

SZAKDOLGOZAT

Lócsei Tamás

Debrecen

2011

Debreceni Egyetem

Informatikai Kar

Dokumentumformátumok adaptálása XML-be

Témavezető:

Jeszenszky Péter

egyetemi adjunktus

Készítette:

Lőcsei Tamás

Programtervező Informatikus BSc

Debrecen

2011

Köszönetnyilvánítás

Ezúton szeretném köszönetemet kifejezni témavezetőmnek, Jeszenszky Péternek, hogy hasznos meglátásai és iránymutatásai alapján sikerült a dolgozatot elkészítenem. Továbbá külön köszönet azért, hogy közvetítésével megismerkedhettem a dolgozat során bemutatásra kerülő technológiákkal, amely ismereteknek a jövőben is hasznát tudom majd venni.

Tartalomjegyzék

1. Bevezetés	4
1.1. Motiváció	4
1.2. Célok	5
2. Felhasznált eszközök és technológiák	7
2.1. Java	7
2.2. Eclipse IDE	7
2.3. XML	8
2.3.1. Áttekintés	8
2.3.2. Felépítés	10
2.3.3. Jólformázottság	13
2.3.4. Érvényesség	14
2.3.5. Névterek	15
2.4. MathML	15
2.5. XML Schema	16
2.6. StAX API	17
2.7. parboiled	19
2.8. Apache Maven	19
2.9. L ^A T _E X	21
2.9.1. T _E X	21

2.9.2. $\text{\LaTeX}$	21
3. XML alapú nyílt dokumentumformátumok	23
3.1. Open Document Format	23
3.2. DocBook	24
4. Nyelvek közötti átalakítás	25
4.1. Szintaktikai elemzés	25
4.1.1. Reguláris kifejezések	26
4.1.2. Szintaktikai elemzők	26
4.2. Átalakítási módszerek	27
4.2.1. Szintaxis-vezérelt módszer	27
4.2.2. Szabályalapú módszer	28
4.2.3. Modellvezérelt átalakítási módszer	28
4.3. Köztes forma	28
4.3.1. Fa adatszerkezet	29
4.3.2. Értelmezési fa	29
4.3.3. Absztrakt szintaxisfa	30
4.4. A szintaxisfa bejárása	30
4.4.1. Látogató tervezési minta	31
4.4.2. Látogatás külső osztályból	32
5. A fejlesztett alkalmazás	34
5.1. Követelmények	35
5.2. Használat	36
5.3. Felépítés	37
5.4. Külső adatforrások	39
5.5. Nyelvtani elemek	39
5.6. A szintaxisfa felépítése	44

5.7. Látogatók	57
6. Összegzés	63
Irodalomjegyzék	65
Felhasznált könyvek	65
Internetes források	66
Hivatkozások	68

1. fejezet

Bevezetés

1.1. Motiváció

Napjainkban az Internet vált az első számú publikációs felületté. Az idők során felhalmozódott ismeretanyagok közül sok tudományos publikáció, szakdolgozat, egyetemi jegyzet és egyéb, nyomtatásra szánt dokumentum keletkezett a $\text{\LaTeX}$ felhasználásával. Népszerűsége annak köszönhető, hogy viszonylag kis energia befektetésével igen jó minőségű könyvek, cikkek és egyéb iratok formázhatók meg vele. A matematikai képletek elektronikus tárolásánál pedig etalonnak számít. A publikálás könnyedén megoldható vele különböző platformfüggetlen formátumokat felhasználva kimenetként. Ilyenek lehetnek például a PDF és DVI formátumok.

Manapság azonban egy újabb igény jelent meg, amely szerint a dokumentumokat a felhasználók szívesebben böngésszik azok kinyomtatása, illetve külön programok feltelepítése nélkül. A webböngészőké lett a főszerep az információszerzés területén, főként az információk hatalmas mennyisége miatt. Az információnak hordozhatónak, számítógépek által feldolgozhatónak és kereshetőnek kell lennie. A webes felületen történő információmegjelenítés eszközeként az XML ért el kiemelkedő sikert. Éppen ezért merülhet fel az igény arra, hogy az évek során felhalmozott dokumentumokat – melyek egyébként nyomtatásra kiválóan alkalmas módon

lettek megformázva – át kellene alakítani a weben is használt formátumba. Ezeknek az átírása rendkívül időigényes lenne. Ugyanakkor ezeket a dokumentumokat szerzőik talán hosszú hetek-hónapok munkája eredményeként hozták létre. Így nem várható el tőlük, hogy újragépeljék az egészet, ráadásul közben egy újabb nyelvet is megtanuljanak. Az átalakítás folyamata igen költséges is lehet, az egyes dokumentumok mennyiségével és tartalmi bonyolultságával arányosan. Tehát szükség lenne ennek a műveletnek az automatizálására.

A másik fontos érv az átalakítás mellett a platformfüggetlenség. A létrehozott $\text{\LaTeX}$ dokumentum maga ugyan platformok között könnyedén hordozható (mivel egyszerű szövegfájlról van szó), azonban minden operációs rendszeren külön makró-értelmező hajtja végre a dokumentum feldolgozását. Emellett megemlítenéd még a feladatot végző alkalmazás telepítéséhez igényelt tárterület nem csekély mérete. Gond lehet még az esetleg kevésbé hozzáértő felhasználók számára a megfelelő értelmező alkalmazás kiválasztása, megtalálása és megfelelő beállítása is. Újabb előny lehet az XML szempontjából, hogy rendkívül könnyen feldolgozható dokumentumformátumot eredményez a használata, így a konverzió is könnyebb lesz egyéb formátumokra, ha esetleg igény mutatkozna rá a jövőben.

1.2. Célok

A célkitűzés egy magas szintű programozási nyelv segítségével elkészített szoftver kifejlesztése, amely $\text{\LaTeX}$ dokumentumokból képes egy előre definiált, valamint a felhasználó által is bővíthető, módosítható felépítésű XML dokumentum előállítására. A dolgozat folyamán ennek az alkalmazásnak, illetve az átalakításnál felhasznált módszereknek a bemutatására kerül sor.

Az elkészült program a bővíthetőség érdekében lehetőséget biztosít XML alapú konfigurációs állományokon keresztül az előre alapértelmezetten bekonfigurált nyelvi elemeken kívül egyéb elemek megadására is. A külső állományok segítségével tehát lehetővé válik a már megadott adatok módosítása a program forrásszövegének átalakítása nélkül. Ezáltal a

program segítségével generált kimenet megváltoztatása kivitelezhető anélkül, hogy a forráskód módosítására szükség lenne. Ennek eredményeképpen jelentősen könnyebb lesz az újabb – a program számára eddig ismeretlen – nyelvi elemek kezelésének megvalósítása, valamint a felhasználó által definiált saját eszközök támogatásának fejlesztői tapasztalat nélküli beépítése.

A programozásban jártasabb felhasználók számára a szoftver lehetőséget biztosít egyéb kimeneti formátumok létrehozására is, azaz rugalmasan bővíthető architektúrájú is egyben.

2. fejezet

Felhasznált eszközök és technológiák

2.1. Java

A fejlesztés Java nyelven valósult meg. A Java nyelvet a SUN Microsystems-nél [H14] fejlesztették ki és 1995-ben debütált. Jelenleg az Oracle tulajdonában áll. A szintaktikája a C, C++ nyelvekből származtatott és a memóriakezelésének egyszerűsítése miatt széles körben elterjedté vált az évek során. Objektum-orientált paradigmát vall, amely az alkalmazás kifejlesztése során nagyban ki is lett használva, így a mai kor igényeinek szakmai szempontból is megfelelő programot eredményez ennek a nyelvnek a használata. A projekt szempontjából lényeges még, hogy igen terjedelmes az elérhető értelmező („parser”) eszközök listája erre a platformra. Előnyös tulajdonsága még a platformfüggetlensége, amely a létrehozott szoftvert így igen széles körben használhatóvá teszi operációs rendszerkörnyezettől függetlenül.

2.2. Eclipse IDE

A fejlesztési folyamat egyes lépéseinek megkönnyítésére az Eclipse IDE [H5] lett felhasználva. Ez egy széles körben elterjedt, szabadon elérhető fejlesztői környezet a Java

fejlesztők számára. Azonban beépülő modulkezelő rendszere segítségével számos más nyelv támogatására is képes.

A fejlesztést a következő leggyakrabban használt képességei teszik könnyebbé:

- Tartalmaz szövegszerkesztőt, amely a forráskód megfelelő elrendezésének segítségével, illetve a programnyelv kulcsszavainak színezésével teszi átláthatóbbá a létrehozott kódot.
- Fordítóprogramot, amely a JVM által értelmezhető „Java bytecode”-ra fordítja az elkészített forráskódot.
- Hibakeresőt („debugger”), amely a futás közbeni hibák felderítésében van a fejlesztő segítségére.
- Kódgenerátort, amely a gyakran használt kódrészletek automatikus létrehozásával gyorsítja a munkát (például: „getter/setter” generálása, metódustörzs generálása stb.)
- Automatikus kiegészítő eszközt, amellyel ugyancsak rövidíti a monoton kódolással töltött időt.

Ezekon kívül számos egyéb, a program összeállítását elősegítő eszközt is szolgáltat a felhasználó számára, amely gyorsítja a munkát.

Minden platformon elérhető, amelyre van Java virtuális gép, azaz Java program futtatására képes.

2.3. XML

2.3.1. Áttekintés

Az XML [K5][I13, I12, I10, I2] az „Extensible Markup Language” rövidítése, amelynek leginkább megfelelő magyar fordítás a „kiterjeszhető jelölőnyelv” lehet. Strukturált felépítésű dokumentumok ábrázolására szolgál. Az XML az SGML-en [H12] alapul, – amely egy

korábban hasonló céllal létrehozott szabvány – azonban elődjénél jóval egyszerűbb használatot tesz lehetővé. A dokumentumban tárolt szerkezeti elemeket az úgynevezett címkék jelölik. Kiterjeszthetősége abból adódik, hogy a szerkezetét meghatározó elemek azonosítására szolgáló címkekészlete nem kötött, hanem a felhasználóra vannak bízva a címkék elnevezései.

Az XML egy W3C ajánlás, amely nem egy konkrét nyelvet ad meg. Az XML a felhasználás szempontjainak megfelelően specifikált leírónyelv létrehozásához biztosít eszközkészletet. Megalkotásakor a legfőbb cél egy olyan nyelv kialakítása volt, amely mind az emberek, mind a gépek számára viszonylag egyszerűen olvasható, értelmezhető. Legfontosabb tulajdonságai az egyszerűség, általánosság és a weben való felhasználhatóság.

Mivel napjaink programozási nyelveinek mindegyike lehetővé teszi XML dokumentumok feldolgozását, így az XML dokumentumok létrehozása és kezelése a felhasználó számára egyszerűen kivitelezhető. Az egyes alkalmazások egy úgynevezett XML-feldolgozón („XML processor”) keresztül kapják meg a dokumentum által tárolt adatokat további felhasználásra.

Rendkívül elterjedtté váltak az irodai alkalmazások körében az XML alapú tárolási formátumok. Így például tárolási formátuma a Microsoft Office, az OpenOffice illetve LibreOffice és az Apple iWork irodai programcsomagoknak. Egyes programok konfigurációs állományként hasznosítják, míg mások egyfajta adatbázisként a működésük során felhasználandó adatok tárolására. Persze számtalan egyéb felhasználási módja létezik még az XML-nek.

Előnyei:

- Könnyen tanulható.
- A létrehozott dokumentum egyszerű felépítésű.
- Gyártó- és platformfüggetlen.
- Az évek során „de facto” szabvánnyá vált az iparban az adatsere területén, így ismerete elengedhetetlen.

Az XML szabvány célkitűzései:

1. A weben könnyen használható legyen.
2. Az alkalmazások széles körét támogassa.
3. Elődjével, az SGML-lel kompatibilis legyen.
4. A létrehozott dokumentumok feldolgozására egyszerűen lehessen programokat írni.
5. Az általa nyújtott opcionális lehetőségek száma minimális legyen.
6. A dokumentumok forrásának világosnak és jól olvashatónak kell lennie.
7. A szabványt rövid idő alatt ki kell dolgozni.
8. A szabvány formális és tömör legyen.
9. A segítségével felépített dokumentumok legyenek egyszerűen elkészíthetők, szerkeszthetők.
10. A létrejött dokumentumoknál a tömörség másodlagos szerepű.

2.3.2. Felépítés

Egy XML dokumentum egy egyszerű szöveges állomány, amely a W3C XML ajánlás által specifikált Unicode karaktereket tartalmazhat. Az XML teljesen nyelv-, alkalmazás- és platformfüggetlen.

A dokumentum két részre bontható:

- A fejrész tartalmazhat:
 - XML deklarációt: Megadható benne a felhasznált XML specifikáció verziószáma, a dokumentum kódolása. A specifikáció ugyan előírja a használatát, de nem kötelező megjelennie a dokumentumban. Ha szerepel XML deklaráció, akkor annak a

dokumentum legelején kell elhelyezkednie. A leggyakoribb formája:

```
<?xml version="1.0" encoding="UTF-8"?>
```

- Dokumentumtípus-deklarációt (DTD [H4]): Megadhatók benne a dokumentum tartalmára, szerkezetére vonatkozó információk. Az XML deklaráció után kell szerepelnie, ha van.
- A törzs a lényegi tartalmat hordozza. Ebben szerepel az úgynevezett gyökérelem, amely tartalmazza az összes többi elemet. Az XML dokumentumnak mindig rendelkeznie kell egy gyökérelemmel.

A dokumentum logikai szerkezete elemekből épül fel. Az elemek tartalma a következő lehet:

- egyéb elemek,
- attribútumok (az elemek nyitó címkéjében),
- karakteres adat (szöveg),
- a fentiek tetszőleges kombinációja (a jólformázottságot előíró feltételek betartása mellett).

Az elemek egymással hierarchikus viszonyban vannak. A hierarchia alapján a dokumentumot reprezentáló fa építhető, amelynek gyökere a dokumentum gyökéreleme. Erre a szemléletre épül a DOM [H3] szabvány, amely az XML dokumentum elemeinek elérését és módosítását teszi lehetővé egy fastruktúrán keresztül. Minden egyes elem a fában csomópontként jelenik meg, amely csomópontokkal műveletek végezhetők.

Az XML dokumentumok felépítésük szerint kétféle szerkezeti elemet tartalmaznak: jelölőket és karakteres adatot. A jelölőelemek által használt szintaxis: "<" jellel kezdődnek és ">" jellel fejeződnek be, valamint előfordulhatnak olyan formában, hogy "&" karakterrel kezdődnek és ";" karakterrel záródnak. Jelölők a következők: nyitó, záró és üreselem címkék, megjegyzések, dokumentumtípus deklaráció, feldolgozási utasítások, CDATA rész határolók, egyedhivatkozások, karakterhivatkozások, jelentés nélküli térközök. Az

előzőekben felsoroltakon kívül minden más dokumentumelem karakteres adatnak számít. A karakteres adatok hordozzák a dokumentumban tárolt tényleges információtartalmat. A megjegyzések használatával javítható, a dokumentum olvashatósága és érthetősége. Ezek bárhol megjelenhetnek a jelölőkön kívül, kivéve az XML deklaráció előtt. A megjegyzések szintaxisa:

```
<!-- Ez itt egy megjegyzés -->
```

A jelölőelemek megadásánál használt speciális karakterek nem jelenhetnek meg közvetlenül szövegben. Ezek helyett úgynevezett egyedhivatkozásokat kell használnunk. Ilyenek a következők: & ("&"), < ("<"), > (">"), ' ("'"), " ("").

Az elemek elnevezésénél a következő szabályokat kell figyelembe venni:

- A névnek betűvel vagy alulvonás ("_") karakterrel kell kezdődnie. Ezt követően szerepelhetnek: betűk, számok és egyéb karakterek ("_", "-", ".", ":").
- A nevek nem tartalmazhatnak „whitespace” karaktereket.
- A név nem kezdődhet az „xml” karaktersorozattal, akár kis-, akár nagy betűkből áll (pl. XML, Xml stb.).

Az elemek tulajdonságai:

- Minden elemet egy nyitó és egy záró címke határol. Kivétel ez alól az üres elem, melyet úgynevezett üreselem-címke ábrázol.
- A nyitó (<elemnév>), záró (</elemnév>) és üreselem (<elemnév/>) címkékben megadott neveket elemtípusnak nevezzük.
- Az elemekhez – a nyitó, illetve üreselem címkékben – megadhatók egyenlőségjellel elválasztott név-érték párok, amelyek az adott elemről további információt hordozhatnak.
(attribútumnév="érték")

Feldolgozási utasítások: A dokumentumban bárhol megjelenhetnek a jelölőkön kívül. A feldolgozó alkalmazások vezérlésére szolgálnak.

Megjelenési formájuk: <? név szöveg ?>, ahol a név azt a feldolgozó alkalmazást

azonosítja, amelynek az utasítás szól. A szöveg magát az utasítást tartalmazza, amelyet a névben hivatkozott alkalmazás dolgoz fel.

2.3.3. Jólformázottság

Az XML dokumentumnak meg kell felelnie néhány olyan alapvető szabálynak, amelyek betartása biztosítja, hogy az beolvasható legyen az XML feldolgozó alkalmazások számára. Ezeknek a szabályoknak a betartása esetén a dokumentumot jól formázottnak mondjuk.

Ezen szabályok közül néhány:

- A dokumentumnak XML deklarációs utasítással kell kezdődnie.
- A dokumentumnak pontosan egy gyökérelemmel kell rendelkezni.
- Az elemeket megfelelően kell egymásba ágyazni: Azaz amelyik elemben megnyitottunk egy elemet, ugyanabban az elemben is kell azt lezárni.
- Minden egyes elemnek rendelkeznie kell nyitó és záró címkével. Kivétel ez alól az üres elem, amelyet úgynevezett üreselem-címke is ábrázolhat.
- Az XML kis- és nagy betűkre érzékeny nyelv. Ennek következtében különös figyelmet kell fordítani annak ellenőrzésére, hogy a nyitó címke lezáró párjaként, annak elnevezésével betűre pontosan megegyező címkét kell használnunk.
- Az attribútumok csak az elem nyitó címkéjében jelenhetnek meg, kivéve az üreselem-címke esetében, amely ugyancsak tartalmazhat attribútumokat.
- Az attribútumokhoz tartozó értékeket idézőjelek vagy aposztrófok között kell szerepeltetni.

2.3.4. Érvényesség

A felhasználók akár egy egyszerű szövegszerkesztő segítségével is könnyedén változtathatnak az XML dokumentumok tartalmán. Azonban a feldolgozó számítógépes alkalmazások szempontjából nem mindegy, hogy a módosított állomány szerkezetileg és tartalmilag hogyan is néz ki. Ha dokumentum változtatása érinti az értelmezését végző feldolgozó alkalmazás működési mechanizmusát, akkor a tárolt adatok beolvasására tett próbálkozások hibát eredményezhetnek.

Éppen ezért szükséges egy olyan módszer, amelynek segítségével bizonyos korlátok között tarthatók a változtatások a dokumentum szerkezetére, és részben tartalmára vonatkozóan. A legkorábban megjelent megoldás erre a problémára a DTD volt. Ennek a segítségével különböző megszorításokat lehet tenni a dokumentumot alkotó elemekre vonatkozóan, azonban a nyelve nem XML alapú, kezelése nehézkes és az általa nyújtott eszközkészlet is egy idő után kevésnek bizonyult.

Ezeknek a problémáknak a megoldására kerültek kifejlesztésre az XML sémák. Az XML sémák segítségével definiálható, hogyan is kell egy, az adott XML leírónyelvet reprezentáló dokumentumnak kinéznie. A sémát alkotó szabályrendszer definiálására sémaleíró nyelvek szolgálnak. Ezeknek segítségével bizonyos megszorításokat tehet a szerző arra vonatkozóan, hogy az általa alkotott XML alapú nyelvet felhasználva, milyen logikai struktúra szerint kell a dokumentumokat felépíteni. A legismertebb ilyen nyelvek a következők: XML Schema, RelaxNG, Schematron [H11].

Egy XML dokumentumot érvényesnek tekintünk, ha egy sémaleíró nyelv segítségével megadott séma szabályainak megfelel. Az érvényesség meghatározására rendelkezésre állnak érvényesség-ellenőrző eszközök („XML validator”). Ha egy dokumentum megfelel az alkalmazás fejlesztője által kidolgozott követelményeknek, akkor elviekben minimalizálható az értelmezési problémákból fakadó hibák valószínűsége a programban.

2.3.5. Névterek

Ha különböző XML dokumentumok tartalmának részét vagy egészét kívánjuk egy dokumentumban újra felhasználni, akkor előfordulhat, hogy azonos elnevezésű elemeket tartalmaznak ezek a dokumentumok, így névütközés keletkezik. A legegyszerűbb megoldás az elemek átnevezése lenne, de ez igen nagy munkát igényelne. Az XML egy olyan lehetőséget biztosít a probléma megoldására, amely ennél jóval hatékonyabb. Ez pedig a névterek használata. Az elemek megkülönböztetésének alapja az, hogy melyik névtérbe tartoznak.

Ahhoz, hogy névteret használhassunk, definiálni kell annak az elemnek a nyitó címkéjében, amelyen belül majd használni kívánjuk. A névtér-deklaráció általános formája: `xmlns:prefix="URI"`. A deklaráció megadható természetesen a gyökérelembe is. A névterek azonosítására URI-k [H15] használhatók. Ezek biztosítják a névterek egyediségét. A névtér előtag – amely lényegében a névtér számára megadott URI rövidítéseként is felfogható – használható az egyes elemek minősítésére. A minősítésre használt karakter a kettőspont (:). A minősítés a következőképpen történik: `előtag:név`. Ezzel jelezhetjük, hogy az adott elem a megadott előtag által reprezentált névtérhez tartozik. Egy adott névtér a deklarációját tartalmazó elem által közrezárt elemek esetében használható, amely elemek a névtér-deklaráció hatáskörét képezik.

Tovább egyszerűsíthető a munkánk alapértelmezett névtér megadásával (`xmlns="URI"`). Ha megadunk egy ilyet egy elem kezdőrészében, akkor a benne található előtag nélküli elemek minősítés nélkül is az alapértelmezett névtér részei lesznek.

2.4. MathML

A `MathML` [I4] egy W3C ajánlás, amely egy XML alapú, a matematikai formulák leírását lehetővé tevő szabványos nyelvet definiál. Célja egy hordozható, nyílt formátum kialakítása a matematikai tartalom tárolására és megjelenítésére. Mind az ember, mind a számítógépek

által egyszerűbbé teszi a természettudományos képletek, formulák, jelzések elektronikus feldolgozását és nagyközönség számára publikálását, mivel egy szabványos, közös felületet biztosít számukra.

Kétféle jelölőrendszert definiál.

- A tartalmi jelölésrendszer („Content markup”), mely a tárolt formulák elemei közötti szemantikai kapcsolatok kiemelésére és az elemek matematikai jelentésére fókuszál.
- A megjelenítési jelölésrendszer („Presentation markup”), mely a formulák megjelenítése szempontjából közelíti meg a címkék elnevezését.

A webböngészők többsége külön bővítmény telepítése nélkül is lehetővé teszi a MathML-ben megadott formulák megjelenítését. Éppen ezért az átalakító program ezt használja fel a matematikai elemek tárolására, megjelenítésére.

A projekt szempontjából előnyös, hogy rendelkezésre áll egy ilyen széles körben támogatott, folyamatosan tökéletesített és nyíltan elérhető formátum, így nem szükséges a matematikai elemek reprezentálására egy saját, XML alapú nyelv kifejlesztése.

2.5. XML Schema

Az XML Schema [I9] egy sémadefiníciós nyelv, amelynek alkalmazásával az XML dokumentumok felépítését meghatározó szabályokat hozhatunk létre. Az előírásainkat egy XML alapú nyelvet felhasználva fogalmazhatjuk meg. Definiálhatjuk, hogy hogyan is kell egy, a szerző igényeinek megfelelő XML dokumentumnak kinéznie.

Az XML Schema egy W3C ajánlás, amely ennek köszönhetően széles körű támogatottságot élvez. Többek között a Java API XML feldolgozáshoz segítséget nyújtó része is külső könyvtárak felhasználása nélkül támogatja a vele való munkát. Előnyös tulajdonsága még, hogy XML szintaxist használ, így viszonylag könnyen tanulható. Akár egy XML szerkesztő segítségével is létrehozhatunk sémadefiníciós állományt, amivel az XML dokumentumunkat

egyébként is létrehozzuk.

Az XML Schema segítségével a következő megkötéseket tehetjük a dokumentumra vonatkozóan:

Meghatározhatjuk

- az elemeket és attribútumaikat, amelyek megjelenhetnek a dokumentumban,
- az egyes elemek milyen hierarchikus viszonyban lehetnek egymással,
- az elemek gyermekeinek számát és sorrendjét,
- adott elem tartalmazhat-e egyéb elemeket, szöveget vagy üresnek kell lennie,
- az elemek és attribútumaik által felvehető értékek típusát,
- alapértelmezett- és megadott tartományból származó értékek felvételét az elemek és az attribútumok számára.

Mivel a programon kívül fognak elhelyezkedni az átalakítási folyamathoz szükséges adatokat tartalmazó XML dokumentumok, így a felhasználó is módosíthatja ezeket az állományokat a feldolgozó alkalmazástól függetlenül. Ezért szükséges a program bemenetére érkező dokumentumok érvényességének ellenőrzése. Ez az ellenőrzés azt a célt szolgálja, hogy a program csak a fejlesztő által definiált szabályoknak megfelelő dokumentumokat kezdje felhasználni a működése során, amelyek feldolgozására fel lett készítve. Ezzel a mechanizmussal elkerülhetők a hibás felépítésű dokumentumok feldolgozása közben jelentkező programhibák.

2.6. StAX API

Az XML dokumentumok kezeléséhez a Java szabványos osztálykönyvtárában megtalálható `StAX API` [I6] került felhasználásra. Így nincs szükség külső könyvtárak használatára az XML állományok érvényességének ellenőrzéséhez, azok beolvasásához és kiírásához.

Egyéb lehetőségek XML dokumentumok feldolgozásához:

- A `DOM API` [H2] felépíti a teljes XML dokumentumból származtatható faszerkezetet a memóriába. Habár ez bizonyos esetekben szükséges lehet, ez a funkció nem lesz kihasználva a fejlesztett alkalmazásban. Használata tehát nem optimális erőforrás-kihasználást okozna.
- A `SAX API` [H10] értelmezője folyamatosan halad végig az XML dokumentumon, és amint találkozik egy elemmel, egy, az elem típusának megfelelő eseményt küld a feldolgozó alkalmazásnak. Mindezt addig végzi, amíg végig nem haladt a dokumentum teljes tartalmán. Ennek a megközelítésnek a hátránya, hogy az alkalmazás semmiféle vezérléssel nem rendelkezik a dokumentumon való végighaladás közben a feldolgozás felett. Csak az események halmazának feldolgozása a feladata. Másik problémája, hogy nem tud XML dokumentumot létrehozni.

A `StAX API` további két részre bontható:

- A kurzor API esetén egy mutató, úgynevezett kurzor halad végig a dokumentumon, amely mindig az XML dokumentum egy szintaktikai elemére mutat a feldolgozás során. Az éppen mutatott szintaktikai elemet lehet a megfelelő metódussal beolvasni. A kurzorral csak előre felé lehet a dokumentumban haladni.
- Az iterátor API az XML dokumentumot mint egy eseményhalmazt tekinti. Az alkalmazás az iterátor API-tól kéri le az olvasás során a következő elemhez tartozó eseményt. Az esemény típusától függően lehet a megfelelő feldolgozási műveletet végrehajtani.

Kategóriáját tekintve az XML feldolgozó eszközök között a `StAX API` által biztosított XML elemző az úgynevezett „pull parser”-ek közé tartozik, mivel az adatok feldolgozásában a továbblépést az alkalmazás vezérli saját igényei szerint. További előnyeként tekinthető, hogy gyorsabb értelmezési folyamatot biztosít az alkalmazás számára, valamint nem igényel akkora memóriaterületet a használata, mint a `DOM API`-é.

2.7. parboiled

A nyelvi elemek felismerésére a `parboiled` [I7] elnevezésű nyelvi elemző került felhasználásra, amely egy meghatározott nyelvtan alapján képes egy bemenő karaktersorozatról eldönteni, hogy az megfelel-e a megadott szabályoknak, azaz a megadott bemenetet a definiált szabályok alapján szintaktikailag elemzi. Az egyes felismerési egységekben lehetővé teszi különböző tevékenységek végrehajtását, így végezhető el vele a nyelvek közötti átalakítás is.

A következő előnyös tulajdonságai miatt esett erre a választás:

- Kisméretű osztálykönyvtár.
- Könnyen tanulható, következetes elnevezéseket használ.
- A nyelvet megadó nyelvtani szabályokat Java nyelvi konstrukciókkal lehet definiálni, így nincs szükség másik nyelv megtanulására a használatához.
- Mivel Java nyelvi elemeket használ, így bármely Java nyelvet támogató forráskód színezésére képes szövegszerkesztő támogatja a kulcsszavainak kiemelését. A kódújrászervezés, hibakeresés és egyéb, a programfejlesztési környezetek által támogatott funkció használatához nincs szükség kiegészítő programok vagy beépülő modulok használatára.

2.8. Apache Maven

Az `Apache Maven` [I11] egy projektkezelő eszköz, amely lehetővé teszi a fordításautomatizálást is. Elsősorban Java projektek kezelésére használják, azonban nem kizárólag Java specifikus eszköz. Funkcionalitását gazdagítja a beépülő modulok támogatása. Számos beépülő modul létezik hozzá. Ezek segítségével lehetővé válik például:

- Verziókövetési információk publikálása.
- A projekt számára weboldal generálása.

- Egységteszt eredményeket tartalmazó jelentések készítése.
- Függőségek kezelése.
- Kódolási konvenciók betartásának ellenőrzése.

Egy `Maven` projekt alapvető eleme a POM (Project Object Model), amely lényegében egy XML állomány, ami információkat és konfigurációs bejegyzéseket tartalmaz a projektről. Ilyenek lehetnek például:

- A forrásállományokat tartalmazó könyvtár elérési útja.
- A különböző, a fordításhoz elengedhetetlen külső komponensek, azaz a függőségek adatai.
- A fordítási folyamatra vonatkozó beállítások.

A `Maven` képes a függőségként definiált külső programegységek automatikus megkeresésére és letöltésére a központi-, illetve a felhasználó által megadott elérési információk alapján egyéb tárolókból. Így elegendő csak a program forrását hordozni a számítógépek között, a megfelelő verziójú külső könyvtárak beszerzése a hálózaton keresztül automatikusan végbemegy, még a fordítás előtt.

Az IDE-ktől való függetlenség elérése érdekében a fejlesztett szoftver ezt használja a források lefordításához és a külső programelemek letöltésére a megfelelő tárolókból. A különböző IDE-k (pl. `Eclipse`, `NetBeans` [H6]), megfelelő bővítmények segítségével képesek a `Maven` projektek kezelésére.

2.9. $\LaTeX$

2.9.1. $\TeX$

A $\TeX$ egy szövegszedő programrendszer. Donald E. Knuth fejlesztette ki, miután „A számítógép programozás művészete” [H17] című műve létrehozása közben sok gondja adódott a könyv megfelelő vizuális megjelenítésével és a matematikai elemek beillesztésével. Végeredményként egy olyan programot hozott létre, amellyel nyomdai minőségű dokumentumok hozhatók létre akár a szerző által is használt számítógépes környezetben. A $\TeX$ felhasználásával szinte minden elképzelhető tipográfiai feladat kivitelezhető, igaz nem mindig a legegyszerűbb módon juthatunk el a kívánt megoldáshoz. Az egyes parancsok a legelemibb műveletek végrehajtását teszik lehetővé. Az összetettebb feladatok elvégzésének megkönnyítésére saját parancsokat írhatunk, ezeket újrafelhasználási céllal külön állományokba, úgynevezett makrócsomagokba szervezhetjük. Az első ilyen csomagot maga Knuth állította össze plain $\TeX$ néven, melynek használatával kapcsolatos információkat az ugyancsak általa írt $\TeX$ book-ban [H16] találhatunk.

2.9.2. $\LaTeX$

A $\TeX$ alapjaira építve Leslie L. Lamport létrehozta a $\LaTeX$ [K4, K2][I3] rendszert, amely egyszerűsítette a szedési feladatok végrehajtását, ezért szélesebb körben is elterjedté vált. A műszaki és tudományos területeken „de facto” szabvánnyá alakult az évek során az ilyen jellegű művek létrehozásánál. A $\LaTeX$ egy olyan dokumentumkezelő rendszer, amely saját jelölőnyelvre segítségével a mű szerzőjének kezébe adja a dokumentum vizuális megjelenésének kezelését. Az alapértelmezett stílusok lehetővé teszik, hogy időigényes beállítások nélkül is egy, a kiadványszerkesztésben kevésbé jártas személy is professzionális minőségű kiadványt hozzon létre. Az előre definiált sémák megváltoztatására is lehetőséget biztosít, ami a dokumentum megjelenítésével többlet foglalatosskódó, a tipográfiai műveletekben jártasabb felhasználók

számára is vonzóvá teszi a rendszert. A manapság használatos szövegszerkesztő alkalmazások ugyan a dokumentum vizuális megjelenítése révén közvetlen képet adnak a mű végleges kinézetéről, azonban a tipográfiai feladatokat is ráhárítják a felhasználóra. A megfelelő megjelenésű dokumentum létrehozásának szabályaival a szerzők többsége nincs tisztában, így az elkészült műveket nem biztos, hogy megfelelő minőségben hozzák létre. Ugyanakkor ezen ismeretek megléte számukra nem is követelmény. A $\text{\LaTeX}$ leveszi a szerző válláról az ezzel kapcsolatos problémákat, a szerzőnek így csak a tartalommal kell foglalkoznia. Az előre definiált stílusok alapján felépül a megfelelő kinézetű dokumentum. Mivel azonban a program nem tudja kikövetkeztetni a szerző által megálmodott struktúrát, így a logikai tagoláshoz szükséges a jelölőelemek segítségével megadni például, hogy hol kezdődik egy fejezet, hol akarunk újabb bekezdést indítani.

3. fejezet

XML alapú nyílt dokumentumformátumok

Egy saját XML alapú nyelv kifejlesztése az átalakítási folyamat kimeneteként több hátránnyal is járna. A két legszembevetőbb a közösségi támogatás, illetve a szabványosság hiánya lenne. Persze demonstrációs céllal a program egy saját, végtelenül egyszerű XML leírónyelvet fog kimenetként szolgáltatni. Azonban egy ilyen alkalmazás szélesebb körű elterjedésének – azaz sikerének – alapja egy már szabványosított, az ipar és a közösség által is elfogadott, valamint számos egyéb alkalmazás által támogatott kimeneti formátumban rejlik. Mivel már születtek ilyen jellegű megoldások, melyek sikeresen alkalmazzák az XML nyelvet dokumentumok tárolási formátumaként, érdemes lehet ezekkel foglalkozni. Az alábbiakban ilyen lehetséges XML alapú célformátumok kerülnek bemutatásra.

3.1. Open Document Format

Az „Open Document Format for Office Applications” (rövidítve ODF [I5]) formátumot a Sun Microsystems fejlesztette ki, de végül az OASIS [H7] által került szabványosításra.

Ez egy nyílt, XML alapú formátum, amely leginkább formázott szöveges dokumentumok, táblázatok és prezentációk tárolására alkalmas. Többek között az `OpenOffice.org` irodai programcsomagban található szoftverek is ezt a formátumot használják a dokumentumok tárolására. 4 különböző XML állományban tárolja elkülönítve az adattartalmat, a stílus- és alkalmazásbeállításokat és a metaadatokat. Ezeket ZIP tömörítési eljárással foglalja egy egységbe – archiválja – valamint a veszteségmentes tömörítésből fakadóan helytakarékoság és az egyszerűbb mozgathatóság végett az eredetnél kisebb méretre tömöríti. Ezt a formátumot rendkívül sok irodai szoftver használja alapértelmezettként és sok támogatja a saját formátuma mellett. Ilyenek például a `Corel WordPerfect Office`, `OpenOffice`, `LibreOffice`, `IBM Lotus Symphony`, `KOffice`, `Google Docs` és még a `Microsoft Office` is. Több mint 600 vállalat tagja az „Open Document Format Alliance”-nak [H8] és a NATO tagállamai számára is ezt a formátumot írja elő a dokumentumok tárolására.

3.2. DocBook

A `DocBook` [I1] egy XML alapú nyelv, amelyet eredetileg a hardverek és szoftverek dokumentálásához fejlesztettek ki, de bármilyen területen felhasználható és az utóbbi években rendkívüli népszerűsége tett szert. Lehetővé teszi, hogy a felhasználó a végső megjelenítéstől függetlenül tudja létrehozni a dokumentumot, kizárólag a logikai felépítésre alapozva. A feldolgozó szoftver feladata a vizuális megjelenés kidolgozása. A dokumentum rendkívül sok formátumba publikálható – ilyenek például a `HTML`, `XHTML`, `PDF`, `Unix man` oldalak – anélkül, hogy a felhasználónak bármit módosítania kellene a dokumentumon. A nyelv maga `RelaxNG` sémával van definiálva, de a séma elérhető még `XSD` és `DTD` formátumban is.

4. fejezet

Nyelvek közötti átalakítás

A számítógép általi feldolgozásra kifejlesztett nyelvek közötti átalakítás [K3] folyamata viszonylag egyszerűnek mondható. A gépi felhasználás alapja, hogy az adott nyelv viszonylag egyszerűen értelmezhető legyen. A nyelvi elemek és a közöttük lévő kapcsolatok valamilyen absztrakt modellel leírhatók. Ez a modell valamilyen formális nyelvet takar. Az értelmezéshez szükséges az egyes nyelvi elemek és elrendeződésük szabályainak megadása egy nyelvtan segítségével. A nyelvtan az adott nyelvre jellemző szabályrendszert írja le, amely alapján feldolgozhatóvá válnak a gép számára is a nyelvhez tartozó elemek, a bemenetként érkező szavak, mondatok. A fejezet az átalakítási folyamattal foglalkozik.

4.1. Szintaktikai elemzés

A nyelvek közötti átalakítás első lépéseként a forrásnyelvet kell felismernünk. A felismeréshez az azt alkotó szintaktikai elemeket kell megtalálnunk. Ehhez nyújtanak segítséget a szintaktikai elemző rendszerek. Egy ilyen rendszert többféleképpen is megvalósíthatunk. A két legelterjedtebb módszer a reguláris kifejezések, illetve a szintaktikai elemzők használata.

4.1.1. Reguláris kifejezések

A reguláris kifejezésekkel [H9] egy megadott szabályrendszer szerint felépített mintát hasonlíthatunk a szöveg elemeihez. A feldolgozó által implementált keresési algoritmus ezen minta alapján dönti el, hogy adott szövegelem megfelel-e a kifejezés által leírt szabályoknak.

Előnyei:

- Könnyedén és gyorsan felépíthetők a megfelelő kifejezésminták.
- Az egyszerűbb konstrukciók felismertetésére tökéletesen megfelel, gyorsan jó megoldást kaphatunk a problémára.

Hátrányai:

- Az összetettebb, illetve nagyobb mennyiségű elem felismertetésére létrehozott minták áttekinthetősége és karbantarthatósága egy idő után kétségessé válik.
- Nincsenek hibakeresési lehetőségek (lényegében a felhasználónak kell a kimenetből kikövetkeztetni, hogy mi lehet a gond).
- A kifejezőerejük nem megfelelő minden konstrukció elemzésére (például: egymásba ágyazott elemek felismerése problémát okozhat).

4.1.2. Szintaktikai elemzők

A szintaktikai elemzők is mintaillesztéssel oldják meg a felhasználó által felépített szabályrendszernek megfelelő nyelvi elemek felkutatását, azonban a minta megadása eltér a reguláris kifejezéseknél megszokottól. Ezek a rendszerek valamilyen magasabb szintű lehetőséget biztosítanak a nyelv felismeréséhez szükséges, a nyelvet megadó nyelvtan leírására. A felhasználó sokkal összetettebb utasítások segítségével, jóval egységesebb és átláthatóbb formában adhatja meg a felismertetési mintát. Az összetettebb feladatok megoldásánál egyértelműen ez a megoldás élvez prioritást.

A szintaktikai elemzők lehetőséget biztosítanak az egyes felismerési egységeknél műveletek végrehajtására. Ezek segítségével megvalósítható, hogy a program bemenetként kapott szöveget ne csak az alapján tudjuk bírálni, hogy megfelel-e a meghatározott nyelvtani szabályoknak. Lehetővé válik különböző tevékenységek végrehajtása a felismert elemekkel az elemzés bizonyos pontjain. Ilyen tevékenység lehet például egy szintaxisfa csomópontjainak felépítése a megfelelő mennyiségű információ begyűjtését követően. A következőkben ismertetett átalakítási eljárások is ezt a képességet használják ki az elemző rendszereknek.

4.2. Átalakítási módszerek

A következőkben a leginkább használt módszerek kerülnek bemutatásra a nyelvek közötti átalakítás területéről.

4.2.1. Szintaxis-vezérelt módszer

A szintaxis-vezérelt megközelítésben a bemenetként kapott forrásnyelvet közvetlenül a célnyelvre alakítjuk, csupán a szintaktikai elemek ismeretében. Amint megtörténik egy ilyen elem felismerése, rögtön elvégezzük a nyelvtani szabályban megadott tevékenység segítségével az átalakítást. Ha a forrás- és célnyelvek nagyon hasonló felépítésűek és viszonylag egyszerűek, ez a módszer igen könnyedén implementálható és gyors megoldást biztosít. Azonban nem minden esetben elégséges, mivel elesünk a szemantikai elemzés lehetőségétől, ami az összetettebb nyelvek esetén elengedhetetlen. A visszafelé hivatkozások, illetve az egyes szövegelemek közötti kapcsolatok nem deríthetők fel, mivel az elemző csak egyszeri alkalommal halad át a bemeneten.

4.2.2. Szabályalapú módszer

Egyfajta továbblépést jelent a szabályalapú megközelítés. Ebben az esetben a bemenet egyes konstrukcióihoz egy megfelelő kimenetet kell hozzárendelnünk. Ezt általában nem általános célú programozási nyelven valósítjuk meg, hanem külön erre a célra kifejlesztett eszközökkel (pl. Stratego/XT [H13], ANTLRMorph [H1]). Az értelmező felépít a megadott bemeneti szabályok alapján egy értelmezési fát, és az egyes részfákra próbálja illeszteni az összerendeléseinkben szereplő szabályokat. Ha illeszkedést talál, akkor az összerendelésben szereplő kimenetet adja eredményül az adott szabály esetében.

4.2.3. Modellvezérelt átalakítási módszer

A modellvezérelt módszer esetében egy bemeneti modellen hajtunk végre elemzést, nem magán a bemeneten. Egy köztes reprezentációját hozzuk létre a bemenetként kapott szövegnek. Ez a reprezentáció egy absztrakt szintaxisfa építésével áll elő. A fa építését a szabályokban elhelyezett tevékenységek segítségével kivitelezhetjük. Ha elegendő információhoz jutottunk a fa egy megadott csomópontjának létrehozásával kapcsolatban, akkor létrehozuk azt, így építve a modellbe az egyes nyelvi elemeket. Ezt a fát később az átalakításhoz szükséges feldolgozási fázisban legalább egyszer bejárjuk. A bejárása közben hozzuk létre a célnyelvű kimenetet a legegyszerűbb esetben. Az összetettebb nyelvek esetén szükséges lehet a fa többszöri bejárása a megfelelő szemantikai kapcsolatok felderítéséhez. Még bonyolultabb esetekben a létrejött szintaxisfa átírására is szükség lehet az összegyűjtött információk alapján.

4.3. Köztes forma

Mint azt láthattuk, a modellvezérelt közelítésben van csak lehetőségünk a legösszetettebb feladatok végrehajtására. Éppen ezért ez a módszer a legelterjedtebb az iparban is. Ahhoz, hogy

a módszer alkalmazásához szükséges bemeneti modellt létrehozhassuk, meg kell ismerkedni az alapjául szolgáló adatszerkezettel.

4.3.1. Fa adatszerkezet

A fa adatszerkezet hierarchikus struktúrát definiál. A fát alkotó elemek a csomópontok. Az egyes csomópontok egymáshoz való hierarchikus kapcsolatuk alapján lehetnek gyermek, illetve szülő szerepkörűek. A szülő-gyermek viszony mindig relatív, azaz az éppen vizsgált csomópontok hierarchiában egymáshoz képest elfoglalt helye alapján állapítható meg. Speciális csomópontot definiál a hierarchia csúcsán álló, úgynevezett „gyökér” elem, melynek nincsen szülője. Különleges tulajdonsággal bírnak még a „levél” elemek, melyeknek nincsenek gyermekei. Ezek az elemek a hierarchia legalján találhatók. Az összes további csomópont rendelkezik pontosan egy szülővel, illetve legalább egy gyermekkel. A fa bármely csomópontját és az alatta elhelyezkedőket együttesen tekintve beszélhetünk részfáról.

4.3.2. Értelmezési fa

Az értelmezési fa – mint azt neve is mutatja – egy fa adatszerkezetű reprezentációja a szintaktikai elemzés során, a megadott nyelvtan alapján felismert szövegelemeknek. A külön elemzési célokra kifejlesztett elemző eszközök automatikusan is képesek ezt előállítani, ahogy haladnak végig a bemeneten. Elemei a szabályok alapján felépülő csomópontok, melyek az adott szabály alkalmazására vonatkozó információkat is tartalmaznak. A fa levélelemei fogják a számunkra is hasznosítható információforrást nyújtani, ugyanis ezek tartalmazzák a felismerési folyamat eredményeként előálló szövegelemeket.

A létrejött fa azonban túlságosan terjengős információforrást biztosít és leginkább a szintaktikai elemzés módjáról tudhatunk meg információkat köztes csomópontjai alapján. További feldolgozásra így ez a szerkezet nem túlságosan hatékony reprezentációs technika. Viszont a szintaktikai elemek felismerésének folyamatáról sok hasznos információt tartalmaz,

így hasznosítható például a megadott szabályok által definiált nyelvtan helyességének ellenőrzésére, illetve hibakeresésre az elemzési folyamat ezen szakaszára vonatkozóan.

4.3.3. Absztrakt szintaxisfa

A forrásnyelv elemzésének további szakaszaiban (például a szemantikai kapcsolatok elemzésének fázisában) hasznosabb lenne egy olyan konstrukció, amely csak a feldolgozás szempontjából lényeges információkat tartalmazza. Erre a igényre nyújt megoldást az absztrakt szintaxisfa. Ez az előzőekben ismertetett elemzési fától abban különbözik, hogy egy jóval magasabb szintű absztrakciót valósít meg a szövegelemeken. Ez abban valósul meg, hogy eredményeképp csupán az adott elemeket és a közöttük lévő szemantikai kapcsolatokat, valamint a további feldolgozásukhoz nélkülözhetetlen információkat tartalmazó csomópontokból épül fel. A feldolgozási lépések következő fázisaiban így jóval egyszerűbbé válik a fával való munka.

Egy absztrakt szintaxisfa tervezésénél a következőket kell figyelembe vennünk:

- Lehetőleg ne tartalmazzon felesleges csomópontokat.
- Könnyedén bejárható legyen. A terv alapján történő felépítés eredményéül egy viszonylag egyszerűen értelmezhető, következetes forma álljon elő.
- A csomópontok a nyelvi elemeket és a közöttük lévő kapcsolatokat reprezentálják.

4.4. A szintaxisfa bejárása

Ahhoz, hogy a felépített absztrakt szintaxisfa által tárolt adatokat fel tudjuk dolgozni, végig kell mennünk minden egyes csomóponton, azaz be kell járnunk a fát. Ennek a műveletnek a megvalósítására az évek során kialakultak jól használható módszerek, melyek mint legjobb gyakorlatok tervezési mintává váltak. Ilyen, az egyben legismertebb bejárást megvalósító

programozási minta a „látogató” tervezési minta. Mivel ennek több ízben is kipróbált, illetve tökéletesített módszerek szolgálnak alapjául, így erősen javallott felhasználása ilyen jellegű feladatok megoldásánál.

4.4.1. Látogató tervezési minta

A látogató tervezési minta [K1][I8] egy viselkedési objektumminta. Hierarchikus adatszerkezetek elemeinek feldolgozása esetén használatos. Ilyen az absztrakt szintaxisfa is, amelyet a fejlesztett alkalmazás a dokumentumformátumok közötti fordítási folyamat során fog felhasználni. Segítségével a fában szereplő csomópontokon kívül helyezhetjük el a rajtuk elvégzendő műveletek kódját. Ráadásul, ha többször is be kívánjuk járni az adatszerkezetet különböző műveletcsoportok végrehajtásának céljából, az ezeket végrehajtó kódokat is külső osztályokba helyezhetjük.

Ha behelyeznénk a csomópontok kódjába a rajtuk végrehajtandó műveleteket, az több hátránnyal is járna. Ilyen például a kód átláthatatlanná válása. Említhetnénk még azt az esetet, ha többféle feldolgozási célhoz tartozó műveletet is elhelyezünk ugyanabban a csomópontban, akkor a megfelelő végrehajtásának kiválasztásához szükséges elágaztatás is többletterhelést jelentene a feldolgozás során, így lassítva a végrehajtást.

A kitűzött céljaink tehát a következők:

- A bejárési műveletek a csomópontokat leíró osztályok kódjától függetlenek legyenek.
- A különböző célú feldolgozási műveletek egymástól elkülönítve kerülhessenek definiálásra.

A látogató minta ezekre az igényekre nyújt megoldást. Az azonos feldolgozási célt szolgáló műveleteket egy külső osztályba szervezzük. Ez az osztály lesz az úgynevezett látogató („Visitor”). Az egyes csomópontokban szerepeltetni kell egy fogadó („accept”) metódust, amely majd a látogató osztályból példányosított objektum „látogatására” reagál. Maga a látogatás úgy zajlik, hogy a látogató objektum referenciája átadásra kerül az adott csomópont

fogadó metódusának. A fogadó metódus meghívja saját referenciájával („this”) a megkapott látogató látogat („visit”) metódusát. Ez a metódus túl van terhelve az adatszerkezet által tartalmazott csomópontok típusainak megfelelő paraméterrel. A futtató környezet a dinamikus kötési mechanizmus segítségével kiválasztja a megfelelő metódust a látogató osztályból. Ezt követően végrehajtja az abban szereplő programkódot, amely természetesen felhasználhatja a paraméterül kapott csomópont objektumnak a látogató számára is elérhető adatait.

Ez a minta egy úgynevezett kettős közvetítési („double dispatch”) mechanizmust használ. Az egyszeres közvetítés esetén azt, hogy melyik művelet teljesít egy adott kérést a kérelem fajtája és a kérelmet fogadó elem típusa határozza meg. A kettős közvetítés jelen esetben azt jelenti, hogy a fogadás végrehajtó metódus jelentése két típustól is függ. Az egyik a látogató típusa, a másik pedig a csomóponté. Így valósulhat meg az, hogy a különböző látogatók, különböző műveleteket végezhetnek a csomópontokon azok típusától függően.

4.4.2. Látogatás külső osztályból

A fa bejárása során a csomópontok által tárolt adatok feldolgozásához szükséges minden kódot a csomópontokon kívül, egy külön osztályban implementáljuk.

Kétféleképpen is megvalósítható:

- Létrehozunk egy túlterhelt metódus sorozatot és ezt hívogatjuk a bejárás során. A megfelelő kód kiválasztását a futtatórendszerre bízunk, amely a túlterhelt metódusok végrehajtása között az objektumok típusa alapján dönt. Létrehozunk egy interfészt, ami tartalmazza az összes implementálandó túlterhelt metódus specifikációját. Ezt az interfészt implementálva hozzuk létre a bejáró osztályunkat. Így megakadályozhatjuk, hogy esetleg mi vagy más fejlesztők kihagyják valamely csomópont bejárásához szükséges metódus megvalósítását.
- Az egyes csomópontok esetében eltároljuk azok típusának azonosítóját. A feldolgozó metódusok között egy, a típusként eltárolt érték alapján választunk. Előny, hogy bármi

lehet az adott típusú csomópont feldolgozását megvalósító metódus neve. Hátrány, hogy elveszítjük a fordítási idejű ellenőrzés lehetőségét, ha esetleg egy metódus nem kerül implementálásra.

Előnyei:

- A fa bejárásához szükséges összes kód egyetlen osztályban összpontosul.
- A feldolgozási utasításokat a csomópontok forrásától függetlenül lehet módosítani.
- Más funkcionalitású feldolgozást megvalósító osztályt könnyebb implementálni és a projektbe építeni.
- Mind homogén, mind heterogén csomópontokból felépülő fákra alkalmazható.

5. fejezet

A fejlesztett alkalmazás

A szakdolgozat kereteiben kifejlesztett alkalmazás tehát a $\text{\LaTeX}$ dokumentumok XML formátumba alakításának problémájára próbál egyfajta megoldást nyújtani. Ennek a műveletnek a kivitelezéséhez először a következőket kell megtennünk:

1. A nyelvi elemek felismeréséhez egy megfelelő nyelvtant kell konstruálnunk.
2. A nyelvtan felhasználásával építenünk kell egy megfelelő köztes reprezentációt a dokumentum számára, amely segítségünkre lesz az összetettebb szerkezetek feldolgozásához.
3. A felépített köztes reprezentáció az esetek többségében – valamint az aktuális megoldásban is – egy absztrakt szintaxisfa. Ennek bejárásával fel kell dolgoznunk az azt alkotó csomópontokat, és az így gyűjtött információk segítségével a bemeneti $\text{\LaTeX}$ dokumentumnak – valamilyen logika szerint – megfelelő XML dokumentumot kell létrehoznunk.

A következőkben ezeknek a műveleteknek a jelen programban történő megvalósításáról lesz szó.

5.1. Követelmények

A legelső lépés a követelmények felderítése. A követelmények jelen esetben azoknak a dokumentum létrehozásához alapvetően szükséges nyelvi építőelemeknek a meghatározását jelentik, amelyeket a felhasználók jelentős többsége alkalmazni szokott a $\text{\LaTeX}$ -kel való munkája során. Ezeket még a tervezési fázist megelőzően kell kiválasztanunk és pontokba foglalni, hogy a fejlesztés során egyértelmű képünk legyen az elvárt funkciókról. Ehhez meg kell ismerkednünk a $\text{\LaTeX}$ nyelvi konstrukcióival és több különböző ezzel kapcsolatos forrásra támaszkodva kell kutatásokat végeznünk, hogy a különféle nézőpontok megismerésével egyértelműen ki tudjuk választani a megfelelő követelményeket. Ezek összegyűjtését követően megfogalmazhatjuk az alkalmazással szemben támasztott igényeinket.

A kifejlesztett program, így a következő nyelvi konstrukciók támogatását valósítja meg:

- Különböző vezérlési elemeket, parancsokat
- A parancsokhoz tartozó opcionálisan megadható paramétereket
- A parancsokhoz tartozó kötelezően megadandó paramétereket
- Az úgynevezett deklarációs utasításokat
- A csoportosított elemeket, melyek névtelen környezetet definiálnak
- A környezeteket
- A normál nyelvi környezetben található környezeteket amelyek különleges feldolgozási eljárást igényelnek, úgy mint például:
 - Táblázatok
 - Tömbök
 - Listák, egymásba ágyazott listák
- „Whitespace” karakterek megfelelő kezelését

- Speciális karaktereket
- Az ékezetes karaktereket
- Matematikai elemeket
- Megjegyzéseket
- A dokumentum logikai tagolását lehetővé tevő parancsokat

5.2. Használat

A felhasználók számára a feldolgozást végző logika teljesen rejtve marad. Az alkalmazás működtetése parancssori felületen történik a következő szintaxis szerint:

```
<programnev> [-d] <bemenet> [<kimenet>]
```

A `programnev` természetesen a program nevét takarja. A `-d` kapcsoló segítségével egyfajta hibakeresési módba állíthatjuk az alkalmazást, amely ilyenkor nem készít kimeneti állományt, csupán a képernyőre írja a dokumentum feldolgozása során felépített absztrakt szintaxisfa egyfajta szöveges reprezentációját. Ennek a kapcsolónak a megadása nem kötelező. A következő argumentum a `bemenet`, amelynek a feldolgozási művelet forrásául szolgáló $\text{\LaTeX}$ dokumentum elérési útját kell tartalmaznia. Ezt követően megadható egy kimeneti állománynév, amelynek megadása csupán opcionális. Ha nem adjuk meg a kimenet nevét, akkor a bemeneti állomány kiterjesztés nélküli nevével lesz azonos (`.xml` kiterjesztéssel). A feldolgozási művelet kimenete minden esetben a program futtatható állománya mellett elhelyezkedő `./output/` mappában fog megjelenni.

5.3. Felépítés

A következőkben azokról az osztályokról történik rövid ismertetés, amelyek valamilyen műveletet végeznek. Így az interfészek, a csomópontokat leíró osztályok és csomópont típusokat tartalmazó felsorolós típusok itt nem kerülnek említésre.

Main: Ez az osztály szolgál a program belépési pontjaként, azaz tartalmazza a

```
public static void main(String[] args)
```

metódust. Ezen felül itt történik:

- a program parancssori argumentumainak feldolgozása,
- a külső XML dokumentumok érvényesség-ellenőrzésének kezdeményezése,
- az XML állományokból az adatok betöltésének kérése,
- a bemeneti állomány beolvasása,
- a beolvasott adattartalom átadása az értelmezőnek elemzésre,
- a megfelelő látogató osztály létrehozása és `translate(ASTNode node)` metódusának meghívása, ami a bemenetként szolgáló dokumentum alapján felépített absztrakt szintaxisfa gyökerét várja paraméterként.

DataLoader: Ennek az osztálynak a metódusai segítségével lehet az XML dokumentumokat beolvasatni és érvényességüket ellenőrizni. Metódusai statikusak, így bárhonnan elérhetőek és a példányosítására nincs szükség.

DataProvider: Az „egyke” tervezési minta alapján nincs nyilvános konstruktora, a `getInstance()` metódusa segítségével kérhető egy belőle származó objektum, így a program bármely más osztályában elérhetőek az általa szolgáltatott adatok. Különböző, a program működése során felhasznált konstansokat és az XML adatforrásokból származó adatokból felépített adatszerkezeteket tárol. Ilyenek például azokból a nyelvi elemekből képzett csomópontok elnevezései, amelyekhez nincs és nem is szükséges a külső állományokból

származó adat. Olyan metódusokat tartalmaz amelyek lehetővé teszik a különböző parancsok XML dokumentumokból kinyert adatainak lekérdezését a fa építése, illetve a célformátumba alakítás folyamán.

LaTeXGrammar: Ez az osztály tartalmazza az értelmező által felhasznált nyelvtani szabályok leírását. Mivel a korábbiakban említett `parboiled` programkönyvtár lett felhasználva az értelmező generálásához, így ez az osztály szabványos Java nyelvi konstrukciókból (metódusok) épül fel. A nyelvtani szabályok és az osztály tartalmának részletesebb leírása megtalálható a „Nyelvtani elemek” szekcióban.

ASTBuilder: Ebben az osztályban zajlik az absztrakt szintaxisfa (AST – „Abstract Syntax Tree”) felépítése. Itt találhatóak meg azok a metódusok, amelyek az értelmezés során meghívásra kerülnek az egyes nyelvtani elemek felismerése során (az értelmező szemszögéből ezek a „tevékenységek”). Ez az osztály a legösszetettebb az egész programban, ebből következően a legtöbb feladat elvégzése is itt zajlik. A későbbiekben („A szintaxisfa felépítése” részben) ennek részletesebb elemzése következik, mivel a segítségével végrehajtott feldolgozási műveletek alkotják az átalakítási folyamat – így az alkalmazás – központi részét.

XMLTranslatorVisitor: Itt zajlik az előző osztályban felépített szintaxisfa feldolgozása, tehát az azt felépítő csomópontok bejárása. Továbbá itt történik a kimenetet alkotó XML dokumentum létrehozása. A fa gyökerét kell egyetlen nyilvános metódusának átadni a feldolgozás megkezdéséhez. Ennek az osztálynak részletesebb kifejtése megtalálható lesz a „Látogatók” résznél.

TreePrinterVisitor: A használatot leíró részben említett `-d` kapcsoló használata esetén ez az osztály kerül példányosításra. Lényegében csak annyit csinál, hogy kiírja a felépített szintaxisfa szerkezetét.

5.4. Külső adatforrások

Külső adatforrásokként XML dokumentumokat használ a program. Ezek a `./res/data/` könyvtárszerkezet alatt helyezkednek el. A pont (.) azt a könyvtárat jelöli, amelyben az alkalmazás is megtalálható.

A következő XML állományokat tartalmazza:

- A `commands.xml` tárolja a nyelvtani elemek feldolgozásához szükséges információkat a parancsokkal kapcsolatban. Ilyen információk a következők:
 - a parancs neve,
 - a parancshoz kapcsolódó argumentumokat leíró minta (ebben *m* jelöli a kötelező- és *o* az opcionálisan megadható argumentumokat),
 - a nyelvi elem típusa (környezet, parancs, lista stb.).
- A `commandsSchema.xsd` az előző állományhoz kapcsolódó sémadefiníciós adatokat tartalmazza. A dokumentum szerkezetére vonatkozó előírásokat hordoz. A dokumentum érvényességét ellenőrző fázisban van rá szükség.
- A `tagnames.xml` a parancsnevekhez tartozó XML címkeneveket tárolja. A köztes forma feldolgozása során az XML-be való átalakítási fázisban szükséges a használata.
- A `tagnamesSchema.xsd` az előző sémadefiníciós állományhoz hasonló célt szolgál, a címkeneveket tartalmazó dokumentumra vonatkozóan.

5.5. Nyelvtani elemek

A $\text{\LaTeX}$ nyelvi elemeinek egy értelmezővel való felismertetését lehetővé tevő nyelvtani szerkezet kialakítása nem egyszerű feladat. Az Interneten való kutatás során sajnos nem sikerült a nyelv szerkezetét általánosan leíró formális nyelvtant találni. Sőt mi több, egyes helyeken

egyenesen lehetetlennek tartják ennek elkészítését a nyelv szabad bővíthetősége és a nyelvtani szerkezeteinek módosíthatóságából kifolyólag. Egyedül maga a $\text{T}_{\text{E}}\text{X}$ rendszer képes teljes mértékben – minden funkcionalitást és módosítást figyelembe véve – a $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ illetve a többi $\text{T}_{\text{E}}\text{X}$ -re épülő megoldás teljes értékű feldolgozására.

Ebből kifolyólag az alkalmazás megvalósítása érdekében saját nyelvtani rendszert kellett kidolgozni a nyelv egyes elemeinek leírásához. Ez a követelményeknél említett módon, a $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ dokumentumok alapvető felépítésének tanulmányozása során tett felismerések felhasználásával lett végrehajtva.

A nyelvtani elemek felismerését végző értelmező generálásához szükséges nyelvtan a `parboiled` osztálykönyvtár által kínált eszközrendszer felhasználásával lett megadva. Az általa kínált metódusok neve igen beszédes, így funkciójukra már ránézésre is következtetni lehet.

Az átalakítás szempontjából alapvető funkció a tevékenységek végrehajtása. A tevékenységeket többféleképpen meg lehet adni, esetünkben az `ACTION()` metódusban való szerepeltetés lett kiválasztva. Ezzel a megadási móddal a nyelvtani szabályokat felépítő elemektől jól elkülöníthető módon kerülnek a tevékenységek a kódba. A tevékenységekről érdemes még tudni, hogy a megkötés velük szemben annyi, hogy egy `boolean` értéket szolgáltatassanak visszatérési értéként. Ettől az értéktől függ ugyanis, hogy a végrehajtás folytatódni fog-e a tevékenység elhelyezkedését követő kódrészletekre az adott szabályban. A következőkben ezeket a nyelvtani elemeket tekintjük át.

```
95 Rule WhiteSpace() {  
96     return AnyOf(" \t\n\r\f");  
97 }
```

Ez a szabály – ahogyan az már a nevéből is kitűnik – a „whitespace” karakterek felismerésében játszik szerepet. A felsorolt „whitespace” karakterek valamelyikére illeszkedő szövegegység esetén fog ez a szabály érvényesülni.

```
99 Rule SpecialCharacter() {  
100     return Sequence(AnyOf("&$"),
```



```

21 |                                     Comment(),
22 |                                     SpecialCharacter(),
23 |                                     Text()
24 |                                     )
25 |                                     ),
26 |                                     EOF
27 |                               );
28 | }

```

Az összes lehetségesen előforduló nyelvtani elem felsorolása történik ebben a szabályban. Ez lesz a kezdő szabály, amelyet az értelmező első alkalommal felhasznál a bemenet feldolgozásának megkezdésekor. Figyelnünk kell a lehetséges nyelvtani elemek felsorolási sorrendjére, ugyanis ennek megfelelően teszteli az adott bemeneti egységre a szabályokat az értelmező. Éppen ezért esetleges optimalizáció is elképzelhető a helyes sorrend megválasztásával. Azonban hibát is eredményezhet egymást kizáró szabályok rossz sorrendben való szerepeltetése.

```

30 | Rule ControlSequence() {
31 |     return Sequence('\',
32 |                     Sequence(FirstOf(ControlWord(),
33 |                                     ControlSymbol())
34 |                               ),
35 |                     Optional('*'))
36 |     ),
37 |     ACTION(converter.convertCommand(match()))
38 | );
39 | }

```

A $\text{\LaTeX}$ dokumentumot felépítő nyelvi elemek közül az egyik legfontosabb a vezérlési szerkezetek (makrók) felismerése. Ezek felépítése viszonylag egyszerű. Kétféle ilyen szerkezet létezik:

- A vezérlő-szavak, melyek szintaktikai felépítésére jellemző, hogy „\” karakterrel kezdődnek amit csak betűk követhetnek. Egy ilyen makró azonosítója az első nem betű karakterig tart. Ezeknek a felismerésére a kódban szereplő `ControlWord()` szabály alkalmas.
- A másik makrócsoportot a vezérlő-szimbólumok alkotják. Ezek is egy „\” karakterrel

kezdődnek és egyetlen nem betű karakterből állnak.

```
42 Rule ControlWord() {  
43     return OneOrMore(Letter());  
44 }
```

A vezérlő-szavak felismerését végzi.

```
47 Rule ControlSymbol() {  
48     return NotLetter();  
49 }
```

A vezérlő-szimbólumok felismerésére szolgál.

```
51 Rule Grouping() {  
52     return Sequence('{',  
53         ACTION(converter.convertBlockBegin()),  
54         ZeroOrMore(Optional('#'),  
55             FirstOf(ControlSequence(),  
56                 Grouping(),  
57                 Text())  
58         ),  
59         ),  
60     ACTION(converter.convertBlockEnd()),  
61     '}'  
62 );  
63 }
```

A $\text{\LaTeX}$ nyelvtani rendszerét felépítő elemek közül kiemelhető még a karakterek csoportosítását végző „{” „}” szimbólumpár. A kötelező argumentumok határolására is szolgál. A $\text{\TeX}$ rendszerben az argumentumokat váró parancsok csak egyetlen szöveges egységet olvasnak be. Az ezekkel a karakterekkel körülzárt – csoportosított – elemek egyetlen egységként kezelendők, így egyszerre több nyelvi egység átadása is lehetséges a parancsok számára argumentumként (nyelvi egységnek számítanak még például a makrók és a karakterek is).

A másik szerepe, hogy egyfajta névtelen környezetet képez. Azaz hatásköri egységet alkot bizonyos parancsok számára. Ilyenek a deklarációs utasítások, amelyek hatása nem csak az argumentumként megadott elemekre érvényesül, hanem az őket tartalmazó hatásköri egységen belül változtatják meg a szedési beállításokat. Az ezekkel a jelekkel határolt deklarációs

utasítások hatása tehát csak a nyitó szimbólumtól („{”) a hozzá tartozó záró szimbólumig („}”) terjed.

```
66 Rule Text() {  
67     return Sequence(OneOrMore(Sequence(  
68         TestNot(ForbiddenCharacterInText()),  
69         ANY  
70     )),  
71     ),  
72     ACTION(converter.convertText(match()))  
73 );  
74 }
```

Egyszerű szöveg felismerését végző szabály. A szöveggel szembeni megkötés csupán annyi, hogy nem tartalmazhatja a $\text{\LaTeX}$ számára speciális jelentésű karaktereket.

```
77 Rule Comment() {  
78     return Sequence("%",  
79         ZeroOrMore(TestNot("\n"), ANY),  
80         ACTION(converter.convertComment(match())),  
81         FirstOf("\n", EOF)  
82     );  
83 }
```

A megjegyzések felismerésének feladata is egyszerűen kivitelezhető. A szabály annyi, hogy a „%” speciális karaktertől az adott sor végéig mindent szövegként kell értelmezni és ezt a karaktersorozatot kell átadni további feldolgozásra.

5.6. A szintaxisfa felépítése

A szintaxisfa felépítése az előzőekben ismertetett nyelvtani szerkezetek leírására szolgáló metódusokban meghívott „tevékenységek” segítségével történik meg. Mivel a nyelvtan nagyon primitív nyelvi elemek felismerésére lett kialakítva, ezért szükséges további műveletek és segédváltozók felhasználása is az átalakítást végző `ASTBuilder` osztályban.

A szintaxisfát alkotó csomópontok heterogének lehetnek, azonban viszonylag kevés típus áll

csak rendelkezésre. Ennek oka az, hogy a közös tulajdonságok kiemelésével kialakított öröklési hierarchia alján szereplő konkrét osztályok végül alacsony számú különböző típus kialakítását tették lehetővé. Nem lett volna szerencsés azonos felépítésű, viszont csak elnevezésben különböző osztályokból nagy számú különböző típust létrehozni. Ez a megközelítés sem a bővíthetőséget, sem az átláthatóságot nem szolgálná. Éppen ezért született a döntés, hogy a közös tulajdonságú osztályokba tartozó nyelvi elemek megkülönböztetésének alapjául egy attribútum szolgáljon (ez pedig a `type`).

A fejlesztés során az egyik legnehezebb feladatnak a parancsargumentumok kezelése bizonyult. Mivel a parancsok és argumentumaik kezelése a legalapvetőbb művelet a $\text{\LaTeX}$ dokumentumok feldolgozása szempontjából, már a legelső fázisban meg kellett ezzel a problémával birkózni. Több lehetőség áttekintését követően a következőre esett a választás: A parancsokhoz egy argumentum mintát adhatunk meg a `commands.xml` állományban, amely a parancs által várt argumentumokat sorolja fel. Külön kellett választani a kötelező és opcionális elemeket. Ezeket rendre az „o” és „m” betűkből álló karaktersorozat szemlélteti. Ha sikerült egy argumentum beolvasása, akkor továbblépünk a sorozatban. Ha a sorozat végére értünk és nincs már több beolvasandó argumentum, akkor visszalépünk a fában az adott parancsot szimbolizáló csomópont szülőjébe.

Szükséges volt a környezetek és csoportosított elemek miatt egyfajta mélységérték megadására. Ezek hatáskör korlátozó elemek, tehát ezeken belül érvényesek a parancsargumentumnál nagyobb kiterjedésű hatáskörre vonatkozó parancsok hatásai. Ez a kétfajta elem fog tehát beágyazottsági mélységet megadni. Mivel a csoportosított elemek, illetve a környezetek többszörösen egymásba ágyazhatók, ezért külön problémát jelent annak meghatározása, hogy mikor olvassuk a parancs egy argumentumát és mikor vagyunk a parancs egyazon argumentumában csak többszörösen beágyazott elemen belül. Erre megoldás a mélységértékek felhasználása. A mélységértéket eltároljuk, ha egy olyan parancs érkezik a feldolgozás során, amely argumentumokat várhat. Ha egy ilyen mélységen vagyunk, akkor a parancs argumentumaként beolvassuk az adott csoportosító elemet vagy opcionális argumentumot

és léptetjük a mintában az aktuális elemet jelző mutatót, valamint ennek megfelelően módosítjuk a fát. Ha nem tartalmaz az adott mélység argumentumot váró parancsot, akkor a jelenleg is aktuálisként szereplő csomóponthoz adjuk továbbra is az ezt követően létrehozott csomópontokat.

A környezetek is alapvető nyelvi elemeknek számítanak. Mivel ezek megjelenése megegyezik a parancsokéval, külön nyelvtani szabály létrehozását nem láttam szükségesnek. Viszont ebben az esetben problémát okozhat, hogy a környezeteket reprezentáló csomópontok elnevezésének és esetleges argumentum mintájának megszerzéséhez szükséges a környezet neve. Végül úgy lett kialakítva ezek kezelése, hogy a név beolvasását (ami jelen parancsok esetében az első kötelező argumentumuk) és az argumentum minta meghatározását a parancsokétól eltérő módon kellett megoldani. A szövegbeolvasást kezelő tevékenységhez kapcsolódó metódusban történik mindez. Nem a minta szerint olvassuk a `\begin` és `\end` parancsok (amelyek a környezeteket határolják) esetében az első argumentumot, hanem két `boolean` típusú változó segítségével döntjük el a szövegfeldolgozóban, hogy éppen környezet nevét olvassuk-e. Ha igen, megkeressük az ehhez a névhez tartozó argumentum listát és létrehozunk egy, a környezetekhez kialakított tulajdonságú csomóponttípust és beállítjuk emellett a megfelelő mintát is, ha ezt igényli. Ha elértük az aktuális környezethez kapcsolódó záró elemet akkor viszont a csomópontból kijebb lépésen kívül más dolgunk nincsen mialatt az első argumentumot beolvassuk, mivel azzal már nem tudunk mit kezdeni. A záró résznél már csak jelzés értékű a környezet nevének szerepeltetése, ezért mivel nem tudjuk ezt mi sem felhasználni, figyelmen kívül hagyjuk.

Az értelmezés során hívott tevékenységeket végrehajtó metódusok leírása következik az alábbiakban. Megfigyelhető, hogy minden esetben igaz érték visszaadása történik, hogy ne befolyásoljuk az értelmezés menetét.

```
50 @Override
51 public boolean convertSpecialChararacter(char character) {
52     switch (character) {
53         case '&':
54             if (inTable) {
```

```

55         closeContainer(NodeType.TABLE_CELL);
56         createNewNode(DataProvider.TABLE_CELL_NAME,
57                        NodeType.TABLE_CELL);
58     }
59     break;
60     case '$':
61         if (mathMode) {
62             mathMode = false;
63             closeContainer(NodeType.MATH_EXPRESSION);
64         }
65         else {
66             mathMode = true;
67             createNewNode(DataProvider.MATH_EXP_NAME,
68                          NodeType.MATH_EXPRESSION);
69         }
70     break;
71 }
72
73 return true;
74 }

```

Fentebb látható a speciális jelentésű karakterek kezelését leíró tevékenység. Az alkalmazás jelen állapotában csak a következők kerültek implementálásra: a táblázat celláit elválasztó „&” karakter és a matematikai módba lépésre szolgáló „\$” karakter.

```

76 @Override
77 public boolean convertComment(String text) {
78     createNewTextNode(text, NodeType.COMMENT);
79
80     return true;
81 }

```

A megjegyzések esetében más egyéb teendőnk nincs, mint egy új szöveges csomópont létrehozása megjegyzés típusúként.

```

84 public boolean convertCommand(String name) {
85     if (isArgumentContainerDepth())
86         closeThisDepthArguments();
87
88     if (name.equals(DataProvider.ENV_BEGIN)) {
89         environmentBegin = true;
90         ++depth;
91     }
92     else if (name.equals(DataProvider.ENV_END)) {
93         environmentEnd = true;
94         --depth;

```

```
95     }
96     else if (inTable && name.equals("\\")) {
97         closeContainer(NodeType.TABLE_ROW);
98         createNewTableRow();
99     }
100    else if (name.equals("(") || name.equals "[")) {
101        String mathName = (name.equals("(")
102                           ? DataProvider.TEXT_MATH_NAME
103                           : DataProvider.DISPLAY_MATH_NAME);
104        mathMode = true;
105        createNewNode(mathName, NodeType.MATH_EXPRESSION);
106    }
107    else if (name.equals(")") || name.equals "]")) {
108        mathMode = false;
109        closeContainer(NodeType.MATH_EXPRESSION);
110    }
111    else {
112        createNewNode(name,
113                      dataProvider.getCommandData(name).getType());
114    }
115
116    return true;
117 }
```

A parancsok felismerésekor hívott tevékenység már bonyolultabb működésű. Először is azt ellenőrizzük, hogy az adott szinten van-e argumentumot váró parancs. Ez arra az eset lett elhelyezve, ha a parancs utolsó argumentuma opcionális lenne és nem élünk a megadásának lehetőségével.

Környezet kezdete, illetve vége esetében a korábban említett mélységet változtatjuk. Az `environmentBegin` és `environmentEnd` változók felhasználása a szöveg ellenőrzésekor fog megtörténni, ugyanis a környezet nevének megszerzése után tudjuk majd csak létrehozni a megfelelő csomópontot.

Ha éppen táblázatot építünk, az adott sor lezárását és egy újabb elkezdését a „\” karakter segítségével tudjuk kezdeményezni.

Azt, hogy matematikai módban vagyunk, a megadott határoló karakterek segítségével tudhatjuk meg és az ebbe való belépés, illetve belőle kilépés hatására létrehozhatjuk a megfelelő matematikai kifejezést tároló csomópontot, vagy éppen kiléphetünk abból.

Ha a fentebb felsoroltak közül egyik speciális eset sem érvényesül, akkor a parancs típusának

és nevének megfelelő csomópontot hozunk létre.

```
120 public boolean convertText(String text) {
121     if (environmentBegin) {
122         NodeType nodeType =
123             dataProvider.getCommandData(text).getType();
124
125         if (nodeType == NodeType.DECLARATION)
126             createNewNode(text, NodeType.ENVIRONMENT);
127         else
128             createNewNode(text, nodeType);
129
130         if (nodeType == NodeType.MATH_EXPRESSION) {
131             mathMode = true;
132         }
133         else if (nodeType == NodeType.TABLE) {
134             inTable = true;
135             createNewTableRow();
136         }
137     }
138     else if (environmentEnd) {
139         NodeType nodeType =
140             dataProvider.getCommandData(text).getType();
141
142         if (nodeType == NodeType.DECLARATION)
143             closeContainer(NodeType.ENVIRONMENT);
144         else
145             closeContainer(nodeType);
146
147         if (nodeType == NodeType.MATH_EXPRESSION)
148             mathMode = false;
149         else if (nodeType == NodeType.TABLE)
150             inTable = false;
151     }
152     else if (isArgumentContainerDepth()) {
153         if ( ((CommandNode)actNode).getArgumentsPattern().
154             charAt(actArgumentPointer) == 'o' ) {
155             collectOptionalArgument(text);
156         }
157         else {
158             closeThisDepthArguments();
159             createNewTextNode(text, NodeType.TEXT);
160         }
161     }
162     else if (mathMode) {
163         createMathNodes(text);
164     }
165     else {
166         createNewTextNode(text, NodeType.TEXT);
167     }
168
169     return true;
```

```
170 }
```

Az egyik legösszetettebb feladata a szövegátalakítónak van. Ilyen, hogy a környezet nevének megszerzésekor létrehozzuk a megfelelő csomópontot. Matematikai környezet esetén matematikai módba váltunk. Táblázat esetén létrehozzuk a táblázat első sorát és első celláját. Ha környezet végénél vagyunk, az előzőekkel ellentétes műveleteket hajtunk végre. Ha opcionális argumentumot várunk éppen a parancshoz, meghívjuk a megfelelő metódust, amely ellenőrzi, hogy az opcionális argumentumok feltételének megfelelő-e a kapott szöveg. Matematikai módban kiválogatjuk a szövegből karakterenként a kifejezés megfelelő építőelemeit és csomópontokat képzünk belőlük.

```
173 public boolean convertBlockBegin() {
174     if (environmentBegin || environmentEnd) {
175         ;
176     }
177     else if (isArgumentContainerDepth()) {
178         CommandNode command = (CommandNode)actNode;
179         actArgumentPointer = command.getArgumentsPattern().
180             indexOf('m', actArgumentPointer);
181
182         if (actArgumentPointer != -1) {
183             createNewNode(DataProvider.MAN_ARG_NAME,
184                 NodeType.GROUPING);
185         }
186         else {
187             closeThisDepthArguments();
188             createNewNode(DataProvider.GROUPING_NAME,
189                 NodeType.GROUPING);
190         }
191     }
192     else {
193         createNewNode(DataProvider.GROUPING_NAME,
194             NodeType.GROUPING);
195     }
196
197     ++depth;
198
199     return true;
200 }
```

Ha egy csoportosítás kezdődik, az jelenthet hatáskör képző környezetet, illetve kötelező argumentumot egyaránt. Mint az látható, ha környezet kezdődik vagy fejeződik be – azaz annak

nevét gyűjtjük be éppen –, akkor nem kell tennünk ebben az esetben semmit.

Ha argumentumot váró parancs szintjén érkezik egy ilyen csoportosító elemet nyitó karakter, akkor először is ellenőriznünk kell, hogy nem vagyunk-e opcionális paraméternél a mintában. Ehhez az első nem opcionális paramétert szemléltető „m” betűre ugrunk, ami a jelenlegi pozíciónkhoz képest legközelebb van. Ha találunk még ilyet, akkor létrehozunk egy csoportosító csomópontot, ha viszont nem, akkor az aktuális szintet kirakjuk a parancsokat várók közül. Mélységértéket is módosítunk, ugyanis határoló szerepe van bizonyos parancsok hatásköre szempontjából ennek a nyelvi szerkezetnek.

```
203 public boolean convertBlockEnd() {
204     if (!environmentBegin && !environmentEnd)
205         closeContainer(NodeType.GROUPING);
206
207     --depth;
208
209     if (environmentBegin) {
210         environmentBegin = false;
211     }
212     else if (environmentEnd) {
213         environmentEnd = false;
214     }
215     else if (isArgumentContainerDepth()) {
216         moveArgumentPointer();
217     }
218
219     return true;
220 }
```

A csoportosító elemet záró karakterhez érve ugyanazt a logikát követjük mint a nyitó elemnél, csak minden műveletet az ellentétes módon hajtunk végre. Ha argumentumot igénylő parancshoz tartozik ez a csoportosítás, akkor léptetjük az elvárt argumentumokat szemléltető minta mutatóját.

```
226 private void createMathNodes(String expression) {
227     StringBuilder digitBuffer = new StringBuilder();
228
229     for (int ch = 0; ch < expression.length(); ch++) {
230         char actChar = expression.charAt(ch);
231
232         if (Character.isDigit(actChar) || actChar == '.') {
```

```

233         digitBuffer.append(actChar);
234     }
235     else if (Character.isLetter(actChar)) {
236         if (digitBuffer.length() != 0) {
237             createNewTextNode(digitBuffer.toString(),
238                             NodeType.M_NUMBER);
239             digitBuffer.delete(0, digitBuffer.length());
240         }
241
242         createNewTextNode(Character.toString(actChar),
243                             NodeType.M_IDENTIFIER);
244     }
245     else if (Character.isWhitespace(ch)) {
246         if (digitBuffer.length() != 0) {
247             createNewTextNode(digitBuffer.toString(),
248                             NodeType.M_NUMBER);
249             digitBuffer.delete(0, digitBuffer.length());
250         }
251
252         continue;
253     }
254     else {
255         if (digitBuffer.length() != 0) {
256             createNewTextNode(digitBuffer.toString(),
257                             NodeType.M_NUMBER);
258             digitBuffer.delete(0, digitBuffer.length());
259         }
260
261         if (isOperator(actChar))
262             createNewTextNode(Character.toString(actChar),
263                             NodeType.M_OPERATOR);
264     }
265 }
266 }

```

A matematikai módban a szöveg három részre tagolható. Az egyik ilyen a számok csoportja. Egy átmeneti tárolóban gyűjtjük a szám számjegyeit, illetve tört szám esetén a tizedes pontot. Ha elérünk egy számjegytől vagy ponttól eltérő karaktert, akkor kiírjuk a számot és mellé az operátor vagy változó karaktert. Operátor a felsorolásban szereplő karakter lehet. A betűk változókat szimbolizálnak.

```

273 private boolean isOperator(char character) {
274     char[] operators = {'+', '-', '*', '/', '=',
275                         '(', ')', '>', '<', '!'};
276
277     for (int c = 0; c < operators.length; c++) {
278         if (character == operators[c])
279             return true;

```

```
280     }
281
282     return false;
283 }
```

Ha a felsorolt operátorok között megtalálható a paraméterül kapott karakter, akkor igaz értékkel térünk vissza, azaz ez egy operátort reprezentáló karakter.

```
298 private void closeContainer(NodeType type) {
299     NodeType actNodeType = null;
300
301     while (true) {
302         actNodeType = actNode.getType();
303
304         if (actNodeType == type) {
305             actNode = actNode.getParent();
306             break;
307         }
308
309         if (actNodeType == NodeType.ROOT)
310             break;
311
312         actNode = actNode.getParent();
313     }
314 }
```

A metódus célja kilépni egy tároló csomópontból. A paraméterül kapott típusnak megfelelő csomópontig megy fel a fában és még egyet lép, hogy elérje és aktuálissá tegye a lezárandó típusú csomópont szülőjét.

```
322 private void collectOptionalArgument(String text) {
323     String textToAnalyze = text;
324     StringBuilder optionalArgument = new StringBuilder();
325
326     while (true) {
327         int firstNotSpaceIndex = 0;
328
329         for (firstNotSpaceIndex = 0;
330              firstNotSpaceIndex < textToAnalyze.length();
331              firstNotSpaceIndex++) {
332             if (textToAnalyze.charAt(firstNotSpaceIndex) != ' ')
333                 break;
334         }
335
336         String trimmedText =
337             textToAnalyze.substring(firstNotSpaceIndex);
```

```
338     char firstCharacter = trimmedText.charAt(0);
339
340     if (isArgumentContainerDepth() && firstCharacter == '[') {
341         for (int c = 0; c < trimmedText.length(); c++) {
342             char actChar = trimmedText.charAt(c);
343
344             optionalArgument.append(actChar);
345
346             if (actChar == ']') {
347                 int startIndex = firstNotSpaceIndex + c + 1;
348                 if (startIndex >= textToAnalyze.length())
349                     textToAnalyze = "";
350                 else
351                     textToAnalyze =
352                         textToAnalyze.substring(startIndex);
353
354                 createNewNode(DataProvider.OPT_ARG_NAME,
355                             NodeType.OPT_ARG);
356                 createNewTextNode(optionalArgument.toString(),
357                                 NodeType.TEXT);
358                 closeContainer(NodeType.OPT_ARG);
359                 moveArgumentPointer();
360                 optionalArgument.delete(0,
361                                     optionalArgument.length() - 1);
362
363                 break;
364             }
365         }
366     }
367     else {
368         closeThisDepthArguments();
369         createNewTextNode(textToAnalyze, NodeType.TEXT);
370         textToAnalyze = "";
371         break;
372     }
373
374     if (textToAnalyze.isEmpty())
375         break;
376 }
377 }
```

Az opcionális argumentumok összegyűjtése nem könnyű feladat. A különböző csomagok lehetővé teszik, hogy ne csak az első argumentum lehessen opcionális, hanem akárhányadik az argumentumok listájában. A másik nehézséget az jelenti, hogy a határoló karaktereik nem számítanak speciálisnak a $\text{\LaTeX}$ számára, így egyszerű szöveggént foghatók fel. Tehát nem célszerű rájuk nyelvtani szabályt írni. Ebben a metódusban ha az argumentummintában „o” betűnél jár éppen a feldolgozás ellenőrizzük karakterenként a bemeneti szöveget és

megpróbáljuk szétválogatni a „[” és „]” jelek által határolt argumentumokat.

```
408 private void createNewNode(String name, NodeType type) {
409     switch (type) {
410         case COMMAND: case DECLARATION:
411             CommandNode commandFromDatabase = new CommandNode(name);
412
413             commandFromDatabase.setType(type);
414             commandFromDatabase.setParent(actNode);
415             commandFromDatabase.setArgumentsPattern(
416                 dataProvider.getCommandData(name).getArgumentsPattern());
417
418             addToTree(commandFromDatabase);
419
420             break;
421         case GROUPING: case TABLE_ROW: case TABLE_CELL:
422         case OPT_ARG: case MATH_EXPRESSION:
423             ContainerNode builtinContainer = new ContainerNode(name);
424
425             builtinContainer.setType(type);
426             builtinContainer.setParent(actNode);
427
428             addToTree(builtinContainer);
429
430             break;
431         case ENVIRONMENT: case LIST: case LIST_ITEM: case TABLE:
432             ContainerNode containerFromDatabase =
433                 new ContainerNode(name);
434
435             containerFromDatabase.setType(type);
436             if (type == NodeType.LIST_ITEM &&
437                 actNode.getType() == NodeType.LIST_ITEM)
438
439                 closeContainer(NodeType.LIST_ITEM);
440
441             containerFromDatabase.setParent(actNode);
442
443             addToTree(containerFromDatabase);
444
445             String argumentPattern =
446                 dataProvider.getCommandData(name).getArgumentsPattern();
447             if (!argumentPattern.isEmpty()) {
448                 containerFromDatabase.setArgumentsPattern(argumentPattern);
449
450                 int depthOffset = (type == NodeType.LIST_ITEM)?0:-1;
451                 argumentContainerDepths.push(depth + depthOffset);
452
453                 if (actArgumentPointer != -1)
454                     argumentPointers.push(actArgumentPointer);
455
456                 actArgumentPointer = 0;
457             }
```

```
458  
459         break;  
460     }  
461 }
```

A beérkező típusnak megfelelő csomópontot hozunk itt létre. Ha külső forrásból gyűjtjük be az adatait, akkor létrehozásával egy időben ellenőrizzük, hogy vannak-e argumentumai, és ha vannak, akkor beállítjuk hozzá a megfelelő argumentum mintát. Itt dől el az is, hogy tároló vagy egyszerű parancs csomópont keletkezik. A csomópontok létrehozásának folyamata a következő:

1. Létrehozunk egy megfelelő típusú új csomópontot.
2. Beállítjuk a nevét.
3. Beállítjuk az aktuális csomópontot a szülőjeként.
4. Beállítjuk a típusát.
5. Ha van argumentum mintája, azt is beállítjuk.

```
468     if (actNode instanceof ContainerNode) {  
469         ContainerNode container = (ContainerNode)actNode;  
470         if ((container.getType() == NodeType.ENVIRONMENT ||  
471             container.getType() == NodeType.LIST ||  
472             container.getType() == NodeType.TABLE ||  
473             container.getType() == NodeType.LIST_ITEM)  
474             && isArgumentContainerDepth())  
475             container.addArgument(node);  
476         else  
477             container.addChild(node);  
478     }  
479     else if (actNode instanceof CommandNode) {  
480         ((CommandNode)actNode).addArgument(node);  
481     }  
482  
483     if (node instanceof ContainerNode) {  
484         actNode = node;  
485     }  
486     else if (node instanceof CommandNode) {  
487         CommandNode command = (CommandNode)node;  
488  
489         if (!command.getArgumentsPattern().isEmpty()) {  
490             actNode = node;  
491  
492             if (actArgumentPointer != -1)  
493                 argumentPointers.push(actArgumentPointer);
```

```
494 |
495 |         actArgumentPointer = 0;
496 |         argumentContainerDepths.push(depth);
497 |     }
498 | }
499 | }
```

A fához hozzáadás történik itt meg. Esetlegesen argumentumot váró tárolóhoz eldöntjük, hogy argumentumként vagy gyermekként kívánjuk hozzáadni a következő csomópontot. Meghatározzuk, hogy szükséges-e aktuális csomópontként beállítani az újonnan érkezőt.

```
506 | private void createNewTextNode(String text, NodeType type) {
507 |     TextNode textNode = new TextNode(text);
508 |     textNode.setType(type);
509 |     textNode.setParent(actNode);
510 |
511 |     addToTree(textNode);
512 | }
```

Szöveges csomópont létrehozása. Létrehozunk egy neki megfelelő típusú csomópontot és hozzáadjuk a fához.

5.7. Látogatók

A látogatók (a látogató tervezési minta alapján) azok az osztályok, amelyek az absztrakt szintaxisfa bejárását végzik. Minden egyes csomóponton végighaladnak és az „útjuk” során gyűjtött információk alapján valamilyen tevékenység végrehajtása van rájuk bízva a kódjuknak megfelelően. A csomópontok közötti kapcsolatok és az általuk tárolt információk segítségével a programozó által előírt feldolgozási műveletet hajtanak végre a fát alapul véve.

A program jelenlegi verziójának fejlesztése során két látogató osztály került megvalósításra. Az egyik az XML alakítást hajtja végre, míg a másik mind a fejlesztés, mind a felhasználás során hibakeresésre használható. A korábban megadott célok megvalósításához az átalakítást végző osztály (XMLTranslatorVisitor) szolgál alapul, így ennek bemutatására kerül itt sor.

```
250 private String correctText(String text) {
251     StringBuilder correctedText = new StringBuilder();
252     boolean whitespace =
253         (text.charAt(0) == ' ' && endedWithWhitespace) ? true : false;
254
255     for (char ch : text.toCharArray()) {
256         if (ch == '\n' || ch == '\r' || ch == '\f' || ch == '\t')
257             ch = ' ';
258
259         if (ch == ' ') {
260             if (whitespace) {
261                 continue;
262             }
263             else {
264                 whitespace = true;
265             }
266         }
267         else if (whitespace){
268             whitespace = false;
269         }
270
271         switch (ch) {
272             case '<':
273                 correctedText.append("&lt;");
274                 break;
275             case '>':
276                 correctedText.append("&gt;");
277                 break;
278             case '&':
279                 correctedText.append("&amp;");
280                 break;
281             case '\':
282                 correctedText.append("&apos;");
283                 break;
284             case '\"':
285                 correctedText.append("&quot;");
286                 break;
287             default:
288                 correctedText.append(ch);
289                 break;
290         }
291     }
292
293     if (correctedText.charAt(correctedText.length() - 1) == ' ')
294         endedWithWhitespace = true;
295     else
296         endedWithWhitespace = false;
297
298     return correctedText.toString();
299 }
```

Ez a metódus a „whitespace” karakterek és az XML dokumentumban közvetlenül nem

szerepeltethető karakterek kezelését végzi. Mivel a $\text{\LaTeX}$ -ben akárhány szóköz, új sor stb. karakter egyetlen egy szóköz karaktert jelent, így szükséges a többszörös megjelenésük megszüntetése. Az XML dokumentumok szöveges adatai között bizonyos, a nyelvi elemek felépítésénél használt karakterek nem jelenhetnek meg. Ezeknek a karaktereknek a lecserélése történik meg itt. A metódus paraméterként a javítandó szöveget várja, míg a visszatérésekor a megfelelően átalakított szöveget adja vissza.

Minden egyes csomóponttípushoz szükség van egy, a típus tulajdonságainak megfelelően kialakított látogató metódusra. Ezek segítségével válik lehetővé a paraméterül kapott és az általa tartalmazott csomópontok bejárása.

```
183 private void visitTextNode (ASTNode node)
184     throws XMLStreamException {
185
186     TextNode textNode = (TextNode)node;
187     xmlWriter.writeCharacters (correctText (textNode.getText ()));
188 }
```

A szöveges csomópontok bejárása igen egyszerű. Mivel felépítésükből adódóan csak egy típust és magát a szöveges adatot hordozzák. Bejárásuk jelen esetben a szöveges tartalom XML állományba írását jelenti.

```
195 private void visitCommandNode (ASTNode node)
196     throws XMLStreamException {
197
198     CommandNode command = (CommandNode)node;
199
200     if (!command.getArguments().isEmpty()) {
201         xmlWriter.writeStartElement ("args");
202
203         for (ASTNode actArg : command.getArguments()) {
204             translate (actArg);
205         }
206
207         xmlWriter.writeEndElement ();
208     }
209 }
```

A parancsokat reprezentáló csomópontok esetében annyival lesz bonyolultabb a helyzet,

hogy a parancshoz tartozó argumentumokat is tartalmazhatják. Így fel kell dolgoznunk mind az adott csomópontot, mind pedig az argumentumait. Az argumentumok bejárása a `translate(ASTNode node)` metódus hívásával valósul meg. Ha végeztünk az argumentumokkal, akkor végül lezárjuk a parancsot szemléltető XML elemet.

```
216 private void visitContainerNode(ASTNode node)
217     throws XMLStreamException {
218
219     ContainerNode container = (ContainerNode)node;
220
221     if (!container.getArguments().isEmpty()) {
222         xmlWriter.writeStartElement("args");
223
224         for (ASTNode actArg : container.getArguments()) {
225             translate(actArg);
226         }
227
228         xmlWriter.writeEndElement();
229     }
230
231     if (!container.getContent().isEmpty()) {
232         if (!mathMode)
233             xmlWriter.writeStartElement("content");
234
235         for (ASTNode actChild : container.getContent()) {
236             translate(actChild);
237         }
238
239         if (!mathMode)
240             xmlWriter.writeEndElement();
241     }
242 }
```

A tároló típusú csomópontok bejárása annyiban tér el a parancsokat tartalmazóakétól, hogy a gyermekelemeiket is be kell egyesével járnunk a csomópont teljes feldolgozásának megvalósítása érdekében.

Mivel a feldolgozást megvalósító és koordináló `translate` metódus minden egyes csomóponttípust tartalmaz, ezért a következőkben – a helytakarékosság érdekében – csupán a nagyon eltérő funkcionalitású eseteket tekintjük át.

```
42 case ROOT:
43     xmlWriter.writeStartElement("LaTeXDocument");
```

```
44 visitContainerNode(node);  
45 xmlWriter.writeEndElement();  
46 xmlWriter.flush();  
47 break;
```

A szintaxisfa gyökerét reprezentáló csomópont esetében a fa bejárása előtt létrehozuk a gyökérelemet, illetve ha végeztünk, lezárjuk azt.

```
48 case COMMAND:  
49     String commandTag = dataProvider.getTagname(  
50         ((CommandNode)node).getName());  
51     xmlWriter.writeStartElement(commandTag);  
52     visitCommandNode(node);  
53     xmlWriter.writeEndElement();  
54     break;
```

A parancs esetében lekérdezzük a korábban a megfelelő külső adatforrásként használt XML állományból beolvasott parancsnévhez tartozó címkét. Érdeemes megemlíteni, hogy ha nincs a beolvasott adatok között a parancsnévhez tartozó címke, akkor magát a parancsnevet adja vissza és ez lesz a parancshoz tartozó XML elem neve. Itt is az előző esethez hasonlóan írunk egy XML nyitó címkét, majd bejárjuk a parancshoz kapcsolódó csomópontot. Ezt követően lezárjuk a megnyitott címkét.

```
71 case TEXT:  
72     visitTextNode(node);  
73     break;
```

Mint a bejárásnál, itt is könnyű dolgunk van az aktuális implementációban, ugyanis csak be kell járnunk a szöveges csomópontot a megfelelő szöveges adat begyűjtéséért.

```
55 case DECLARATION:  
56     CommandNode declarationNode = (CommandNode)node;  
57     String declarationTag = dataProvider.getTagname(  
58         declarationNode.getName());  
59  
60     if (declarationNode.getArguments().isEmpty()) {  
61         xmlWriter.writeEmptyElement(declarationTag);  
62         xmlWriter.writeAttribute("type", "declaration");  
63     }  
64     else {  
65         xmlWriter.writeStartElement(declarationTag);
```

```
66     xmlWriter.writeAttribute("type", "declaration");
67     visitCommandNode(node);
68     xmlWriter.writeEndElement();
69 }
70 break;
```

A deklarációs utasítás itt egyszerű csomópontként jelenik meg. A hozzá tartozó XML elemnek különlegessége, hogy tartalmazni fog egy `type` elnevezésű attribútumot. Ennek értéke pedig az utasítás típusára utal. A többi ehhez hasonló utasítás is így van megoldva (pl. a logikai struktúrát jelző utasítások). Eltérés az előzőektől, hogy ha nincs az adott utasításhoz argumentum kapcsolva, akkor üreselem címkét hozunk létre.

```
74 case COMMENT:
75     xmlWriter.writeStartElement("comment");
76     visitTextNode(node);
77     xmlWriter.writeEndElement();
78 break;
```

Vannak elemek, amelyek rögzített címkenevet kapnak. Ilyenek például a megjegyzéseken felül a kötelező és opcionális argumentumok, listaelem, tábla sor, tábla cella, változók, műveleti jelek, számok.

```
146 case MATH_EXPRESSION:
147     mathMode = true;
148     xmlWriter.writeStartElement("math");
149     xmlWriter.writeDefaultNamespace(
150         "http://www.w3.org/1998/Math/MathML");
151     visitContainerNode(node);
152     xmlWriter.writeEndElement();
153     mathMode = false;
154     break;
```

A matematikai kifejezések esetében egy boolean típusú változót (`mathMode`) is állítgatnunk kell, hogy egyéb metódusokban tudhassuk, hogy éppen matematikai környezetben tartózkodunk-e. A korábbiakhoz hasonló a bejárás és a kezdő címke írásának kezdeményezése. Az eltérést a `MathML`-hez tartozó névtér jelenti. Ennek kiírása is szükséges a matematikai elemek szerepeltetéséhez.

6. fejezet

Összegzés

Így a dolgozat végére érve összegzésképpen kijelenthetjük, hogy sikerült a korábban kitűzött célokat megvalósítani.

Megismerkedhettünk a dokumentumformátumok, illetve különböző nyelvek közötti átalakításnak az évek során kialakított leghatékonyabb módszereivel. A különböző technológiákkal, amik a program fejlesztése során felhasználásra kerültek, és mindezek közben egy kicsit magával a $\text{\LaTeX}$ nyelvvel is.

Sikerült az összegyűjtött követelményeknek megfelelő programot létrehozni és mindezt kizárólag szabadon elérhető technológiák felhasználásával, valamint az objektum-orientált szemléletmód és a témával kapcsolatos programozási minták alkalmazásával. A program ugyan csak a legalapvetőbb nyelvi elemeket támogatja, viszont ezek azok amiket a felhasználók jelentős többsége napi munkája során használ. Így az alapelemekre építkezve lehetőség nyílhat a program esetleges továbbfejlesztésére is. A $\text{\LaTeX}$ egy olyan speciális nyelvet biztosít, amely teljesen személyre szabható, módosítható és kiegészíthető. Ebből fakadóan nem könnyű rá minden elemére kiterjedő feldolgozó programot írni (hacsak nem lehetetlen). Mivel már régóta elérhető a nagyközönség számára, több ezer makrócsomag jött létre a funkcionalitásának bővítésére. Ezek mindegyikének értelmezése és leprogramozása lehetetlen feladat, de ha

össze gyűjtjük a leginkább elterjedt és a mindennapokban legtöbbek által használt csomagokat jelentősen csökkenthetjük a ránk váró munkát. Azonban meg kell jegyezni, hogy még ebben az esetben is több száz lehet azoknak a kiegészítéseknek a száma, amikre jövőbeli felhasználóink támaszkodnak. Tehát ez nem egyetlen embernek való feladat.

Személy szerint nagyon örülök, hogy megismerkedhettem az informatika ezen rendkívül érdekes témakörével, amely a nyelvek felismerésével és a szövegek elemzésével foglalkozik. Közben sikerült több, az iparban is használt, széles körben elterjedt technológia alapjait is elsajátítanom. Külön hasznosnak érzem, hogy a tanulmányaim során szerzett ismereteket, úgy mint például a fa adatszerkezetek használatát, a formális nyelvtanokkal való munkát, az objektum-orientált paradigma alkalmazását vagy akár az XML technológia felhasználását a gyakorlatban is alkalmazhattam egyetlen alkalmazásban.

Felhasznált könyvek

- [K1] Ralph Johnson John M. Vlissides Erich Gamma, Richard Helm. *Programtervezési minták*. Kiskapu, 2004.
- [K2] Leslie Lamport. *L^AT_EX - A Document Preparation System*. Addison-Wesley, 2nd edition, 1994.
- [K3] Terence Parr. *Language Implementation Patterns*. Pragmatic Bookshelf, 1st edition, 2009.
- [K4] Szabó Péter Wettl Ferenc, Mayer Gyula. *L^AT_EX kézikönyv*. Panem, 2004.
- [K5] Michael J. Young. *XML lépésről lépésre*. Szak kiadó, 2002.

Internetes források

- [I1] DocBook. URL: <http://en.wikipedia.org/wiki/Docbook>.
- [I2] Extensible Markup Language (XML) 1.0 (Fifth Edition) W3C Recommendation. URL: <http://www.w3.org/TR/xml/>.
- [I3] L^AT_EX/Introduction. URL: <http://en.wikibooks.org/wiki/LaTeX/Introduction>.
- [I4] Mathematical Markup Language (MathML) Version 3.0 W3C Recommendation. URL: <http://www.w3.org/TR/MathML3/>.
- [I5] Open Document Format. URL: <http://en.wikipedia.org/wiki/OpenDocument>.
- [I6] Oracle StAX API tutorial. URL: <http://download.oracle.com/javase/tutorial/jaxp/stax/api.html>.
- [I7] parboiled wiki. URL: <https://github.com/sirthias/parboiled/wiki>.
- [I8] Visitor Design Pattern. URL: http://sourcemaking.com/design_patterns/visitor.
- [I9] W3C XML Schema Tutorial. URL: <http://www.w3schools.com/schema/default.asp>.

- [I10] W3Schools XML tutorial. URL: <http://www.inf.unideb.hu/~jeszy/download/xml/xml-intro-2x2.pdf>.
- [I11] What is Maven? URL: <http://maven.apache.org/what-is-maven.html>.
- [I12] Jeszenszky Péter. Az XML 1.0 ajánlás. URL: <http://www.inf.unideb.hu/~jeszy/download/xml/xml-spec-2x2.pdf>.
- [I13] Jeszenszky Péter. XML Bevezető. URL: <http://www.inf.unideb.hu/~jeszy/download/xml/xml-intro-2x2.pdf>.

Hivatkozások

- [H1] ANTLRMorhp. URL: <http://www.antlr.org/wiki/display/Morph/Home>.
- [H2] DOM API. URL: <http://download.oracle.com/javase/tutorial/jaxp/dom/index.html>.
- [H3] DOM Technical Reports. URL: <http://www.w3.org/DOM/DOMTR>.
- [H4] DTD Tutorial. URL: <http://www.w3schools.com/dtd/default.asp>.
- [H5] Eclipse IDE. URL: <http://eclipse.org>.
- [H6] NetBeans IDE. URL: <http://netbeans.org>.
- [H7] OASIS. URL: <http://www.oasis-open.org/home/index.php>.
- [H8] ODFAlliance. URL: <http://www.odfalliance.org/>.
- [H9] Reguláris kifejezések. URL: http://en.wikipedia.org/wiki/Regular_expression.
- [H10] SAX API. URL: <http://download.oracle.com/javase/tutorial/jaxp/dom/index.html>.
- [H11] Schematron. URL: <http://www.schematron.com/>.
- [H12] SGML. URL: <http://www.w3.org/TR/html4/intro/sgmltut.html>.

-
- [H13] Stratego/XT. URL: <http://www.strategoxt.org/>.
- [H14] Sun Microsystems. URL: http://en.wikipedia.org/wiki/Sun_Microsystems.
- [H15] Uniform Resource Identifier. URL: <http://www.ietf.org/rfc/rfc3986.txt>.
- [H16] Donald Ervin Knuth. *The T_EXbook*. Addison-Wesley, 1984.
- [H17] Donald Ervin Knuth. *A számítógép programozás művészete*. Műszaki Könyvkiadó, 1994.