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

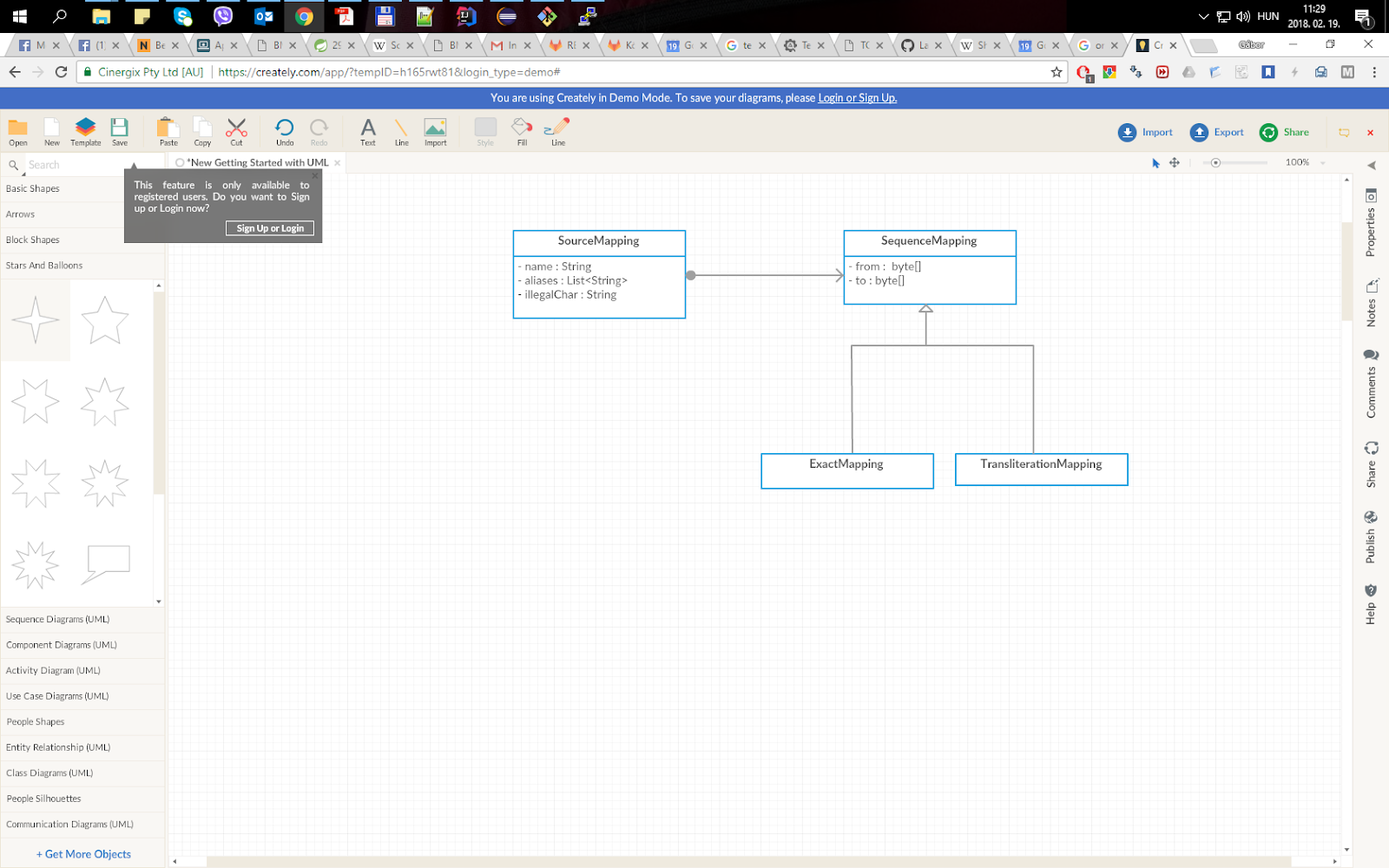
0x0001 = 0xff01 // this will be an exact mapping

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)

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.

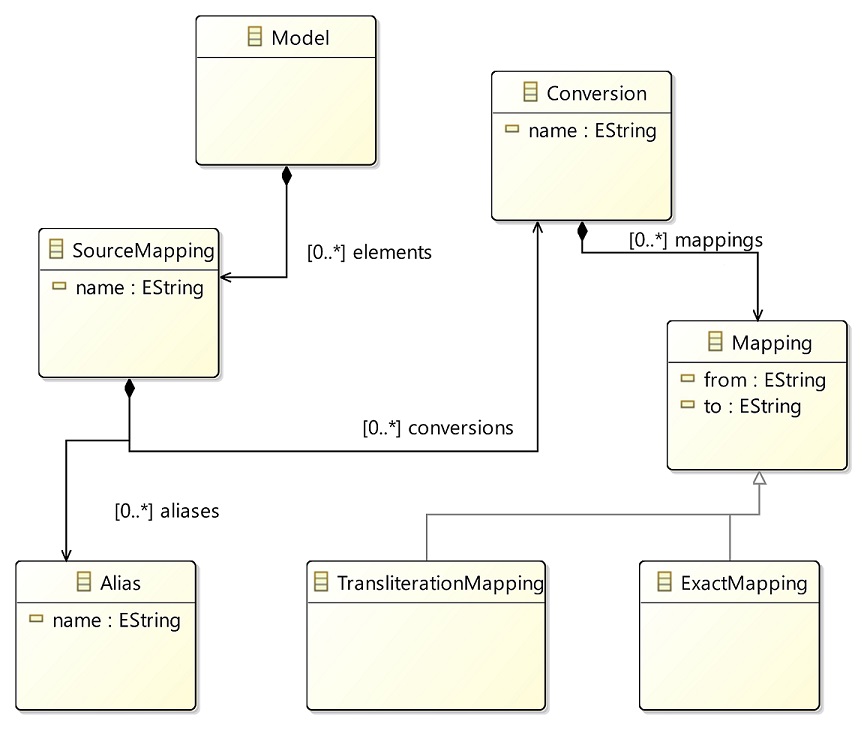
Az én DSL nyelvemben egy új terminált is létre kellett hoznom, melynek neve INPUTCHAR lett. A terminál segítségével a karakterkódolásokban használt kódpontokat lehet hexadecimális formában leírni. Egy ilyen karakterlánc egy nullás karakterrel kezdődik, x-szel folytatódik, majd utána páros számú hexadecimális 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'))+

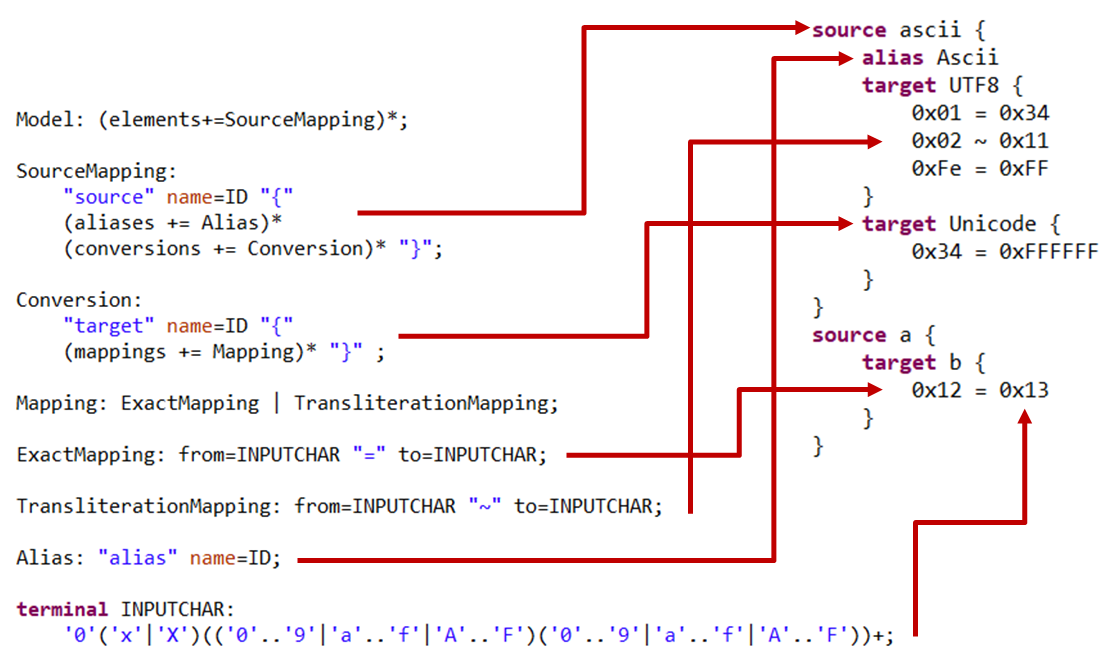
;

Ezzel elkészült a DSL nyelv nyelvtana. 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 képes voltam 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 okazhat 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.

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)