájlok. A header fájl azt tartalmazza, amit a generáló függvényben megírtunk. Az include guardnál figyelhető meg, hogy helyesen lett kiszámítva a fájl neve.

//Generated by HIM

#ifndef TEST2\_GEN\_H

#define TEST2\_GEN\_H

#include <stdint.h>

#include <assert.h>

#include "hashtable.h"

hashtable\* aliases\_hashbtable();

hashtable\* unit\_lenghts();

hashtable\* mappings();

#endif

A másik fájlban, a test2\_gen.txt-ben a függvények implementációja található. Elől találjuk az aliasok függvénye. Látható, hogy lesznek egységes hosszúságúak a sztringek, mennyire átlátható a program a whitespacek miatt és hogy a változók nevei egyediek.

//Generated by HIM

#include "test2.h"

hashtable\* aliases\_hashbtable(){

hashtable\* aliases = hashtable\_init(256, sizeof(char[10]), sizeof(char[10]));

//aliases for test2

const char\* encoding\_test2 = "test2\0\0\0\0";

hashtable\_put(aliases,encoding\_test2,encoding\_test2);

//put to test2 alias TesT

const char\* alias\_TesT\_for\_test2 = "TesT\0\0\0\0\0";

hashtable\_put(aliases,alias\_TesT\_for\_test2,encoding\_test2);

return aliases;

}

A következő függvény a kódhosszak függvénye. Itt a kódhosszak értékei helyesen lette kiszámolva, ugyanis a 0xAB tárolásához egy, míg a 0x45EF tárolásához 2 bájtra van szükségünk.

hashtable\* unit\_lenghts(){

hashtable\* unit\_lengths = hashtable\_init(128, sizeof(char[10]), sizeof(uint8\_t));

//unit legnths for test2

const char\* encoding\_test2 = "test2\0\0\0\0";

uint8\_t length\_test2 = 1;

hashtable\_put(unit\_lengths, encoding\_test2, &length\_test2);

//unit length for a

const char\* encoding\_test2\_to\_a = "a\0\0\0\0\0\0\0\0";

uint8\_t length\_test2\_to\_a = 2;

hashtable\_put(unit\_lengths, encoding\_test2\_to\_a, &length\_test2\_to\_a);

return unit\_lengths;

}

A karakterkonverziók függvénye a fájl legvégén található. Itt helyesen három hashtábla lett inicializálva, melyek hierarchiája is megfelelő. Az utolsó hashtáblába elhelyezett integer értékek is helyénvalóak, hiszen 0xAB = 171 és 0x45EF = 17903. Továbbá az értékeket tároló integerek mérete is helyes (uint8\_t, uint16\_t).

hashtable\* mappings(){

// hashtable for everything

hashtable\* mappings = hashtable\_init(128, sizeof(char[10]), sizeof(hashtable\*));

// hashtable for test2

const char\* encoding\_test2 = "test2\0\0\0\0";

hashtable\* mappings\_from\_test2 = hashtable\_init(128, sizeof(char[10]), sizeof(hashtable\*));

hashtable\_put(mappings, encoding\_test2, &mappings\_from\_test2);

// hash for test2->a

const char\* encoding\_test2\_to\_a = "a\0\0\0\0\0\0\0\0";

hashtable\* mappings\_from\_test2\_to\_a = hashtable\_init(

512,

sizeof(uint8\_t),

sizeof(uint16\_t));

hashtable\_put(mappings\_from\_test2, encoding\_test2\_to\_a, &mappings\_from\_test2\_to\_a);

//filling in test2->a

uint8\_t from\_value\_test2\_to\_a;

uint16\_t to\_value\_test2\_to\_a;

from\_value\_test2\_to\_a = 171;

to\_value\_test2\_to\_a = 17903;

hashtable\_put(

mappings\_from\_test2\_to\_a,

&from\_value\_test2\_to\_a,

&to\_value\_test2\_to\_a);

return mappings;

}

A kódgenerátor tesztelését ues casek kipróbálásával teszteltem a fenti módon. Különböző darabszámú SourceMappinget, Conversiont és Aliast tartalmazó programokból generáltam kódot. Továbbá tekintettem voltam az egyes értékekre, azok határaira és a hozzájuk illő C-beli változótípusokra. A generátort ezen eredmények függvényében módosítottam, ha szükséges volt.

A keretprogram

A generált kódokhoz készítettem egy C keretprogramot, mely a felhasználja az eltárolt adatokat. A generált kód és a keretprogram az iconv függvényeket valósítják meg.[[4]](#footnote-4) Az iconv karakterkódolások közötti konverzióhoz ad lehetőséget. A iconv követelményeit és működését a keretprogramom header fájlján keresztül szeretném bemutatni.

A header fájlban először is a következő headeröket importáljuk: stdlib.h; errno.h; hashtable.h, mellyel hashtábla-műveleteket végezhetünk; és a test3.h, mely a generált kód headerje.

#ifndef ICONV\_HIM

#define ICONV\_HIM

#include<stdlib.h>

#include<errno.h>

#include "test3.h"

#include "hashtable.h"

A függvények használatakor egy iconv\_t nevű struktúrát kell lekérnünk és használnuk, mely tartalmaz minden információt, mely egy adott konvertáláshoz szükségesek. Azonban, hiba esetén az struktúrát előállító függvénynek -1 értékkel kell visszatérnie, melyet át kell kasztolnia az iconv\_t struktúrára. Ennek a problémának a megoldására létrehoztam egy másik struktúrát mely a tényleges információkat tartalmazza és a neve iconv\_t\_data. Az iconv\_t pedig egyetlen darab void\* lesz, melyre át lehet kasztolni egy iconv\_t\_datat vagy egy -1-et. Mellesleg, más iconv implementációk is ezen az elven működnek.[[5]](#footnote-5)

typedef struct {

hashtable \*allmappings;

hashtable \*mapping;

uint8\_t from\_length;

uint8\_t to\_length;

} iconv\_t\_data;

typedef void\* iconv\_t;

Ezután három függvény deklarációja jön, melyek a használatukat is sejtetik. Először az iconv\_open függvénnyel lekérünk egy iconv\_t-t adott kiinduló- és célkódoláshoz. Ezután az iconv függvénnyel átkonvertáljuk a kívánt buffereket. Végül a lezárjuk az iconv\_t struktúrát az iconv\_close segítsgégével.

iconv\_t iconv\_open(const char \*tocode, const char \*fromcode);

int iconv\_close(iconv\_t cd);

size\_t iconv(iconv\_t cd,

char \*\*restrict inbuf, size\_t \*restrict inbytesleft,

char \*\*restrict outbuf, size\_t \*restrict outbytesleft);

#endif // ICONV\_HIM

A header fájl végére érve a függvények implementációját ismertetem. Értelemszerűen, az első függvény az incov\_open. Ez a függvény használja fel a generált kódot. Egyesével lekéri a hashtáblákat, és a hashtáblákból eléri a kívánt adatokat. Ha hibás bemenet miatt egy hashtáblában nem találja meg a kívánt értéket, hibát jelez. A kívánt adatok, melyeket egy iconv\_t\_data struktúrában tárol, a következők: a két kódolás kódhossza, a konverziókat tartalmazó hashtábla, illetve az összes konverziót tartlamazó hashtábla, hogy használat után fel tudjuk szabadítani. Legelőször a kiinduló kódolásnak keressük meg a megfelelő elnevezését. A calcString függvény átalakítja a sztringeket, hogy egységnyi legyen a hosszuságuk.

iconv\_t iconv\_open(const char \*tocode2, const char \*fromcode2)

{

char\* tocode=calcString(tocode2);

char\* fromcode=calcString(fromcode2);

iconv\_t\_data \*returnval=(iconv\_t\_data\*) malloc(sizeof(iconv\_t\_data));

char \*fromcode\_realname= (char\*) malloc(sizeof(char[10]));

hashtable \*aliases = aliases\_hashbtable();

if( hashtable\_get(aliases,fromcode,fromcode\_realname)

== HASH\_NOTFOUND )

{errno=EINVAL;return ((iconv\_t)-1);}

hashtable\_free(aliases);

Ezután az egyes kódhosszakat állítjuk be.

hashtable\* lengths = unit\_lenghts();

if( hashtable\_get(lengths, fromcode\_realname, &(returnval->from\_length))

== HASH\_NOTFOUND ||

hashtable\_get(lengths, tocode, &(returnval->to\_length))

== HASH\_NOTFOUND)

{errno= EINVAL;return (iconv\_t) -1;}

hashtable\_free(lengths);

Legvégül a konverziónak a hashtábláját keressük ki a rekordok közül.

returnval->allmappings = mappings();

hashtable \*frommapping;

if( hashtable\_get(returnval->allmappings,fromcode\_realname,&frommapping)

== HASH\_NOTFOUND)

{errno= EINVAL;return (iconv\_t) -1;}

if( hashtable\_get(frommapping, tocode, &(returnval->mapping))

== HASH\_NOTFOUND)

{errno= EINVAL;return (iconv\_t) -1;}

return (iconv\_t)(void\*)returnval;

}

Következő függvény az iconv. Ez a függvény egy bemeneti buffert konvertál át az adott iconv\_t szerint, és az átalakított bájtokat egy kimeneti bufferbe menti el (a kimeneti buffer is egy paramétere a függvénynek, csupán a neve az, hogy kimeneti). Először is megvizsgálja, hogy használhatóak-e a bemeneti paraméterek.

size\_t iconv(iconv\_t cd,

char \*\*restrict inbuf, size\_t \*restrict inbytesleft,

char \*\*restrict outbuf, size\_t \*restrict outbytesleft)

{

if(cd==(iconv\_t)-1)

{errno= EBADF;return (size\_t)-1;}

iconv\_t\_data \*con=(iconv\_t\_data\*) cd;

if(\*outbytesleft<con->to\_length)

{errno = E2BIG;return (size\_t)-1;}

Ezután egy ciklus következik, mely az átalakítást végzi. Az átalakítás első lépése, hogy kiveszünk annyi bájtot a bemeneti bufferből, amennyi a kiinduló kódolás kódhossza. Ehhez bitműveleteket használok.

do

{

fromvalue=(\*inbuf)[0];

for(int i=2; i<=con->from\_length; i++)

fromvalue=(fromvalue<<8)+(\*inbuf)[0+i-1];

Második lépésként a hashtáblából lekérdezem a bemeneti értékhez tartozó kimeneti értéket. Ha ilyen kulcshoz nincs érték a táblában, akkor a kimeneti érték ugyanaz lesz, mint a bemeneti.

if(hashtable\_get(con->mapping,&fromvalue,&tovalue)==HASH\_NOTFOUND)

tovalue=fromvalue;

Majd a kimeneti értékeket bájtonként bemásolom a kimeneti bufferbe, szintén bitműveletek segítségével.

char b;

for(int i=1; i<=con->to\_length; i++)

{

b=(tovalue&(0xFF<<((con->to\_length-i)\*8)))>>((con->to\_length-i)\*8);

\*\*outbuf=b;

\*outbuf=\*outbuf+sizeof(char);

}

Az iconv definíciója szerint, a bufferek pointereinek a függvény végén az utolsó sikeres konverzióra kell mutatniuk. Vagyis megvizsgálom, hogy lesz-e még sikeres konverzió, pontosabban lesz-e elég hely a bufferekbe, és ha igen, módosítom a pointerket.

if(\*inbytesleft>=con->from\_length && \*outbytesleft>=con->to\_length)

{

\*inbuf=\*inbuf+con->from\_length;

\*inbytesleft=\*inbytesleft-con->from\_length;

\*outbytesleft=\*outbytesleft-con->to\_length;

}

}

Végül lezárom a ciklust és a függvényt.

while(\*inbytesleft>=con->from\_length && \*outbytesleft>=con->to\_length);

return 0;

}

Az utolsó függvény az iconv\_close, mely az icon\_t által foglalt memóriát szabadítja fel. A függvény ellenőrzi a bemeneti iconv\_t változó helyességét, majd átkasztolja iconv\_t\_data típusúvá, és a konverziók fő hashtábláját felszabadítja a hashtable\_free függvény segítségével.

int iconv\_close(iconv\_t cd)

{

if(cd!=(iconv\_t)-1)

{

iconv\_t\_data \*x = (iconv\_t\_data\*) cd;

hashtable\_free(x->allmappings);

}

return 0;

}

Az iconv függvényeim ismertetése után egy rövid példán keresztül bemutatom a használatát. Készítsünk a DSL nyelvben egy programot, mely olyan konverziót ír le, ahol a kis e betűt nagy E betűre, az x betűt pedig 0 karakterré kell konvertálni. Egy ilyen program például a következő: (e=0x65, E=0x45, x=0x78, 0=0x30)

**source** ascii{

**alias** Ascii

**target** UTF8{

0x65=0x45

0x78~0x30

}

}

**source** a{

**target** b{

0x45=0x45BA20

}

}

A DSL programból generált kódot includeoljuk az iconv függvényekhez. Az iconv függvényeket a fent ismertetett módon és sorrendben meghívjuk. A konvertálandó sztring a híres „the quick brown fox jumps over the lazy dog”.

const char \*kiindulo = "Ascii";

const char \*cel = "UTF8";

iconv\_t cd = iconv\_open(cel,kiindulo);

char \*inbuf="the quick brown fox jumps over the lazy dog";

size\_t inbytesleft=strlen(inbuf)+1;

char \*outbuf=(char\*) malloc(sizeof(char[44]));

char \*outbuf2 = outbuf;

size\_t outbytesleft=sizeof(char[44]);

iconv(cd,&inbuf,&inbytesleft,&outbuf,&outbytesleft);

printf("%s\n",outbuf2);

iconv\_close(cd);

A program lefutása után a kimeneten a „thE quick brown fo0 jumps ovEr thE lazy dog” sztring jelenik meg, vagyis a konvertálás sikeres volt.

1. TODO https://github.com/gaborbsd/FREC/blob/master/lib/libfrec/hashtable.c [↑](#footnote-ref-1)
2. TODO Lorenzo Bettini – Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing 2016. , ISBN 978-1-78646-496-5 [↑](#footnote-ref-2)
3. TODO Kent Beck - Test Driven Development: By Example, Addison-Wesley Professional 2002, ISBN: 978-0321146533 [↑](#footnote-ref-3)
4. TODO http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/iconv.h.html [↑](#footnote-ref-4)
5. TODO https://github.com/git-for-windows/git-sdk-64/blob/master/mingw64/include/iconv.h [↑](#footnote-ref-5)