



**NEUMANN JÁNOS
INFORMATIKAI KAR**



SZAKDOLGOZAT

**OE-NIK
2017**

Hallgató neve:
Hallgató törzskönyvi száma:

**Ozsvárt Károly
T/003699/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Alkalmazott Informatikai Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve:	Ozsvárt Károly
Törzskönyvi száma:	T/003699/FI12904/N
Neptun kódja:	GLBXQU

A dolgozat címe:

Pszudokód felismerő készítése
Pseudocode recognizer creation

Intézményi konzulens:	Légrádi Gábor
Külső konzulens:	

Beadási határidő:	2017. december 15.
-------------------	--------------------

A záróvizsga tárgyai:	Számítógép architektúrák Vállalati információs rendszerek specializáció
-----------------------	---

A feladat

Adja meg egy pszeudokód formájú nyelv leírását, nyelvdefinícióját! Tisztázza a téma megértéséhez szükséges alapfogalmakat! Készítse el a nyelv felismeréséhez használható alkalmazás vázlatos és részletes rendszertervét, majd valósítsa meg egy tetszőlegesen választott programozási nyelven a felismerés (analízis) fázisnak feladatát! Oldja meg a feladat megoldása közben felmerülő problémákat, valamint az elkészítés során szerzett tapasztalatokat is dokumentálja! Vizsgálja meg a kódgenerálás lehetséges módozatait!

A dolgozatnak tartalmaznia kell:

- a fordítóprogramok működésének megértéséhez szükséges alapvető ismeretek leírását,
- az elkészített pszeudokód formájú nyelv teljes nyelvdefinícióját,
- a nyelv felismeréséhez szükséges alkalmazás vázlatos rendszertervét,
- a nyelv felismeréséhez szükséges alkalmazás részletes rendszertervét,
- az alkalmazás első fázisának tekintett feladat megoldásának implementációját,
- a feladat megoldásának folyamatát bemutató dokumentációt, részletesen kitérve a felmerült problémák megoldási vagy áthidalási lépéseinek leírására,
- A kódgenerálás módszereit bemutató rendszertervet.

Ph.

.....
Dr. Sergyán Szabolcs
intézetigazgató

A szakdolgozat elévülésének határideje: **2019. május 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens



HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2017.

.....
hallgató aláírása

Tartalomjegyzék

Köszönetnyilvánítás.....	8
Rövid tartalmi összefoglaló a téma területéről, a feladatról	9
A megoldandó probléma és motivációi	10
1. A fordítóprogramok vázlatos működése	11
1.1. Áttekintés	11
1.2. Lexikális elemző	12
1.2.1. Előre olvasás.....	13
1.2.2. Formális felírás és a kimenet.....	14
1.2.3. Szimbólumtábla.....	14
1.1. Szintaktikus elemző	16
1.1.1. A szintaktikus elemzők típusai	16
1.1.2. A szintaxisfa	17
1.1.3. Rekurzió	19
1.2. Szemantikus elemző.....	19
1.3. Kódgeneráló	19
1.4. Kódoptimalizáló.....	19
1.5. A fordítási fázisok komplexitásának összehasonlítása [2].....	20
2. A feladat elemzése.....	21
2.1. Lexikális elemző	21
2.2. Szintaktikus elemző	21
2.3. Szemantikus elemző.....	21
2.4. Kódgeneráló	21
2.5. Kódoptimalizáló.....	21
3. Fejlesztőeszközök kiválasztása	22
3.1. Programozási nyelv választás	22
3.2. Operációs rendszer és fejlesztői környezet választás.....	22
4. A felhasznált pseudonyelv definiálása.....	24
4.1. Áttekintés	24
4.2. Bevezetés.....	25
4.3. Lefoglalt szavak	25

4.4.	Változóhasználat, típusok	26
4.5.	Típuskonverziók.....	27
4.6.	Operátorok.....	27
4.7.	Kommentezés.....	28
4.8.	Vezérlési szerkezetek	28
4.9.	I/O kezelés.....	28
4.10.	Tömbkezelés.....	28
5.	Lexikális elemző.....	29
5.1.	Tervezés és implementáció	29
5.1.1.	Lexikális elem kódok	29
5.1.2.	Kimeneti szimbólumsorozat	30
5.1.3.	Szimbólumtábla.....	30
5.1.4.	Elemző.....	31
5.2.	Tesztelés.....	32
5.2.1.	Tesztelési keretrendszer kiválasztása	32
5.2.2.	A lexikális elemző tesztelése.....	33
5.2.3.	Néhány fontosabb teszteset	34
5.3.	Fejlesztési tapasztalatok	35
6.	Szintaktikus elemző.....	36
6.1.	Tervezés	36
6.1.1.	Elemzőtípus választása.....	36
6.1.2.	Megadási forma választása.....	36
6.1.3.	A szintaktikus nyelvtan definiálása	37
6.2.	Implementáció.....	38
6.2.1.	A szintaxisfa	38
6.2.2.	Az elemző alapja	38
6.2.3.	Statikus szabályok	39
6.2.4.	Dinamikus szabályok.....	39
6.2.5.	Hibakezelés.....	40
6.3.	Tesztelés.....	41
6.4.	Fejlesztési tapasztalatok	42

7.	Szemantikus elemző	43
7.1.	Tervezés	43
7.1.1.	A szemantikus elemző feladatainak meghatározása.....	43
7.1.2.	Működési elv	43
7.2.	Implementáció.....	44
7.2.1.	Az implementáció részei	44
7.2.2.	Hibakezelés.....	45
7.3.	Tesztelés.....	45
7.4.	Fejlesztési tapasztalatok	46
8.	Kódgenerálási lehetőségek	47
8.1.	A célnyelv kiválasztása	47
8.2.	A kódgenerálás módja.....	47
8.3.	Kulcsszó transzformációk	48
8.4.	Operátor transzformációk.....	49
8.5.	Konverziós függvények transzformációja.....	49
8.6.	Megjegyzések.....	49
9.	Az eredmények bemutatása, értékelése	50
10.	Tartalmi összefoglaló.....	51
11.	Abstract.....	52
12.	Irodalomjegyzék	53
13.	Mellékletek	54

KÖSZÖNETNYILVÁNÍTÁS

Szeretném megköszönni Dr. Sergyán Szabolcs segítségét, aki a nyelvdefiníciókra adott véleményeként hasznos tanácsokkal látott el; néhány helyen rávilágított pontatlanságokra és hiányosságokra. Valamint köszönöm Légrádi Gábornak is hogy konzulensemként segítette a munkámat.

RÖVID TARTALMI ÖSSZEFOGLALÓ A TÉMA TERÜLETÉRŐL, A FELADATRÓL

A számítógépek döntő többségében megtalálhatóak szoftverek. A szoftverek elkészítéséhez (ha azt nem gépi kódban írta a készítő) általában szükség van egy programozási nyelvre és ebből következően egy fordítóprogramra (compiler) vagy értelmezőre (interpreter). A fordítás két fő fázisának nevei: analízis fázis és szintézis fázis.

Szakdolgozatom célja egy olyan felismerő alkalmazás megtervezése és elkészítése, amely képes egy általam definiált pszeudonyelven írt programkódot felismerni, azaz a fordítás analízis fázisát elvégezni. Ehhez szükséges a fent említett pszeudonyelv definiálása, a definíció korrekt formában történő leírása, valamint az analízis fázis megtervezése és implementálása egy tetszőlegesen választott programozási nyelven.

Az analízis fázis befejezése után a szintézis fázis elengedhetetlen részét, a kódgenerálás feladatát elemzem, és elkészítek hozzá egy vázlatos rendszertervet.

A MEGOLDANDÓ PROBLÉMA ÉS MOTIVÁCIÓI

Állítom, hogy a számítógép programozás oktatásában nincsenek jól bevált, kiforrott technikák, módszerek. Az állítást empirikus alapon teszem, ugyanis szakdolgozatom írását megelőző nagyjából hét-nyolc évben részt vettem elméleti, illetve gyakorlati programozás tanórákon. Ezt az időtartamot két különböző oktatási intézményben és – ha jól számoltam össze – kilenc különböző tanárral töltöttem és tapasztalataim alapján úgy érzem, hogy fontos lenne a programozás oktatás minőségét javítani. Természetesen nem kívánom felelősségre vonni előző tanárait, ugyanis ez a tudományterület – például egy matematikával összehasonlítva, amit évszázadok, évezredek óta tanítanak – a modern számítógépes világgal együtt az elején tart még, így érthető, hogy nincsenek jól bevált módszerek.

Az algoritmusok tanításához, megértéséhez az oktatók gyakran használnak pszeudókódot, amely segítségével szövegesen definiálható egy algoritmus működése. A pszeudokódnak van néhány tulajdonsága, amelyeket érdemes kiemelni: *formális*, *elméleti*, *nem egységesített*, így egységes fordítóprogramok sem létezhetnek hozzá.

A fentiekből adódik a hátrány, hogy ha egy diák szeretné, nem tudja kipróbálni a pszeudokódban megírt programját. Szakdolgozatom ennek a problémának megoldására törekszik: definiál egy pseudonyelvet majd megad hozzá egy, a nyelvben írt programok ellenőrzésére használható, a fordítás analízis fázisát elvégző programot. A definiált pseudonyelv számos elemében szándékosan hasonlít az Óbudai Egyetem Neumann János Informatikai Karán, Mérnökinformatikus BSc. szakon tanított Programozás I. tárgy keretein belül használt pseudonyelvre, így elkészültével a szakdolgozatom hasznosítható lehet az oktatásban.

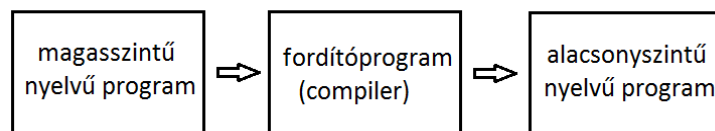
Nem csak a fentiek miatt választottam ezt a témát, hanem azért is, mert érdekel, hogy hogyan működik egy programozási nyelv; hogyan működnek a fordítóprogramok. Azzal, hogy megírok egy sajátot, biztosan megismerem majd a témakört.

1. A FORDÍTÓPROGRAMOK VÁZLATOS MŰKÖDÉSE

1.1. Áttekintés

A tudományos, illetve mérnöki munkának gyakran fontos részét képezi a számítógépek használata. Az általános számítógépek felépítését tipikusan úgy valósítják meg, hogy valamilyen hardvereken futtatnak szoftvereket. Az ezek között történő kommunikáció nem egyértelmű, ugyanis egy a felhasználó által adott parancs (például az, hogy rákattint egy gombra) túlságosan absztrakt a számítógép számára, hogy azt közvetlenül értelmezni tudja; emiatt szükség van úgynevezett fordítóprogramokra. Egy fordítóprogram feladata, hogy egy (úgynevezett) forrásnyelvben adott forráskódot egy célnyelvű futtatható programra fordítsa.

A fordítás nagyon gyakran egy magas szintű programozási nyelvből egy alacsony szintűbe (jellemzően gépi kódba vagy köztes kódba) történik, ezt mutatja az 1.1. ábra.

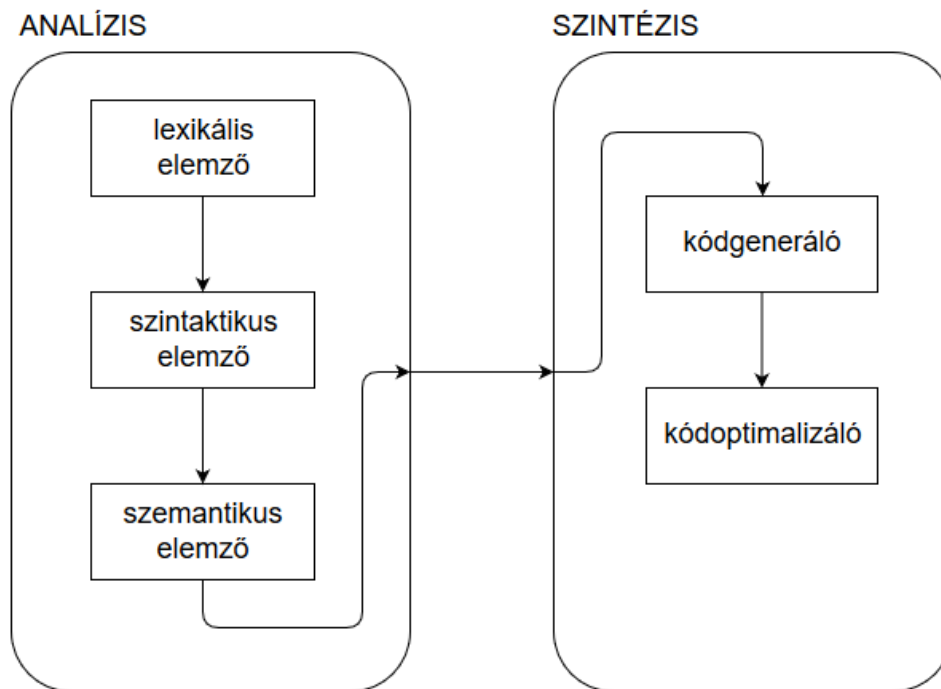


1.1. ábra - A fordítóprogramok általános be és kimenete ([1] alapján)

Egy fordítóprogram két fázisból áll: analízis fázis és szintézis fázis. Ezek a fázisok vázlatosan áttekintve a következőket végzik el:

- Az analízis fázis meghatározza, hogy a bemeneti véges karakterlánc által megadott programkód lexikálisan, szintaktikailag és szemantikailag helyes-e, azaz megfelel-e a nyelvdefinícióban leírt formának. Ha a bemeneti programkód ilyen szempontból helyes, akkor a futtatható kód is kigenerálható.
- A szintézis fázis feladata a kódgenerálás és kódoptimalizálás. Ha ez a folyamat is sikeresen zárul, akkor kimenetként keletkezik a futtatható alacsonyszintű nyelvű program, jellemzően gépi kódú program. Ennek elindításával indul el a kód, amit az eredeti bemenet által meghatároztunk.

Az előzőek átlátását segíti az 1.2. ábra.



1.2. ábra - A fordítóprogramok szerkezete

1.2. Lexikális elemző

A lexikális elemző feladata bemeneti szöveg formális felírása és a szimbólumtábla felépítésének elkezdése. A formális felírás rendszerint úgy történik, hogy az elemző végighalad a teljes bemeneti karakterláncon, és megkísérli felismerni a lehető leghosszabb karaktersorozatú lexikális elemeket. A hosszúság szükségességének megértését segíti a következő példa:

Tegyük fel, hogy egy C nyelvet felismerő lexikális elemző a bemenetén a következő karaktersorozat-részletet látja: „**ifa = 10;**”. Ez esetben a programozó az **ifa** nevű változóba szeretné belemásolni a **10**-es értéket. Ha az elemző nem a fent említett elven működne, akkor esetleg tévesen ismerné fel a szöveget három különböző lexikális elemnek: **i**-nek, **f**-nek és **a**-nak (három különböző azonosító), vagy más esetben **if**-nek és **a**-nak (egy kulcsszó és egy azonosító). Ezek alapján kijelenthető, hogy egy lexikális elemzőnél alapelvnek kell tekinteni, hogy mindig a lehető leghosszabb lexikális elemet ismerje fel.

Ésszerű lehet a fentiek alapján a következő logika szerinti működés: Legyen egy üres karaktersorozat, majd haladjunk végig a bemeneti karakterláncon. Ha a karaktersorathoz fűzött éppen aktuális karakter értelmes lexikális elemet ad, akkor azt tároljuk el és lépünk tovább. Ismételjük ezt addig, amíg a bővítés helyes lexikális elemhez vezet. Amint a bővítéssel helytelen lexikális elemet találunk, szakítsuk meg az aktuális részsorozat elemzését, és lépünk vissza a legutóbb helyesen felismert részletre. Ezt tároljuk el,

majd a részlet után következő karaktertől folytassuk tovább a fentieket. Ezt a működést mutatja be a következő példa:

Elemzendő karaktersorozat: **ifa = 10;**

(az elemzés során jelentse `_` a szóköz karaktert)

#	aktuális szöveg	Lexikális elem?	Utolsó helyes hossz	Eredmény
1	i	igen (azonosító)	1	→ halad tovább
2	if	igen (kulcsszó)	2	→ halad tovább
3	ifa	igen (azonosító)	3	→ halad tovább
4	ifa_	nem	3	→ megáll

Tehát a felismert részsorozat az **ifa**, és ezt az elemző (sikeresen) egy azonosítóként ismerte fel. Ezek után az elemző a következő karaktertől (szóköz) halad tovább. A szóközre a modern nyelvek általában úgy tekintenek, hogy az mindig elválaszt két lexikális elemet, azaz nem állhat elő olyan állapot, hogy egy darab lexikális elemen belül szóköz karakter van. Ebből következően az elemző ténylegesen a szóköz karakter után folytatja a felismerést, de ez már az adott nyelvtől függ; a fenti példa csak az elv megértését szolgálta.

1.2.1. Előre olvasás

Az elemző dolga nem ilyen egyszerű, ugyanis időnként szükség van úgynevezett *előre olvasásra*, mert enélkül a lexikális elemző bizonyos karaktersorozatokat rosszul ismerne fel. Ennek megértését segíti a következő példa:

Egy fiktív nyelvben értelmezzünk egész és lebegőpontos számokat. A nyelv tizedespontot használ. Az egész számok mögött nem lehet pont (nem megengedett pl. ez a forma: „**2.**”). A lebegőpontos számok formája pedig `x.y` formájú, ahol `x` és `y` egész számok. Feladat, hogy ismerjük fel ezen nyelven a következő karaktersorozatot: „**61.52**”:

#	aktuális szöveg	Lexikális elem?	Utolsó helyes hossz	Eredmény
1	6	igen (egész)	1	→ halad tovább
2	61	igen (egész)	2	→ halad tovább
3	61.	nem	2	→ megáll

Látható, hogy habár mi a **61.52** lebegőpontos számra gondoltunk, a lexikális elemző ezt nem képes felismerni. Tulajdonképpen az elemző a fenti működéssel egyetlen lebegőpontos számot sem képes felismerni, mert előtte mindig érzékel egy egész számot, majd egy tizedespontot.

A hiba nem feltétlenül a nyelv definíciójában keresendő, hanem az elemző algoritmusban, ugyanis a fenti esetben szükség van arra, hogy ha egy egész számot elkezdünk felismerni, akkor az egész szám után talált pont esetén ne álljon le az adott szöveget felismerő rész, hanem haladjon tovább, *olvasson előre*, és nézze meg, hogy nem-e lebegőpontos szám van a bemeneten.

1.2.2. Formális felírás és a kimenet

Ez a fejezet az elemző kimenetével foglalkozik. Nyilvánvaló, hogy annak érdekében, hogy a lexikális elemző csak egyszer haladjon végig a bemeneten, már a feldolgozás közben fel kell építeni a kimenetet képező adatstruktúrát.

Ha egy lexikális elemet sikeresen felismertünk, akkor azt strukturált, formális módon el kell tárolni. Az alapeseteket tekintve két fő lexikális elemet definiálhatunk:

- a) Paraméter nélküli lexikális elem
- b) Paraméteres lexikális elem

Társítsunk minden elemhez egy számkódot, például.: literál = 01, azonosító = 02, **if** = 03, **else** = 04, **for** = 05, stb. A számkódos tárolás egyik előnye lehet, hogy különböző elemcsoportokat hasonló számokkal ellátva, könnyen tudjuk őket szűrni (pl. ha a literálokat külön csoportosítjuk típus szerint, akkor lehetnek három számjegyűek és mind-egyik kódja kezdődjön 1-essel, pl. 1xx).

Az **a)** csoportba tartoznak azon nyelvi elemek, amelyek önmagában jelentést hordoznak és nem szükséges hozzájuk további információkat csatolni, pl.: C-nyelvben az **if**, **else**, **for**, **while**; a Pascal nyelvben a **begin**, **end**, **program** kulcsszavak. Ezeknél elegendő a kimenethez az elemhez tartozó számkódot hozzáfűzni, pl. **if** esetén a 03-at.

A **b)** csoportba tartoznak a literálok (konstansok) és azonosítók (változónevek, függvénynevek, stb.). Ezek esetén a kimenethez nem elegendő csak a számkódot hozzáfűzni, hanem azt is szükséges tárolni, hogy mi a hozzá kapcsolódó érték. Például ha a felhasználó leírja az alma szót, akkor a kimenethez hozzáfűzzük az azonosító számkódját (pl. 02), és az **alma** szöveget.

A kimenet adatszerkezeteként a fentiek alapján ideális lehet egy lista használata, ahol a lista egy eleme egy olyan objektum, ami számkódot vagy számkód-érték párost tárol.

1.2.3. Szimbólumtábla

A lexikális elemző kezdi el felépíteni az úgynevezett szimbólumtáblát, amelyre a fordítás több alfázisában is szükség van. „*A szimbólumtábla egy olyan táblázatnak tekinthető, amelyben egy sor egy szimbólum nevét és programbeli jellemzőit, azaz attribútumait tartalmazza. Az attribútumok elsősorban az adott programnyelvtől függenek (...).*”[1] A szimbólumtáblában tárolhatók a következő adatok:

- szimbólum neve
- szimbólumhoz tartozó attribútumok:
 - a szimbólum definíciójának adatai,
 - a szimbólum típusa,
 - a szimbólum tárgyprogrambeli címe,
 - annak a forrásnyelvi sornak a sorszáma, amelyben a szimbólumot definiálták,
 - azoknak a forrásnyelvi soroknak a sorszámai, amelyekben a szimbólumra hivatkoztak,
 - a szimbólum ábécé-sorrendbeli láncolási címe

(forrás: [1])

Célszerű a szimbólumtábla egy bejegyzését ellátni egy azonosítóval (ID), ami egyértelműen azonosítja az adott szimbólumot, így ezt az ID-t lehet beírni az előző fejezetben említett *számkód-érték* páros helyett az *érték* helyére.

A szimbólumtáblán értelmezett fontosabb műveletek:

- keresés: egy értékhez tartozó ID megtalálása
- beszúrás: adott helyre történő beszúrás

Adatszerkezeti lehetőségek a szimbólumtáblához:

- lista
- verem
- fa
- hash

Az adatstruktúra megválasztása azért fontos, mert az adatstruktúrán értelmezett műveletek gyorsasága kihat az egész fordítóprogram fordítási sebességére. A fentiek közül a leggyorsabbnak a hash adatszerkezet tűnik $O(1)$ sebességű műveletei sebességei miatt, azonban ez rendezetlen, és további adatok tárolását eredményezheti. Ez nem feltétlen probléma, ugyanis a modern számítógépek memóriája elég nagy ahhoz, hogy ez ne tűnjön fel; cserébe gyorsabb lesz a fordítási folyamat.

A bemenet feldolgozásának során a szimbólumtábla kezelése a következőképpen kezdődik. A lexikális elemző a program kezdetekor létrehozza a szimbólumtáblát. Ezek után minden szimbólum felfedezésekor a következőt végzi el:

Keressük meg az azonosítót a szimbólumtáblában.

**Ha (a szimbólumtábla már tartalmazza az azonosítót) akkor
a kimenetre helyezzük a megtalált azonosítót**

Különben

hozzáadjuk az azonosítót a szimbólumtáblához

a kimenetre helyezzük az új azonosítót

Elágazás vége

A fentiek segítségével már a lexikális elemző is végezhet hibakezelést, ugyanis így dektálható egy változó újra deklarálása.

A szimbólumtábla kezelését tovább bonyolítja, hogy blokkstruktúrájú programnyelvek esetén azt is el kell tárolni, hogy egy azonosítót melyik blokkban definiálunk és használunk, ugyanis a hatókör-problémákat csak így lehet felderíteni.

1.1. Szintaktikus elemző

„A lexikális elemző által készített szimbólumsorozat a bemenet a szintaktikus elemzőnek. A szintaktikus elemzőnek a feladata a program struktúrájának a felismerése és ellenőrzése. A szintaktikus elemző azt vizsgálja, hogy a szimbólumsorozatban az egyes szimbólumok a megfelelő helyen vannak-e, a szimbólumok sorrendje megfelel-e a programnyelv szabályainak, nem hiányzik-e esetleg valahonnan egy szimbólum. Ez a folyamat hasonlít ahhoz, amikor a magyar nyelvtan órán egy mondat alanyát, állítmányát, tárgyát, határozóit és jelzőit határozzák meg. A szintaktikus elemző működésének az eredménye az elemzett program szintaxisfája, vagy valamilyen ezzel ekvivalens struktúra.” ([1]) Emellett felépíti a szintaxisfát, amely a kódgenerálás fázis egyik bemenete.

1.1.1. A szintaktikus elemzők típusai

A szintaktikus elemzőket az alapelveik alapján három csoportba sorolhatjuk[3]:

- **Univerzális:** például az Earley algoritmus vagy a CYK-algoritmus; ezek habár nevükből adódóan univerzálisak, alacsony hatékonyságuk miatt nem szokták őket tényleges programozási nyelvek fordítóinak implementálására használni. Valójában ezek is a következő pontokban említett top-down vagy bottom-up megközelítést használják, azonban érdemes őket különválasztani a többitől.
- **Top-down** (fentről lefelé): a szintaxisfát a gyökértől (teteje) kiindulva a levelek felé (alja) haladva építi fel.
- **Bottom-up** (lentől felfelé): a szintaxisfát a levelektől (alja) kiindulva a gyökér felé (teteje) haladva építi fel.

Megfogalmazható az utóbbi kettő közötti különbség úgy is, hogy a top-down elemző az egészből halad a részletek felé, a bottom-up elemző pedig a részletek összerakásával képezi az egészet.

Egy elemző a bemenetét mindkét irányból (bal és jobb) elkezdheti egyesével feldolgozni, azonban a gyakorlatban csak a balról-jobbra történő feldolgozást szokták használni. Az

elemzők csoportjait általában egy rövidített, angol névvel szokták ellátni; a nevekben található első, **L** betű általában azt jelzi, hogy az elemző a bemenetet balról-jobbra (**L**eft to **r**ight) dolgozza fel.

Egy másik szempont az, hogy a nyelvtan által definiált produkciós szabályok jobb oldalán szereplő terminálisokat és nemterminálisokat milyen sorrendben dolgozza fel az elemző: balról-jobbra (**L**eftmost derivation – legbaloldalibb levezetés), vagy jobbról-balra (**R**ightmost derivation – legjobboldalibb levezetés). Ennek megfelelően a következő elemző típusok alakulnak ki:

- **LL-elemző**: a bemenetet balról-jobbra dolgozza fel, és a mondat legbaloldalibb levezetését alkalmazza.
- **LR-elemző**: a bemenetet balról-jobbra dolgozza fel, és a mondat legjobboldalibb levezetését alkalmazza.

Természetesen még rengeteg féle elemző típus van, azonban a téma megértéséhez ezek az „alaptípusok” ismerete is elegendő.

Megjegyzendő, hogy a legbaloldalibb levezetést alkalmazó elemzők a top-down, a legjobboldalibb levezetést alkalmazók pedig a bottom-up elvet követik.

A fentiekből következik, hogy attól függően lehet egy nyelvtan szabályrendszere balrekurzív vagy jobbrekurzív, hogy a produkciós szabályokat melyik irányból kezdjük el alkalmazni, ugyanis ha mindig a legbaloldalibb szabályt alkalmazzuk, akkor balrekurzió esetén ez végtelen ciklushoz vezet:

Tekintsük a következő nyelvtant:

$S \rightarrow Sa \mid b$

Amely a következő bemenetet szeretné elemezni: **b**

Ebben az egyszerű esetben egy LL-elemző mindig a legbaloldalibb szabályt próbálja alkalmazni ($S \rightarrow Sa$), így ezt a végtelenségig folytatná. Természetesen egy LR-elemző a másik irányból indulva azonnal alkalmazni képes az $S \rightarrow b$ szabályt, így felismeri a bemenetet.

1.1.2. A szintaxisfa

A szintaxisfa egy olyan gráf, amely a bemeneti programkód szintaktikus szerkezetét ábrázolja egy fa formájában. Ez a kimenete a szintaktikus elemzőnek, és többek között ennek segítségével generálható ki a végleges kód a kódgenerálás fázisában.

Tekintsük a következő példát ([3] alapján). A szabályok a könnyebb hivatkozás érdekében sorszámmal rendelkeznek.

- 1) $E \rightarrow E + T$
- 2) $E \rightarrow T$

$$3) T \rightarrow T * F$$

$$4) T \rightarrow F$$

$$5) F \rightarrow \text{id}$$

Ez a nyelvtan láthatóan balrekurzív, azaz a legjobboldalibb levezetés elvével fogjuk elemezni a bemenetet.

Legyen a bemenet a következő: $\text{id} + \text{id} * \text{id}$

A bemenet a következő sorrendben vezethető le a start-szimbólumból (E) (előtte az alkalmazott szabály sorszáma):

$$1) E \rightarrow E + T$$

$$3) E + T \rightarrow E + T * F$$

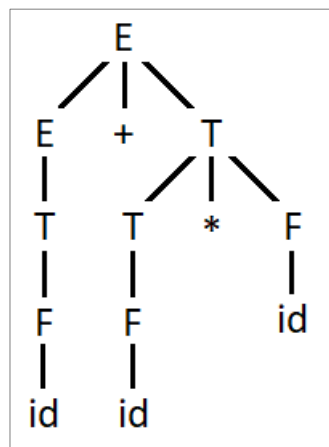
$$4) E + T * F \rightarrow E + F * F$$

$$2) E + F * F \rightarrow T + F * F$$

$$4) T + F * F \rightarrow F + F * F$$

$$5) F + F * F \rightarrow \text{id} + \text{id} * \text{id} \quad (\text{a szabály háromszori alkalmazásával a különböző F-ekre})$$

Ez pedig reprezentálható az 1.3. ábrán látható fával:



1.3. ábra – A fenti levezetés szintaxisfája

Látható, hogy a szintaxisfa leveleit balról-jobbra olvasva megkapjuk az elemzett bemeneti sorozatot: $\text{id} + \text{id} * \text{id}$; szerkezete pedig megmutatja, hogy milyen sorrendben kell alkalmazni a nyelvtani szabályokat a levezetésben.

Megjegyzendő, hogy a szintaxisfa minden levele törvényszerűen mindig terminális jel, minden egyéb csúcsa pedig nemterminális jel.

1.1.3. Rekurzió

Az előző két alfejezet alapján belátható, hogy egy nyelvtan szabályrendszere vagy csak balrekurzív, vagy csak jobbrekurzív lehet, ha ennek megfelelően LL, illetve LR elemzővel szeretnénk feldolgozni.

Ha egy szabályrendszer nem rekurzív, értelemszerűen csak véges számú bemenetet képes feldolgozni, és így ezt a gondolkodásmódot a jelenlegi feladatra alkalmazva a nyelvben csak véges számú különböző programkódot lehet írni. Ez természetesen nem elfogadható, így elengedhetetlen, hogy a nyelvhez elkészített szabályrendszer majd megfelelő számú rekurziót tartalmazzon.

1.2. Szemantikus elemző

A szemantikus elemző a szintaxisfa és a szimbólumtábla felhasználásával ellenőrzi a programkód szemantikai helyességét. Fontos részei ([1] alapján):

- típus ellenőrzés:
 - o egy kifejezés két oldalán összehasonlítható típusok szerepelnek-e
 - o az operátorok és operandusok kompatibilitásának ellenőrzése
 - o a függvények paraméterei megfelelő típusúak-e
- tömbindexek ellenőrzése (kiindexelünk-e a tömbből?)
- hatókör ellenőrzés
- függvénynév túlterhelés egyértelműségének ellenőrzése
- változók deklarációja, többszörös deklarációja, a deklaráció hiánya

Az esetlegesen felmerülő kérdésre – hogy ezeket a megkötéseket miért nem lehet a szintaktikus elemzőben elvégezni – az a válasz, hogy a szintaktikus elemzők csak környezet-független nyelvtanokat (CFG – context-free grammar) képesek felismerni, és ilyen nyelvtanokban nem írhatók le ezek a szabályok.

Megjegyzendő, hogy ugyan az analízis fázis három alfázisa közül az első kettőre – a lexikális és szintaktikus elemzésre – sok jól kiforrott algoritmus, technika létezik, a szemantikus elemzőkről ezt nem jelenthető ki.

1.3. Kódgeneráló

A kódgeneráló feladata, hogy az analízis fázis kimenetéből, az analizált programból futtatható kódot eredményezzen, ahol az utóbbi kód nyelve a fordítóprogram célnyelve.

1.4. Kódoptimalizáló

„A kódoptimalizálás a legegyszerűbb esetben a tárgykódban lévő azonos programrészek felfedezését és egy alprogramba való helyezését, vagy a hurkok ciklusváltozótól független részeinek megkeresését és a hurkon kívül való elhelyezését jelenti. Bonyolultabbak a

gépfüggő kódoptimalizáló programok, amelyek például optimális regiszter-használatot biztosítanak. A fordítóprogramok íróinak a célja, hogy a kódoptimalizáló jobb és hatékonyabb tárgyprogramot állítson elő, mint amit egy (az assembly nyelvű programozásban) gyakorlott programozó készíteni tud.” [1]

Megemlítendő, hogy egyes irodalmak (pl. [3]) a kódoptimalizáló előtti kódgenerálás részben köztes kódot generálnak („köztes kód generálás”), azon végzik el az optimalizálást, és csak ez után generálják ki leképezéses folyamattal a végleges célnyelvű programot.

1.5. A fordítási fázisok komplexitásának összehasonlítása [2]

A korai fordítóprogramok (pl. FORTRAN) esetén a fázisok komplexitását szemlélteti az 1.4. ábra.

lexikális elemző	szintaktikus elemző	szemantikus elemző	kódoptimalizáló	kódgeneráló
------------------	---------------------	--------------------	-----------------	-------------

1.4. ábra - A korai fordítóprogramok komplexitásának fázisonkénti eloszlása ([2] alapján)

A modern fordítóprogramok esetén a fázisok komplexitását szemlélteti az 1.5. ábra.

lexikális elemző	szintaktikus elemző	szemantikus elemző	kódoptimalizáló	kódgeneráló
------------------	---------------------	--------------------	-----------------	-------------

1.5. ábra - A modern fordítóprogramok komplexitásának fázisonkénti eloszlása ([2] alapján)

A lexikális és szintaktikus elemző arányának csökkenése betudható annak, hogy e két fázis végrehajtásához már rendelkezésre állnak jó minőségű és hatásfokú automatizált programok. A kódgeneráló rész aránya pedig azért csökkent, mert a fordítóprogramok tervezői már jobban megértik ezt a fázist, mint a koraiak esetén (akkor még ez új terület volt és nehezebb volt elkészíteni). A kódoptimalizáló rész növekedését pedig a programok bonyolultságának növekedése vonta maga után (nagyobb kódbázison fontos minél többet gyorsítani).

2. A FELADAT ELEMZÉSE

Ebben a fejezetben a megoldandó feladatokat elemzem.

2.1. Lexikális elemző

Habár a lexikális és szintaktikus elemzéshez is állnak rendelkezésre automatizált eszközök (pl. Lex, Yacc, stb.), úgy döntöttem, hogy ezt a részt én implementálom. Döntésemet azzal indoklom, hogy szeretném alaposan megismerni az analízis fázisának folyamatát, és ezt úgy tudom megtenni, ha saját magam készítem el.

2.2. Szintaktikus elemző

A szintaktikus elemző implementálása lesz a legnehezebb feladat, ugyanis e fázis megértéséhez és implementálásához megfelelő szintű tudásra van szükség a *formális nyelvek és automaták* témakörében.

2.3. Szemantikus elemző

A szemantikus elemző jóval kisebb komplexitásúnak ígérkezik, mint a szintaktikus elemző, mivel tervezek már a lexikális elemzőben néhány olyan dolgot ellenőrizni, amit ott is tudok; például hogy egy változót kötelező deklarálni használat előtt és egy változót csak egyszer lehet deklarálni.

2.4. Kódgeneráló

Szakdolgozatomban vizsgálom a kódgenerálás lehetőségeit, azonban erre csak akkor lesz lehetőségem, ha a program „analizáló” (analízis fázist megvalósító) része már implementálásra került.

2.5. Kódoptimalizáló

A kódoptimalizálás fázis nem része a szakdolgozatnak, és nem is érzem szükségesnek, ugyanis, ha olyan célból készül a fordítóprogram, hogy a kezdő programozók képesek legyenek összehasonlítani bizonyos algoritmusok futási idejét, akkor a kódoptimalizáló ennek eredményét módosíthatja.

3. FEJLESZTŐESZKÖZÖK KIVÁLASZTÁSA

3.1. Programozási nyelv választás

Az implementáció elkészítéséhez számos nyelv rendelkezésemre állt. Egyetemi tanulmányaim során (a teljesség igénye nélkül) a következő nyelveket ismertem meg: C#, Java, C++, SQL, PHP, MATLAB. Ezek közül fordítóprogramok írására kevésbé használhatók az SQL és a MATLAB, ezek indoklása a következő: az SQL deklaratív nyelv és teljesen más célokra készült; a MATLAB nyelv korlátozottan támogatja az objektumorientáltságot, amit az eddigi irodalomkutatás alapján fontosnak tartok. A PHP nyelv tipikusan nem asztali alkalmazások írására készült (szükség lenne egy futtató szerverre és az iskolában használt kiváló integrált fejlesztői környezet sem ingyenes), így azt is elvetettem. A maradék három nyelv (C#, C++, Java) közül kellett tehát választanom. Azért nem a C++ vagy Java mellett döntöttem, mert ezt a két nyelvet csak kezdő szinten ismerem, illetve a C++ esetében a nyelv sokkal nagyobb egy hiba elkövetésének esélye a felhasználóra bízott memóriakezelés miatt.

Választásom így a C#-ra esett, ugyanis ezt a nyelvet ismerem a legalaposabban. A C# egy Microsoft által kifejlesztett imperatív, objektumorientált programozási nyelv, amely a .NET keretrendszerre épül. Kitűnő támogatottsággal és dokumentációval rendelkezik; ezt demonstrálja az is, hogy C# nyelven történő programozás során bármikor beleütközök egy problémába, akkor az interneten (google, stackoverflow, stb.) legtöbbször találok hasznos információt vagy kódrészletet a témával kapcsolatban. A fentiek alapján ez a nyelv kitűnően használható a szakdolgozatomhoz.

3.2. Operációs rendszer és fejlesztői környezet választás

Tudomásom szerint a C# nyelvhez az alábbi integrált fejlesztői környezetek (IDE) érhetők el:

- SharpDevelop (Windows)
- QuickSharp (Windows)
- C# Studio (Windows)
- Microsoft Visual Studio (Windows)
- MonoDevelop (Linux)

A Microsoft Visual Studio, számos hasznos eszközt biztosít arra, hogy a kódolás, hibakeresés és optimalizálás könnyebb legyen (lépésenként futtatás, IntelliSense, változók futás közben történő kezelése, stb.). Ez a környezet biztosítja a legbővebb funkcionalitást, ezt használják a legtöbben és saját célra történő szoftverfejlesztésre ingyenes a használata. Tekintettel arra, hogy egyetemi tanulmányaim során is ezt a fejlesztői környezetet ismertették meg velem, ennek használata mellett döntöttem.

Mivel a Microsoft Visual Studio natívan jelenleg csak a Windows operációs rendszer családot támogatja, azt kell használnom. Ez nem okoz problémát, ugyanis eddigi életem során is mindig Windows operációs rendszereken dolgoztam.

4. A FELHASZNÁLT PSZEUDONYELV DEFINIÁLÁSA

Ebben a fejezetben részletezem a pszeudonyelv definiálásának folyamatát. A dokumentum legfrissebb változata a dokumentum I. mellékleteként található meg. Kihangsúlyozandó, hogy e fejezet a tervezés fázisában készült, így a ténylegesen implementált nyelv bizonyos részekben eltér ettől.

4.1. Áttekintés

Egy nyelv lexikális elemeinek definiálása általában reguláris kifejezésekkel történik [4].

A szintaxis definiálására számos lehetőség áll rendelkezésre:

1. Backus-Naur form (BNF)
2. Augmented Backus-Naur form (ABNF)
3. Extended Backus-Naur form (EBNF)

(Megjegyzés: szándékosan nem fordítottam le magyar nyelvűre a neveket, mivel magyarul hasonló a jelentése az augmented és extended szavaknak.)

Saját nyelvdefinícióm azonban szándékosan úgy fogom elkészíteni, hogy azt egy kezdő programozó is meg tudja érteni. Emiatt a fent említett módokat kevésbé tervezem használni; amikor pedig használok, akkor a hozzájuk szükséges ismereteket is leírom ott helyben. A forma, amit valószínűleg mindenki meg fog érteni, az a magyarul megfogalmazott, példákkal gazdagított leírás, így ehhez a módszerhez folyamodom.

A nyelvdefiníciók tartalmi felépítését, szerkezetét szakdolgozatom írása előtt nem ismertem, így példákat kellett keresnem ebben a témában. Konzulensem az Oberon programozási nyelv definícióját javasolta áttekintésre. 1986-os indulása óta a nyelv több változtatáson esett át; áttanulmányozásra a nyelvet leíró dokumentum legújabb változatát választottam. Ez az Oberon-07 nevű nyelv definíciója (más néven: Revised Oberon) [5].

A fent említett dokumentum többek között a következő részekre osztja fel a definíciót:

1. Szókincs/szójegyzék
2. Deklarációk és hatáskör szabályok
3. Konstans deklarációk
4. Típus deklarációk (alapvető, tömb, rekord, mutató, eljárás)
5. Változó deklarációk
6. Kifejezések (operandusok, operátorok)
7. Állítások (értékadások, függvényhívások, állításszekvenciák, „if” állítások, „case” állítások, „while” állítások, „repeat” állítások, „for” állítások)
8. Eljárás deklarációk (formális paraméterek, előre definiált (beépített) eljárások, modulok)
9. Függelék (ahol megtekinthető a nyelv szintaxisa BNF formában)

A fentiek áttanulmányozása és átgondolása alapján saját nyelvdefinícióm a következő fejezetekre osztom fel:

1. Bevezetés
2. Lefoglalt szavak
3. Változóhasználat, típusok
4. Típuskonverziók
5. Operátorok
6. Kommentezés
7. Vezérlési szerkezetek
8. I/O kezelés
9. Tömbkezelés

A további alfejezetek e fejezetekről szólnak majd.

4.2. Bevezetés

Itt általános információkat adok a nyelvről, például hogy nem kisbetű-nagybetű érzékeny, nincsenek függvények, milyen utasításokkal kell kezdődnie és záródnia egy programnak.

Fontosnak tartom, hogy ha a nyelv definíciójából nem derül ki valami egyértelműen, akkor is legyen viszonyítási alapja az olvasónak, így belekerült a bevezetésbe, hogy: „*A nyelv a C# nyelvet veszi alapjául. Ami e dokumentumból nem derül ki egyértelműen, arra a C# nyelv szabályai érvényesek.*”. Így, ha esetlegesen bármit kihagyok a dokumentumból, az kevésbé okoz problémát. (Természetesen a hiányosságokat azok felderítése után pótlom.)

Habár az előző bekezdés szerint a nyelv a C# programozási nyelvet veszi alapjául, több más programozási nyelvben használatos forma is megjelenik.

4.3. Lefoglalt szavak

Itt találhatóak egy listába rendezve a lefoglalt szavak. A Pascal-hoz hasonlóan minden program elejét és végét pontosan meg kell határozni, erre szolgálnak a **program_kezd** és **program_vége** kulcsszavak. Erre azért van szükség, mert a nyelv nem definiál függvényeket, és emiatt fontosnak tartom, hogy a programozónak explicit módon¹ meg kelljen adnia, hogy hol kezdődik és végződik egy program. A lefoglalt szavak, ha több szóból állnak, konzulensem javaslatára a szóköz helyén aláhúzás karaktert tartalmaznak. Ez azért van így, mert így kissé egyszerűsödik a lexikális elemző munkája.

Habár a strukturált programozás elve szerint egy programot a program futását megszakító utasítások (break, continue, return (egy függvény közben)) nélkül érdemes megírni, a nyelv tartalmaz két utasítást, amik ezt teszik lehetővé: kilép, kilépés. Ezeket hasznosnak

¹ Explicit módon: egyértelműen, külön kihangsúlyozva a jelentést

tartom, így beépítem őket. Előfordul, hogy a nyelvben két utasítás is ugyanazt a célt szolgálja; ezt azért terveztem így, hogy a nyelvben való programozás gördülékenyebben történhessen (bosszúságot okozhat, hogy a program azért nem fordul le, mert a „kilép” helyett „kilépés”-t kell írni vagy fordítva). Tekintettel arra, hogy ezeket az „átírányításokat” könnyű beleprogramozni a lexikális elemzőbe (egyszerűen mindkét elem ugyanarra a számkódra mutat); valamint hasznosak, több alkalommal szerepelnek.

Hogy a nyelv bizonyos szinten illeszkedjen az Óbudai Egyetem Neumann János Informatikai Karán tanított pszeudonyelvhez, azon utasítások, amelyek ugyanúgy működnek, itt nem kerülnek részletezésre. Ezek megtalálhatóak *Sergyán Szabolcs – Algoritmusok, adatszerkezetek I.* című jegyzetében [6]. Ezek az utasítások a következők:

- **ha**
- **akkor**
- **különben**
- **elágazás_vége** (azonos az elágazás vége utasítással)
- **ciklus_amíg** (azonos a ciklus amíg utasítással)
- **ciklus_vége** (azonos a ciklus vége utasítással).

Mivel a számlálós és hátultesztelős ciklus kiváltható előtesztelős ciklussal, ezek nem kerülnek bele a nyelvbe; így megkönnyítve a fordítóprogram implementálását.

Tekintettel arra, hogy bemenetre és kimenetre is szükség van, léteznek **beolvas**, **beolvas:**, **kiír**, **kiír:** utasítások is. Ezek használati módja véleményem szerint triviális, külön magyarázatot nem igényelnek.

A tömbkezelést teszi lehetővé a **létrehoz** utasítás, ennek használatát a tömbkezelés fejezetben fejtem ki.

4.4. Változóhasználat, típusok

A nyelvben használható típusok a következők: egész, tört, logikai, szöveg. A nyelv típusainak kitalálása során az volt a szempontom, hogy kevés típus legyen, és azok lehetőség szerint egyszerűen érthetőek és implementálhatók legyenek.

Az egész típus a tört részhalmaza így a számításokhoz feltétlenül nem lenne rá szükség, viszont a tömbök indexelésének ellenőrzését egyszerűsítheti, ha van egy egész típus; valamint a valódi nyelveken történő programozás során is nagyrészt egész típusokat használunk az egyszerűbb, tört számokat nem igénylő számítások elvégzésére (mivel azzal gyorsabban számol a processzor).

A szöveg típusnak a bemenet és kimenet kezelés, valamint fájlírás során (erre a nyelvben nincs lehetőség) van nagyobb jelentősége. A nyelvdefiníció korábbi változataiban szere-

pelt karakter típus is, azonban később rájöttem, hogy egy karakter kiváltható egy 1 hosszúságú szöveg értékkel, így feleslegesen bonyolítanám az implementációt és növelném a típusok számát.

Hasznosnak találom a logikai típust is, így az is szerepel a nyelvben; illetve implementálását egyszerűnek tartom, ugyanis a C-nyelvhez hasonlóan ábrázolható egy egyszerű egész számként, ahol egy speciális érték (C-ben a 0) jelenti a hamist, minden más az igazat.

Az elnevezési konvenció hasonló a C# nyelvéhez, azzal a különbséggel, hogy aláhúzás karakter nem szerepelhet első karakterként ugyanis úgy gondolom, hogy ez egy rossz elnevezési szokás és nem szeretném támogatni a nyelvemben.

Léteznek továbbá (a C#-hoz hasonlóan) szöveg értékbe beilleszthető escape-szekvenciák, amelyek a be és kimeneti értékek kezelését könnyítik meg.

4.5. Típuskonverziók

Tekintettel arra, hogy a nyelv erősen típusos, szükség van konverziós függvényekre. Ezek beépített függvény formájában léteznek, és a típusok közötti átjáráshoz használhatóak. A nyelvben nincs implicit konverzió. Véleményem szerint az egyes konverziós függvények nem igényelnek indoklást, leírásuk megtekinthető a nyelvdefinícióban.

4.6. Operátorok

A nyelv operátorainak megtervezése során több nyelv is megihletett. A legtöbb operátort a C# nyelvből vettem át. Kivételt képeznek ez alól bizonyos operátorok; ezen operátorok forrását és indoklását a 4.1. táblázat tartalmazza.

Jel	Név	Forrásnyelv	Indoklás
és	feltételes és művelet	nincs	sokkal jobban olvasható
vagy	feltételes vagy művelet	nincs	
mod	maradékos osztás	[6] pszeudonyelve	a C#-ban használt % karakter nem utal a céljára
.	szöveg összefűzés	php	a + operátort már használom a szám típusokon értelmezett összeadáshoz, és nem szeretnék félreértéseket ebből (a C# esetén előfordulhat ilyen eset)

4.1. táblázat - Nem C#-ból átvett operátorok

4.7. Kommentezés

Ahogy az a nyelvdefinícióban is leírom, „*A nyelv kommentezési lehetőségei meg-
egyeznek a C# nyelvben lévőkkel (...)*”. A fejezet nem igényel további részletezést.

4.8. Vezérlési szerkezetek

A nyelvben található vezérlési szerkezetek a következők: szekvencia, szelekció, iteráció.

A szekvencia elve azt jelenti, hogy a program az utasításokat szigorú sorrendben, egymás után hajtja végre. Ez a nyelvben konstans módon létezik, mint egy szabály; míg a másik két vezérlési szerkezet opcionálisan felhasználható.

A szelekciós és iterációs szerkezetek szintaktikájának forrása: [6]. Tekintettel arra, hogy a hátul tesztelő ciklust a programozás során ritkán használjuk (pl. a *Sergyán Szabolcs – Algoritmusok, adatszerkezetek I.* című irodalom is [6]), illetve én sem kedvelem, kihagytam a nyelvből.

Az iterációs lehetőségek (elől tesztelő ciklus) esetén feltüntettem a vezérlés folyamatát is, hogy működésük könnyebben megérthető legyen.

4.9. I/O kezelés

A nyelv lehetőséget ad bemeneti és kimeneti adatok kezelésére. Az erre a célra használt utasítások a **beolvas**, **beolvas: kiír**, **kiír**:. A Lefoglalt szavak fejezetben már kifejtettem, hogy miért létezik azonos célra több utasítás. Használata további magyarázatot nem igényel, a nyelv definíciójából kideríthetők.

4.10. Tömbkezelés

A nyelvben definiálhatók tömbök. A tervezés során úgy éreztem, hogy a többdimenziós tömbök implementálásának nehézségi szintje túlmutat a szakdolgozatom keretein, így a nyelv csak az egyszerű típusú, egydimenziós tömböket támogatja.

A tömbök létrehozásának szintaktikája a *Sergyán Szabolcs – Algoritmusok, adatszerkezetek I.* című jegyzetből [6] származik. A jegyzet szerzőjétől kapott tanács alapján a tömbök típusának leírása: **típus[]**.

Az általam definiált nyelvben (a fent említett jegyzetben használt pszeudonyelvvel ellentétben) a tömbindexelés nullától történik. Ezt azzal indoklom, hogy a manapság (2016) használatos leggyakoribb nyelvek nullától indexelnek: az IEEE SPECTRUM által köz-
zétett *The 2016 Top Programming Languages* [7] cikkben megjelölt top 10 nyelv közül legalább 9 nullától indexel (az R nyelv esetén nem találtam egyértelmű leírást ezzel kapcsolatban).

5. LEXIKÁLIS ELEMZŐ

Ebben a fejezetben részletezem a lexikális elemző elkészítésének folyamatát.

5.1. Tervezés és implementáció

5.1.1. Lexikális elem kódok

A program működéséhez szükséges egy egységes adatszerkezet, amelyből futási időben ki tudja olvasni az adott lexikális elemhez tartozó kódot. Ezt a legjobb egy hash táblában tárolni, amelyben a lexikális elem nevéből a hozzá tartozó kódba képezünk; ugyanis a hash táblák jellegzetessége, hogy a beszúrás és a kulcs alapján történő keresés $O(1)$, az érték alapján történő keresés $O(N)$ nagyságrendű (azonban ez is lecsökkenthető $O(1)$ -re, egy inverz hash tábla bevezetésével). Ez a gyorsaság jól megfelel a céljaimnak. A kódokat tároló hash táblát mindenhol el kell tudnom érni, és csak egy példány szükséges belőle, ezt ennek megfelelő helyen és módon kell majd tárolnom.

A program működését megkönnyítve bizonyos kódcsoportokat képeztem ugyanis a feldolgozás során hasznos lehet, hogy egy kódról megmondjuk, hogy az milyen típusú lexikális elem. Így a következő kódcsoportokat alkottam meg:

1. Kulcsszavak
2. Literálok
3. Operátorok
4. Beépített függvények
5. Típusok

Említésre méltó továbbá, hogy kódot kell adni a következőknek is: *azonosító*, *újsor*, *hiba* (ez a feldolgozást könnyíti meg). A nyelvdefinícióból következően bizonyos kulcsszavak azonos jelentést hordoznak; ezeket úgy kezeltem le, hogy minden jelentés azonos számkódra mutat.

A kódokat egy erre szolgáló osztály tartalmazza, a konkrét adatszerkezet pedig egy `Dictionary<string, int>`: a kulcs a lexikális elem neve (string), az érték pedig a hozzá tartozó kód (int).

A fent említett inverz adatszerkezet egy `Dictionary<int, string>`, amiből a kódhoz tartozó nevet lehet konstans idő alatt elkérni. Habár több névhez is tartozhat ugyanaz a kód (pl. `kilép` és `kilépés` utasítások), az inverz tábla csak egyet adhat vissza ezek közül; ez általában a két forma közül a rövidebb/egyszerűbb.

5.1.2. Kimeneti szimbólumsorozat

Az elemző kimenetének tartalmaznia kell minden, a további feldolgozáshoz szükséges lexikális elemet formálisan felírva. Nevezzük egy lexikális elem leírását *token*nek. Minden tokennek tartalmaznia kell a hozzá tartozó szimbólum kódját, valamint a token típusától függően további információkat, pl. literál értéke, változó szimbólumtáblabeli kódja stb. Ezeket listában érdemes tárolni (nem tudjuk előre, hogy hány darab token lesz és a sorrendiség is fontos), és az elemzés során az aktuális lexikális elem felismerése után az adott elemből tokent készítve, hozzáfűzöm a listához.

A konkrét implementációban definiáltam egy absztrakt **Token** osztályt, amiből a szintén absztrakt **TerminalToken** osztály származik. Ennek leszármazottjai a lexikális elemző tokenjei, azaz:

1. **ErrorToken**
2. **IdentifierToken**
3. **InternalFunctionToken**
4. **KeywordToken**
5. **LiteralToken**

Az **ErrorToken** azt jelzi, ha egy részszovegből nem sikerült felismerni egy elemet sem.

Így az elemző kimenete egy **TerminalToken** típusú lista adatszerkezet lesz.

5.1.3. Szimbólumtábla

A szimbólumtábla tárolásra számos lehetőség közül választhattam (e lehetőségek megtekinthetők az 1.2.3. fejezetben). Olyat szerettem volna választani, amely számomra jól értelmezhető, és amelyet könnyen tudok implementálni. Mivel pszeudonyelvem megkülönböztet blokkokat, az egyszerű listaként való tárolás nem megfelelő. Úgy találtam, hogy a verem szerkezetben a blokk kezelés számomra nem olyan jól vizualizálja a kód struktúráját; valamint a hash szerkezet pedig további információk letárolását igényli, így valószínűsíthető, hogy implementációja is nehezebb.

Végül egy olyan szerkezetet találtam ki, amely leginkább a fa struktúrához hasonlít. Definiáltam egy **SymbolTableEntry** absztrakt osztályt, amely jelképez egy bejegyzést. Ennek két leszármazottja van, **SymbolTable** és **SingleEntry**, A **SymbolTable** egy altáblát, a **SingleEntry** pedig egy azonosítót tárol. A **SymbolTable** osztály tartalmaz egy **SymbolTableEntry** típusú listát. Így egy „gyökér” **SymbolTable** objektumban eltárolható az egész szimbólumtábla.

5.1.4. Elemző

Az eddigi alfejezetek az elemző adatstruktúráját taglalták, ez pedig a működést mutatja be. A működés részt számos osztály írja le. Ezek közül (a könnyebb megértés érdekében) néhány vázlatos felépítése a következő:

- **LexicalAnalyzer**: Az elemző fő része; ez végzi az elemzést, vezérli a többi osztályt. Ezen osztály megfelelő metódusai alkotják a lexikális elemző publikus interfészét.
- **LexicalAnalyzerResult**: A lexikális elemző kimenetét reprezentáló adatszerkezet, amely egy **TerminalToken** listát és egy szimbólumtáblát tartalmaz.
- **LexicalAnalyzerState**: Az elemző által megvalósított állapotgép állapotait tároló felsorolás típus.
- **OutputTokenListHandler**: A kimeneti listát kezelő osztály, példányként létezik a **LexicalAnalyzer** osztályban.
- **SymbolTableHandler**: A szimbólumtáblát kezelő osztály, példányként létezik a **LexicalAnalyzer** osztályban.
- **LexicalElementIdentifier**: Megadja, hogy egy karakterlánc milyen lexikális elemnek felel meg.

Az elemzés folyamata a következő:

1. Meghívják egy **LexicalAnalyzer** példány publikus elemző metódusát, amely majd az elemzés végeztével visszatér a **TerminalToken** listával.
2. Némi bemenet ellenőrzés után megtörténik a lényegi elemzés, majd lezajlik a szimbólumtábla tisztítása (üres levél szimbólumtáblák törlése) és frissítése a visszatérés előtt.
3. Az elemző kiválasztja a kezdeti állapotot, majd amíg az aktuális állapot nem a végállapot, végrehajtja az állapotnak megfelelő metódust, majd új állapotot keres.

Az elemző a következő állapotokat definiálja:

- **Initial**: kezdeti állapot
- **Comment1Line**: egysoros kommentet lekezelő állapot
- **CommentNLine**: többsoros kommentet lekezelő állapot
- **Whitespace**: üres karaktereket (szóköz, tabulátor, újsor) lekezelő állapot.
- **StringLiteral**: szöveg literálokat lekezelő állapot
- **NonWhitespace**: további lexikális elemeket lekezelő állapot
- **Final**: végállapot

A **NonWhitespace** állapot kezelése bonyolultsága révén egy külön osztályba került ki (**NonWhitespaceRecognizer**) és eredménye egy külön objektum (**NonWhitespaceRecognitionResult**).

4. Az állapottól függően a következő metódusok futhatnak le, leírásuk:

- a. **Whitespace()**: tovább halad a bemeneti szövegen a következő nem üres karakterhez.
- b. **StringLiteral()**: idézőjel (") észlelése esetén lép érvénybe. Elmegy az adott szöveg végéig, majd hozzáfűzi a kimeneti token listához a literálhoz tartozó tokent.
- c. **Comment1Row()**: két darab perjel (//) észlelése esetén lép érvénybe. Elmegy az adott sor végéig.
- d. **Comment1Row()**: egy perjel és egy csillag (/*) észlelése esetén lép érvénybe. Elmegy az adott többsoros komment végéig (amíg nem észlel */ karaktersorozatot).
- e. **NonWhitespace()**: a legbonyolultabb rész, ez kezeli az összes többi lexikális elemet; szöveg (ami nem szöveg literál) észlelése esetén lép érvénybe. és megkísérli felismerni az onnan kezdődő lehető leghosszabb lexikális elemet (úgy, hogy sorban veszi a jelenlegi pozíciótól számított 1,2,3,...n hosszú részszovegeket). Kezeli az ütközéseket az egész és tört literálok között. Eközben építi a szimbólumtáblát és bővíti a kimeneti token listát. A rész alaposabb megértése a kód és működés mélyebb ismeretét igényli, így ezt jobban itt nem részletezem.

5.2. Tesztelés

A lexikális elemző tesztelését a szakdolgozat készítésének korábbi fázisában kézzel végeztem (összehasonlítottam bemeneti és kimeneti fájlokat), azonban idővel rájöttem (és teljes állású szoftverfejlesztői munkahelyemen megtanultam), hogy minden fontosabb működést automatikus (azaz egy kattintásra „futtatható”) tesztekkel kell ellátni, így amennyiben változtatás kerül a kódba, megfelelő tesztek esetén könnyedén ellenőrizhető, hogy a működés továbbra is helyes.

5.2.1. Tesztelési keretrendszer kiválasztása

Automatikus tesztek futtatásához szükség van egy tesztelési keretrendszerre, és persze célszerű az implementáció összes fázisához ugyanazt a keretrendszert használni. A következő néhány bekezdésben megvizsgálom a választható alternatívákat, és megteszem a választásomat.

A Visual Studio 2017-ben integráltan megtalálható a Microsoft tesztelési keretrendszere, a Visual Studio Unit Testing Framework, rövidebb nevén MSTest (ez valójában a futtatható, parancssoros segédprogram neve, de ilyen néven ismert a fejlesztők körében). Bármiféle telepítés nélkül használható. Korábbi tapasztalataim alapján (és munkatársaim szerint is) ez a keretrendszer teljesítménybeli problémákkal küszködik; sok teszt futtatása

esetén lassú a végrehajtás. Habár nagyszerűen alkalmas a tesztelés alapjainak elsajátítására, sok olyan funkció hiányzik belőle, amely nagyobb szoftverek fejlesztésekor nagyon hasznos tud lenni, pl.: párhuzamos tesztfuttatás, dinamikusan paraméterevezhető tesztek.

A másik két nagyobb alternatíva az NUnit és az xUnit. Az NUnit eredetileg a népszerű Java-s tesztelési keretrendszer, a JUnit átirata .NET-re, azonban a nagy érdeklődés miatt teljesen újraírták, hogy jobban illeszkedjen a keretrendszerhez. Az xUnit-ot pedig az NUnit ihlette; és azt további innovatív gondolatokkal bővítette, mint például „Fact” és „Theory”. Az utóbbi egyik negatívuma a szűk dokumentáció, így nehezebb lehet a keretrendszer elsajátítása.

Tekintettel arra, hogy:

- jelenlegi munkahelyemen már alaposan megismerhettem az NUnitot és lehetőségeit,
- az xUnit véleményem szerint nem nyújt olyan előnyöket az NUnit-tal szemben, amit a szakdolgozatom írása során ki is tudnék használni,
- az xUnit megtanulása a kevés dokumentáció miatt valószínűleg nehezebb lenne,
- az NUnit támogatja a párhuzamos tesztfuttatást és a tesztek dinamikus paraméterevezését,

választásom az NUnit-ra esik.

Szakdolgozatom implementálására az NUnit 3.8.1-es (jelenlegi legfrissebb) verzióját használtam.

5.2.2. A lexikális elemző tesztelése

A lexikális elemző tesztelésekor fontosnak tartottam a kód tesztek általi minél nagyobb lefedettségét és így a helyes működés minél jobb bizonyítását, ezért rengeteg tesztet készítettem.

A tesztek írása során észrevettem, hogy szinte állandóan nagyon hasonló a kódsorokat írok le, így elkészítettem a **TokenTester** és **SymbolTableTester** osztályokat, amelyek segítségével könnyen megfogalmazható lett az elvárt működés, és rengeteg kódDuplikációtól kíméltem meg magam.

A lexikális elemzés folyamatát kétféleképpen teszteltem:

- az elemző egyes részeit egymástól minél inkább izoláltabb környezetben
- az elemzést összesítve, komplex tesztekkel.

Itt szeretném kiemelni az NUnit egyik hasznos funkcióját, amellyel adatorientáltan tudtam tesztelni a szöveg literálok felismerését. Az NUnit definiál egy **TestCaseSource** nevű, metódusokra alkalmazható attribútumot. Ez az attribútum paraméterként fogad egy nevet,

amely nevű statikus vagy konstans változónak arra az objektum tömbre kell mutatnia, ami a teszt bemeneti adatait tartalmazza. Ezt mutatja be az 5.1. ábra.

```
private static readonly string[] StringLiterals =
{
    EmptyString, Newline, EscapedQuotationMark, EscapedBackslash,
    EscapedNewline, EscapedTab, OneDot, OneSpace, SimpleWord, Sentence,
    HungarianCharacters, SpecialCharacters, EverySimpleKeyOnHungarianKeyboard, Complex
};

[TestCaseSource(nameof(StringLiterals))]
public void CanRecognizeSzovegLiterals(string value)...
```

5.1. ábra - Az nUnit TestCaseSource attribútuma

Megjegyzés az ábrához: a **StringLiterals** tömbben található elemek string típusú konstansok, amik a teszt adatokat tartalmazzák.

5.2.3. Néhány fontosabb teszteset

A szakdolgozat terjedelmi határai nem engedik meg az összes teszteset tételes és konkrét felsorolását, így a tesztelési fázisok dokumentálásában csak néhány, fontosabb tesztesetet emelek ki.

A lexikális elemzés esetében ezek a következők:

- **CheckValidNamingConventions:** ez a paraméterezett teszt (paraméterként a tesztelendő azonosító neve érkezik) azt vizsgálja, hogy a megadott azonosító megfelel-e az azonosítók elnevezési követelményeinek. A paraméterek egy stringeket tároló tömbből érkeznek, ezeket az nUnit automatikusan dolgozza fel és osztja szét külön tesztesetekre. Ennek párja a **CheckInvalidNamingConventions**, amely a helytelen eseteket kezeli.
- **CanRecognizeKeyword:** szintén paraméterezett teszt, amely vizsgálja, hogy a megadott kulcsszót helyesen ismeri-e fel az elemző, és megfelelően jelenik-e meg a kimeneti token listában.
- **CanRecognizeIfThenElse:** a szelekció felismerését ellenőrző teszt. Ehhez hasonlóan olyan tesztesetet is írtam, amely a ciklusokat ellenőrzi (**CanRecognizeWhile**).
- **Declarations:** Az összes fajta változódeklarációt ellenőrző teszteset. Ez már a komplex tesztek közé tartozik, amelyek nevükből adódóan komplexebb kódokat tesznek próbára. További komplex tesztesetek ellenőrzik a kommentezést, valamint a kifejezések és a beépített függvények felismerését; illetve készítettem egy egyszerű, másodfokú egyenletmegoldó programkódot is.

Készültek tesztek a következő (lexikális elemzést segítő) osztályokra is, azonban ezeket nem részletezem:

- **LexicalElementCodeDictionary**
- **SymbolTable**

5.3. Fejlesztési tapasztalatok

Már az implementáció során is sikerült bizonyos gyorsításokat eszközölni a programban:

- szelekciók switch-case szerkezetre cserélése
- listák hash-táblákra való cseréje
- ciklusok és metódushívások optimalizálása
- szelekciók vizsgálati sorrendjének valószínűségi alapon történő felcserélése.

A lexikális elemző implementációja két félévén keresztül történt; és ez idő alatt sokat fejlődtem a programozás terén, többek között a munkahelyemnek köszönhetően. A programkód szerkezete korai állapotában nem volt jól áttekinthető, így többször is refaktoráltam; az ebbe fektetett munka mennyiségét is jelzi az, hogy becslésem szerint összesen 50 órát töltöttem el csak annak érdekében, hogy jobban átlátható szerkezetű és tesztelhetőbb legyen a lexikális elemző.

Megjegyzésre méltó a tesztek hasznossága is. Habár a tesztek megírása több tíz órámba telt, a befektetett idő megtérült, ugyanis így biztonsággal állíthatom, hogy a lexikális elemző pontosan úgy működik, ahogyan én azt elvárom; illetve a tesztek írása közben is felismertem néhány hibát. A lexikális elemzőhöz összesen 123 tesztet készítettem, amelyek nagyjából 850 sor kóddal fedik le több, mint 99% arányban az elemző nagyjából 400 sornyi kódját.

6. SZINTAKTIKUS ELEMZŐ

6.1. Tervezés

A szintaktikus elemző tervezése ígérkezett a legnehezebb feladatnak a szakdolgozatom implementációja során, és arányait tekintve ezzel a fázissal töltöttem el a legtöbb időt. Rájöttem, hogy precíz módon (például Backus-Naur formában) definiált szintaktikus nyelvtan nélkül rendkívül nehéz és nem is érdemes belefogni az implementációba, így – habár korábban ezt el szerettem volna kerülni – meg kellett alkotnom egy tényleges nyelvtant.

6.1.1. Elemzőtípus választása

A különböző elemzőtípusokat tipikusan különböző módokon szokás implementálni. Céлом az volt, hogy a lehető legkönnyebben érthető és implementálható megoldást válasszam.

„A kézzel implementált parser-ek gyakran LL nyelvtanokat használnak (...). Az LR-nyelvtanokat (...) általában automatizált eszközökkel szokás elkészíteni.” [3]

Így kézenfekvő megoldás volt, hogy egy LL-nyelvtant alkossak; ehhez pedig jobbrekurzív szabályokra volt szükségem.

6.1.2. Megadási forma választása

Egy nyelvtan formális felírásának módszereit már elemeztem a 4.1. fejezetben, most pedig választanom kellett egyet ezek közül.

Annak érdekében, hogy a nyelvtan a lehető legkönnyebben legyen átültethető az implementációba, érdemes volt minél kevesebb rövidítést, absztrakciót tartalmazó formát választanom. Tekintettel arra, hogy az ABNF és az EBNF is a BNF bővítései/kiterjesztései, egyértelmű, hogy a BNF-et volt érdemes választanom.

A BNF a következő, egyszerű szintaxissal rendelkezik:

<szimbólum> ::= kifejezés

ahol a **kifejezés** egy vagy több további szimbólumot vagy terminális jelet tartalmazhat, amely utóbbit idézőjelek között kell megadni. Továbbá választási lehetőségekre használható a függőleges vonal karakter (|) is, amit a kifejezések közé lehet illeszteni.

Vegyük példaként az egyik korábbi fejezetben említett nyelvtant:

- 1) $E \rightarrow E + T$
- 2) $E \rightarrow T$
- 3) $T \rightarrow T * F$

4) $T \rightarrow F$

5) $F \rightarrow \text{id}$

Ez a nyelvtan a következőképpen írható le BNF-ben:

$\langle E \rangle ::= \langle E \rangle \text{ "+" } \langle T \rangle \mid \langle T \rangle$

$\langle T \rangle ::= \langle T \rangle \text{ "*" } \langle F \rangle \mid \langle F \rangle$

$\langle F \rangle ::= \text{"id"}$

Megjegyzés: a BNF áll a legközelebb a formális nyelvek és gépek tantárgyban használt általános nyelvtanmegadási formához, így érthető lehet bárki számára, aki részt vett ilyen kurzuson.

6.1.3. A szintaktikus nyelvtan definiálása

A szintaktikus nyelvtan definiálásának lépéseit a szakdolgozat terjedelmi limitációi miatt csak vázlatosan dokumentálom.

Mielőtt hozzákezdtem a szabályok megalkotásához, két nyelv nyelvtanát tekintettem át részletesebben:

- **Oberon:** a 4.1 fejezetben említett, konzulensem által ajánlott nyelv. Viszonylag egyszerű nyelvtana jó kiindulási pontot biztosított ahhoz, hogy el tudjam kezdeni a sajátomat.
- **C#:** mivel a nyelvemet a C# ihlette, át tudtam venni néhány ötletet a szabályokhoz.

Az elkészített nyelvtanban a rekurziót a következő nemterminálisok biztosítják:

- **<Állítások>:** egy program állítások sorozata. Ezt a nemterminálist illesztettem be azon szabályok jobb oldalára is, amelyek beágyazott állításokat tartalmazhatnak, ezek a szelekció (ha) és iteráció (ciklus_amíg).
- **<Operandus>:** egy bináris kifejezés két oldalán operandusok állnak; tömbindexelés esetén a tömbindex lehet egy másik operandus is.

A kifejezéseket kezelő szabályokat sajnos nem sikerült rekurzívra megalkotni, főleg a sok operátor különböző precendenciái (fontossági sorrendjei) miatt. Megjegyzendő, hogy a bonyolultabb szabályok a szemantikus elemző munkáját jelentősen megnehezítették volna.

Így a kész nyelvtan rövid, kompakt, és – véleményem szerint – könnyedén érthető lett; ami a megalkotása közben fontos szempont volt.

6.2. Implementáció

A szintaktikus elemző implementációját két lépében végeztem el: először egy olyan vázat készítettem el, amely a szabályokon kívül mindent tartalmaz; később pedig beleillesztettem ebbe a szabályokat.

6.2.1. A szintaxisfa

A szintaxisfát kezelő három osztály működése röviden a következő:

1. **ParseTree<T>**: generikus osztály, amely egyben leírja a fa struktúráját és végzi a felépítést a **StartNode**, **EndNode** és **RemoveLastAddedNode** nevű metódusokkal. Tartalmaz egy referenciát (**Root**) a gyöker elemre. A szintaxisfa ténylegesen egy **ParseTree<Token>**.
2. **TreeNode<T>**: a fa egy csúcsát reprezentáló objektum. Három tulajdonsággal rendelkezik:
 - a. **Value**: A csúcshoz tartozó érték.
 - b. **Parent**: A csúcs „szülő” csúcsa; gyökérem esetén ez nincs kitöltve (a T típus alapértelmezett értéke).
 - c. **Children**: A csúcs „gyermek” csúcsait tároló lista. (A használt adatszerkezet az **IList<T>**, amelyben számít a sorrend.)
3. **ParseTreeExtensions**: Segédmetódusok **ParseTree<Token>** objektumokhoz (ún. *extension method*-ok, magyarul kiterjesztett metódusok).

A fa kiépítése a **ParseTree** objektumon definiált metódusokkal egyszerűen megtehető.

Megjegyzés: amikor az elemzőnek egy produkciós szabály alkalmazása nem sikerül, a szintaxisfának szerkezetileg vissza kell állnia arra az állapotra, ahol még a szabály nem volt alkalmazva (ugyanis a fa szabályok alkalmazásával párhuzamosan épül); ez az objektum memóriaszintű másolásával és visszaállításával történik meg.

6.2.2. Az elemző alapja

Az elemző alapja definiálja az alapl működést és azt a megadási formát, amellyel a szabályok nagyon egyszerűen megadhatók.

A fent említett forma a következő két metódussal valósítható meg:

- **bool Match(params Func<bool>[] predicates)**
- **bool Rule(Func<bool> predicate)**

A **Match** egy tetszőleges számú (de legalább egy) logikai típust visszaadó függvényreferenciát váró metódus, amely megkísérli sorban végrehajtani a megadott függvényeket (amik valójában a produkciós szabályok jobb oldalának elemei). Amennyiben az egyik

átadott függvény hamissal tér vissza (azaz az adott produkciós szabályra nem illik a bemenet), az egész metódus sikertelenné válik és hamissal tér vissza. Sikertelen illesztési próba esetén a szintaxisfát, valamint a bemeneti token listát indexelő számértéket visszaállítja a kiindulási állapotba.

A **Rule** metódus arra szolgál, hogy egy helyen kezelje az azonos bal oldalú produkciós szabályokat, azaz tulajdonképpen a BNF-ben meghatározott **|** operátor szerepét implementálja.

A két metódus együttes használatával egyszerűen leírhatók a nyelv szintaktikus szabályai; erre a következő fejezetekben láthatók példák.

6.2.3. Statikus szabályok

Statikus szabály elnevezéssel illetem az olyan produkciós szabályokat, amelyek nem rekurzívak, és nagyon közel állnak a terminálisokhoz (közvetlenül vagy majdnem közvetlenül terminálisokra hivatkoznak).

Ezeket azért hasznos elkülöníteni, mert nem függenek a szintaktikus szabályoktól, és elszeparáltan tárolhatók a többi szabálytól.

A következő szabályok statikusak:

- **AlapTípus**
- **TömbTípus**
- **IoParancs**
- **BelsőFüggvény**
- **UnárisOperátor**
- **BinárisOperátor**

6.2.4. Dinamikus szabályok

Az előző, statikus szabályokkal szemben dinamikus szabályoknak neveztem el azokat a szabályokat, amelyek jobb oldalán is megtalálható nemterminális. Valójában ezek a szabályok teszik ki a nyelvtan lényegi részét.

A korábban említett egyszerű megadási formára példaként az **Operandus** nevű nemterminálishoz tartozó szabályok leírási módját mutatom be, ez látható a 6.1. ábrán.

```

internal bool Operandus()
{
    return Rule(() =>
        Match(UnárisOperátor, Azonosító)
        || Match(UnárisOperátor, Literál)
        || Match(Azonosító, () => T("[", Operandus, () => T("]"))
        || Match(Azonosító)
        || Match(Literál));
}

```

6.1. ábra - Az Operandus nemterminálishoz tartozó szabályok implementációja

Összehasonlításként íme a szabály BNF-ben:

```

<Operandus> ::=    <UnárisOperátor> "azonosító"
                  | <UnárisOperátor> "literál"
                  | "azonosító" "[" <Operandus> "]"
                  | "azonosító"
                  | "literál"

```

Látható, hogy az implementáció leírási módja erősen hasonlít a BNF-hez.

Megjegyzés: a **T** metódus terminálok illesztésére használatos.

6.2.5. Hibakezelés

Az elemző fontos része, hogy amennyiben szintaktikus hibát talál a kódban, azt minél pontosabban jelezze a programozó számára.

Amikor a szintaktikus elemző úgy véli, hogy szintaktikai hiba van a programban, dob egy kivétel objektumot (**SyntaxAnalysisException**), amelyben megtalálhatók egy hiba fel-derítéséhez szükséges adatok:

- **LastToken**: az a terminális token, amit a bemenetről a legutoljára adott hozzá a fához.
- **CurrentLine**: az aktuálisan elemzett kódsor száma a hiba fellépésekor.
- **FurthestLine**: a szintaktikus elemzés során elért legtávolabbi sor.

Az objektum így behatárolja, hogy a megtalált szintaktikai hiba biztos, hogy nem a **FurthestLine** számú sor után található, és vélhetően a **CurrentLine** számú sorban, vagy ahhoz nagyon közel van.

Az elemzőt – a lexikális és szemantikus elemzővel szemben – sajnos csak úgy sikerült megalkotnom, hogy az elemzés hiba esetén azonnal megszakad; azonban ezt nem érzem

olyan nagy problémának, ugyanis még a legmodernebb fordítók sem működnek tökéletesen ilyen téren.

6.3. Tesztelés

A szintaktikus elemző tesztelése nehéz és időigényes feladat volt, ugyanis minden bemeneti adathoz (kódhoz) le kellett ellenőrizni, hogy a teljes szintaxisfa minden csúcsa a megfelelő helyen van-e és csak azok a csúcsok szerepelnek-e, amelyeknek kell.

A tesztek nagy része a következő részekre oszlik:

1. Bemeneti kód átadása a lexikális elemzőnek, és a szintaktikus elemző lefuttatása a lexikális elemző kimenetével.
2. Annak ellenőrzése, hogy a szintaxisfa levelei sorban a bemeneti nemterminálisokat adják-e. Ezt viszonylag egyszerűen ellenőrizni tudtam, ugyanis az implementációs fázisban szándékosan ilyen célra készítettem egy metódust, amivel lekérhetők a fa levelei.
3. A szintaxisfa kézzel való bejárása és minden csúcs teljes ellenőrzése.

A harmadik lépés megkönnyítése érdekében számos segédmetódust készítettem (ezek a **SyntaxAnalysisTests** projekt **TestHelper** osztályában találhatóak), azonban még így is kihívást jelentő feladat volt ennek a (valójában a teszt nagy részét kitevő) fő résznek a megírása, ugyanis közben fejben követnem kellett, hogy pontosan melyik szabálynál jár a szintaktikus elemző.

A szintaktikus elemző 300 sorához 70 db. tesztet írtam (a tesztek összesen 650 sort tesznek ki); és a lexikális elemzőhöz hasonlóan a szintaktikus elemző kódjának is több, mint 99%-át lefedtem tesztekkel.

A következő teszteseteket emelném ki a szintaktikus elemző tesztjeiből:

- **TömbLétrehozóKifejezés:** A tömbök létrehozásának módszerét ellenőrző teszt; pontosabban a következő kódot elemzi:

```
program_kezd  
tört[] c = létrehoz[99]  
program_vége
```
- **VáltozóDeklaráció** névvel kezdődő tesztek: ez a négy teszt ellenőrzi a változók deklarációját:
 - o alap típus „nem tömblétrehozó kifejezéssel” létrehozva
 - o alap típus beépített függvény használatával létrehozva
 - o tömb típus azonosítóval létrehozva
 - o tömb típus tömblétrehozó utasítással létrehozva
- **SimpleTheorem_Summation:** a szintaktikus elemzőhöz tartozó egyetlen komplex teszt, ami egy összegzés tételt valósít meg.

6.4. Fejlesztési tapasztalatok

A szintaktikus elemző elkészítése komplexebb feladatnak bizonyult, mint ahogyan azt előtte gondoltam. A legnehezebb része a nyelvtan formális felírása volt, ugyanis kezdetben hetekig próbálkoztam egy olyan nyelvtan kialakításán, ami rekurzív kifejezéseket kezel. Később rájöttem, hogy ezt egyedül nem fogom tudni időben megcsinálni, így azt a döntést hoztam, hogy a nyelv csak egyszerű kifejezéseket támogat majd.

A megalkotott egyszerű leírási mód rendkívül leegyszerűsíti egy adott nyelvtan tényleges implementációját. Valójában közel voltam ahhoz, hogy egy BNF-formában megadott nyelvtant tartalmazó fájl alapján generáljam ki a szabályok implementációját; azonban ezt az ötletet elvettem, mert – habár nagyon jól hangzott, – feleslegesen csináltam volna magamnak többletfeladatot; révéen fordítóprogramot írok, nem pedig fordítóprogram-generátort.

7. SZEMANTIKUS ELEMZŐ

7.1. Tervezés

7.1.1. A szemantikus elemző feladatainak meghatározása

A szemantikus elemző feladatait az 1.2. fejezetben részleteztem. Itt megvizsgálom, hogy milyen feladatok maradtak még a szemantikus elemzőnek.

Típus ellenőrzés: eddig semmi ilyen jellegű vizsgálat nem történt, így meg kell majd vizsgálnom a következőket:

- egy kifejezés két oldalán megfelelőek-e a típusok
- az operandusokra alkalmazott operátorok kompatibilisek-e egymással
- a belső függvények megfelelő típusú bemeneti paraméterrel vannak-e ellátva

Tömbindexek ellenőrzése: itt is csak típus vizsgálat szükséges, a tömbből való kiindulás a kódgenerálás vagy futás során derülhet ki.

Hatókör ellenőrzés: a lexikális elemző már minden szükséges hatókör elemzést elvégzett.

Függvénynév-túlterhelés ellenőrzés: mivel a nyelvem nem definiál függvényeket, erre az ellenőrzésre nincs szükség.

Változók hiányzó vagy többszörös deklarációja: ezt elvégezte a lexikális elemző.

Látható, hogy a szemantikus elemzőnek így csak a típusokat kell ellenőriznie.

7.1.2. Működési elv

Annak érdekében, hogy a típusellenőrzést el lehessen végezni, az elemzőnek a következő bemenetekre van szüksége:

- **szintaxisfa:** a szintaktikus elemző által átadott szintaxisfát bejárva lehet majd eldönteni, hogy a kifejezéseknél típusok megfelelőek-e.
- **szimbólumtábla:** a lexikális elemző által megalkotott szimbólumtáblára azért van szükség, mert a definiált azonosítók típusát itt tárolja a fordítóprogram.

Ezen adatok birtokában elkezdődhet a szemantikus elemzés, ami a következő sorokban leírt módon történik:

A program felfogható állítások sorozataként, és a típusok szempontjából az állítások egymástól függetlenek. Így elegendő egy állítás ellenőrzését megvalósítani, majd ezt az összes állításra alkalmazni. Ebből az is következik, hogy amennyiben szemantikai hibát talál az elemző, nem kell kilépnie, folytathatja az elemzést a következő állítással.

Egy állítás elemzését az „oszd meg és uralkodj” elv alapján valósítottam meg, azaz addig bontottam a feladatot részfeladatokra, amíg egy részfeladatot már viszonylag egyszerűen meg tudtam valósítani. Ennek érdekében például az <Állítások> produkciós szabály ellenőrzése az azt tartalmazó <Állítás> és esetlegesen <Állítások> szabályok ellenőrzéséből áll (az utóbbi természetesen rekurzív hívás), és így tovább a többi szabállyal. Így végül eljutunk a terminálisokat is tartalmazó szabályokhoz, amiket tovább nem kell részekre bontani.

7.2. Implementáció

7.2.1. Az implementáció részei

A szemantikus elemző implementációját a következő részekre bontottam:

SemanticAnalyzer: magas szintű logikát megvalósító osztály. Tárolja a **TypeChecker** és **ExceptionCollector** objektumokat, valamint a szintaxisfát. Azon nyelvtani szabályok ellenőrzése található itt, amelyeknél nem szükséges közvetlen típusellenőrzés, például:

- Állítások
- Állítás
- VáltozóDeklaráció
- Értékadás
- IoParancs

A konkrét típusellenőrzést pedig a **TypeChecker** objektummal végezteti el.

ExceptionCollector: szerepe a szemantikus elemzés során felmerült hibák gyűjtése, és az elemzés végén azok eldobása egy aggregált kivétel (**AggregateException**) formájában.

TypeChecker: a típusellenőrzést végző osztály. Olyan szabályokat kezel, amelyeknél már szükség van típusellenőrzésre; és a kifejezések, azonosítók és további elemek típusait felderítéséhez a **TypeFinder** osztályt használja. A **TypeChecker** osztály deríti fel az elemző hibáinak jelentős részét.

TypeFinder: a típusok felderítésére képes osztály; munkáját a **StaticTypeFinder** osztály segíti, amely olyan elemek típusát biztosítja, amikhez nem szükséges sem a szimbólumtábla, sem a szintaxisfa szerkezete. Bizonyos esetekben a típusok felderítése közben is szükség van típusellenőrzésre, például egy összeadás „kimeneti” típusa a **+** operátor két oldalán szereplő operandusok típusa; azonban ez csak akkor deríthető ki, ha a két oldal típusa megegyezik. Az ilyen ellenőrzéseket az objektum a hozzá csatolt **TypeChecker** objektummal végezteti el (ami egyben a tárolója is).

7.2.2. Hibakezelés

A hibakezelés az előző fejezetekben leírtak szerint működik. Az implementációban háromfajta szemantikus hibatípust definiáltam:

SemanticAnalysisException: a másik két kivétel típus öse. Ide tartoznak az olyan hibák, amelyek a másik két kivétel típusra nem illenek, és annak sem láttam szükségét, hogy külön kivétel típusba kerüljenek. Egy **Line** nevű, egész szám típusú adatot definiál, ami azt adja meg, hogy a kivétel melyik sorban keletkezett.

AnotherTypeExpectedException: azokban az esetekben, amikor a szemantikus elemző nem olyan típust talál, amilyet vár, ilyen kivételt dob. Két adattagja az **Expected** és az **Actual**, amelyek nevükből eredően azt adják meg, hogy mi volt az elvárt és a tényleges típus neve. Megjegyzendő, hogy az elemzés során ez a leggyakoribb hibatípus.

TypeMismatchException: amikor egy kifejezés két oldalán eltérő típusokat talál az elemző, ezt a hibát adja vissza. Két adattagja a **Left** és a **Right**, amelyek megadják, hogy a kifejezés bal és jobb oldalán mik voltak a típusok.

7.3. Tesztelés

A szemantikus elemző tesztelését két részre bontottam: negatív és pozitív tesztekre. A pozitív tesztek olyan kódokat elemeznek, amelyek nem tartalmaznak hibát, és így nem szükséges bővebb ellenőrzés. A negatív tesztek kódsoraiban legalább egy helyen szemantikai hiba van, és a tesztek ezeket a hibákat várják el és a dobott hibákat ellenőrzik.

A negatív tesztekhez – a korábbi fázisokhoz hasonlóan – készítettem egy tesztelést segítő osztályt, amely az adott kódhoz tartozó lexikális, szintaktikai és szemantikai elemzést végzi el, valamint a hibák (dobott kivételek) ellenőrzését könnyíti meg.

Így egy helytelen kód tesztelése két lépésből állt:

1. Kód feldolgoztatása úgy, hogy az esetleges hibát dobás helyett visszaadja a tesztelő metódus (erre készült egy segéd metódus).
2. A visszatért hibákat ellenőrizni, hogy a megfelelő hibát jelezte-e az elemző és a megfelelő helyen (sorban).

Az elemzőhöz írt 123 teszt 300 sor kódból áll, ami a szemantikus elemző implementációjának kb. 250 sorát több, mint 99%-ban lefedi.

A következő (negatív) teszteseteket emelném ki:

- **BinaryOperatorCompatibility** névvel kezdődő tesztek: ez a négy teszteset azt teszteli, hogy ha egy bináris operátort nem támogatott típusú operandusok között próbálunk használni, akkor megfelelően jelez-e hibát a szemantikus elemző.

- **If1_Negative** névvel kezdődő három teszt: ezek a tesztek azt ellenőrzik, hogy ha szelekcióban a kiértékelendő feltétel nem logikai típusú, akkor ezt megfelelően jelzi-e az elemző.
- **Complex**: az ilyen névvel kezdődő négy teszt esetében egy vagy több sorban találhatók különböző szemantikai hibák, és a tesztek azt ellenőrzik, hogy az elemző a megfelelő hibákat találja-e meg a megfelelő sorokban.

7.4. Fejlesztési tapasztalatok

A szemantikus elemző teljes kivitelezése volt a legkönnyebb rész az analízis fázis három része közül. Úgy gondolom, hogy ez annak tudható be, hogy már a szintaktikus elemző tervezésekor, a nyelvtan megalkotásánál figyeltem arra, hogy csak olyan szabályok legyenek, amelyeket a szemantikus elemző is fel tud majd dolgozni. Emellett több szemantikai ellenőrzési feladatot már a lexikális elemzővel elvégeztem.

8. KÓDGENERÁLÁSI LEHETŐSÉGEK

8.1. A célnyelv kiválasztása

A kódgenerálás során az első feladat a célnyelv kiválasztása, ugyanis a teljes fázis nagyban függ ettől. Például, ha az assemblyt választjuk célnyelvnek, akkor a következőkre érdemes odafigyelni:

- **memóriakezelés:** assemblyben a teljes memóriakezelés a programozó feladata, fontos pl. a regiszterek helyes és takarékos használata
- **célprogram helyessége:** assemblyben lényegesebben nehezebb programozni, mint egy magas szintű nyelvben, így könnyű hibát véteni, figyelmetlen lenni
- **átlátható, érthető generáló kód**

Egy magas szintű célnyelv esetén bizonyos szempontokból könnyebb, más szempontokból nehezebb lehet a feladat. Vegyük példaként a C#-ot célnyelvként:

- **memóriakezelés:** menedzseltén történik, ez a C# szemétygyűjtőjének (az ún. Garbage Collector) feladata, így erre kevesebb figyelmet kell fordítani
- **komplexitás:** az assembly-hez képest a C# egy bonyolult nyelv. Rengeteg operátorral, nyelvtani szerkezettel rendelkezik, amit a forrásprogram szintaxisától függően nagyon nehéz feladat is lehet.
- **hatékonyság:** fordítás után a C#-ban megírt célprogramot újra le kell fordítani, majd futtatásakor további fordítás szükséges az ún. MSIL-kódra, amit majd a .NET környezet futtatni fog; ez láthatóan lassítja a végrehajtást.

Az én esetemben a nyelv erősen hasonlít a C#-ra, így egy alapszintű leképezés könnyen megvalósítható lenne.

A továbbiakhoz szükséges egy célnyelv választása. Természetesen bármelyik létező programozási nyelvet választhatnám, azonban a C#-ot választom a következők miatt:

- mivel nyelvem a C#-on alapul, valószínűleg nem lesz bonyolult feladat a kódgenerálás (a két nyelv közötti szintaktikus hasonlóságok miatt)
- a C#-ban menedzselt a memóriakezelés, így egyszerűsödik a feladat
- a hatékonyság nem elsődleges; a nyelvemet csak „laboratóriumi környezetben” tervezem használni, nem pedig valódi problémák megoldására
- ezt a nyelvet ismerem a legjobban, így nem szükséges egy másik programozási nyelvet elsajátítanom a kódgenerálás lehetőségeinek elemzéséhez

8.2. A kódgenerálás módja

A kódgenerálás megtervezéséhez először elemezni kell a két nyelv közötti különbségeket, és ehhez lehetőség szerint egy egyértelmű, mindig alkalmazható transzformációt adni.

A célnyelvben az állításokat a pontosvessző karakterrel zárjuk, a forrásnyelvben ez az új sort jelző karakter (vagy karakterpár) használatával érhető el; így az új sort jelző karaktereket pontosvessző karakterekké kell transzformálni.

8.3. Kulcsszó transzformációk

A 8.1. táblázatban forrásnyelv egyszerűbb kulcsszavait elemzem és megadok hozzá egy lehetséges transzformációt.

Kulcsszó (forrásnyelv)	Célnyelvbe transzformált forma
program_kezd	<code>class Program</code> <code>{</code> <code>static void Main()</code> <code>{</code>
program_vége	<code>}</code> <code>}</code>
kilép	<code>System.Environment.Exit(0);</code>
kilépés	
beolvas x	x = <code>System.Console.ReadLine();</code>
beolvas: x	
kiír x	<code>System.Console.Write(x);</code>
kiír: x	
egész	<code>int</code>
tört	<code>float</code> vagy <code>double</code>
logikai	<code>bool</code>
szöveg	<code>string</code>
igaz	<code>true</code>
hamis	<code>false</code>

8.1. táblázat - Az egyszerű kulcsszavak transzformációi

Az összetett kulcsszavak, szerkezetek transzformációját tartalmazza a 8.2. táblázat.

Kulcsszó (forrásnyelv)	Célnyelvbe transzformált forma
ha x akkor y elágazás_vége	<code>if (x) {</code> y <code>}</code>
ha x akkor y különben z elágazás_vége	<code>if (x) {</code> y <code>}</code> <code>else</code> <code>{</code> z <code>}</code>

ciklus_amíg x y ciklus_vége	while (x) { y }
típus[] név = létrehoz[méret]	típus[] név = new típus[méret]

8.2. táblázat - Az összetett kulcsszavak transzformációi

8.4. Operátor transzformációk

A két nyelv operátorai néhány kivétellel megegyeznek, az ezek közötti transzformációkat a 8.3. táblázat tartalmazza.

Kulcsszó (forrásnyelv)	Célnyelvbe transzformált forma
és	&&
vagy	
mod	%
.	+

8.3. táblázat - Operátor transzformációk

8.5. Konverziós függvények transzformációja

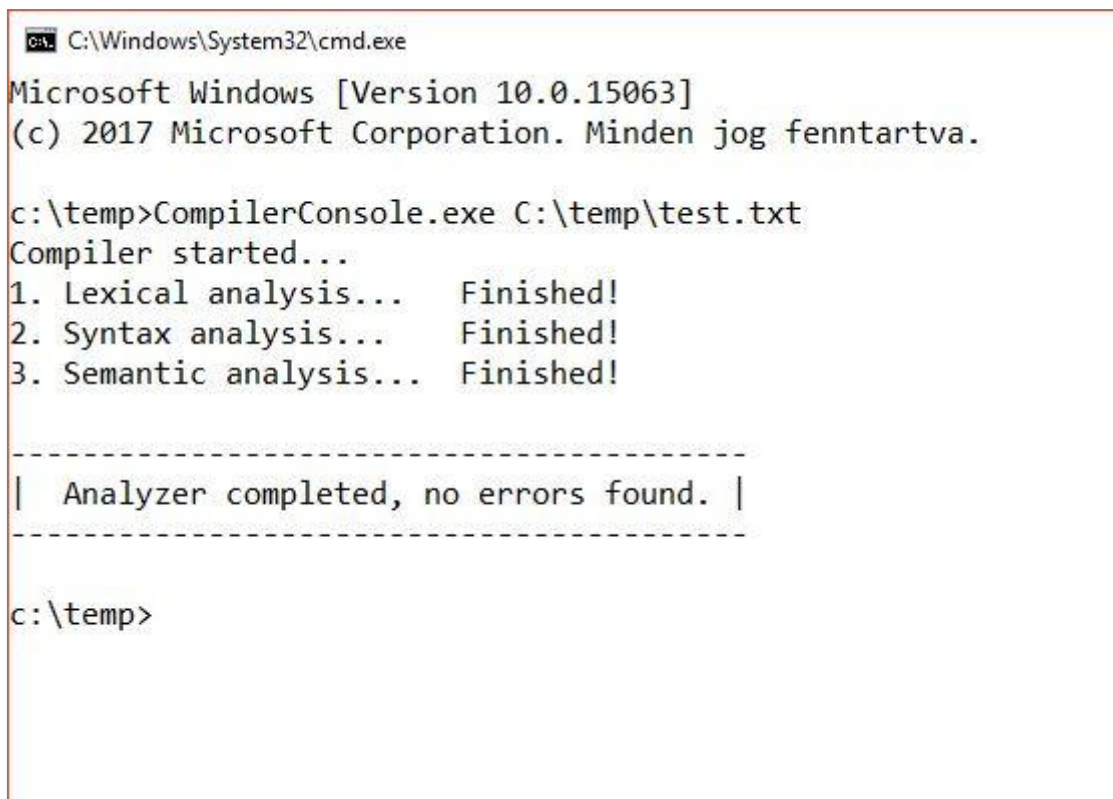
A konverziós függvények transzformációi típuskényszerítéssel (ún. kasztolással), a **System.Convert** osztály metódusainak használatával, és a **ToString()** metódussal megvalósíthatóak, ezeket nem részletezem (azok egyértelműsége és a szakdolgozat terjedelmi határai miatt).

8.6. Megjegyzések

Az előző alfejezetekben részletezett kódgenerálási mód tulajdonképpen egy transzformáció-halmazt adott meg a forrás és a célnyelv között. Ugyan a két nyelv hasonlósága miatt ebben az esetben ez akár a lexikális elemző kimenetéből megtehető, bármilyen más nyelv esetén rendkívül nehéz vagy lehetetlen lenne a szintaxisfa és a szimbólumtábla nélkül.

9. AZ EREDMÉNYEK BEMUTATÁSA, ÉRTÉKELÉSE

Az analízis fázis végeztével elkészítettem egy, a három fázist összefogó DLL-t és konzolalkalmazást. Az előzőben egyetlen publikus metódus szerepel, amely bemenetként a fordítandó programkódot várja szöveg formátumban, kimenetként pedig egy szöveg listát ad, ami tekinthető a fordító által küldött üzeneteknek. A konzolalkalmazás ezt használja fel: argumentumként egy fájl elérési útját várja, és az elemzés közben pedig kiírja a konzolra az érkező üzeneteket. Az alkalmazás működését a 8.3. ábra mutatja be.



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. Minden jog fenntartva.

c:\temp>CompilerConsole.exe C:\temp\test.txt
Compiler started...
1. Lexical analysis... Finished!
2. Syntax analysis... Finished!
3. Semantic analysis... Finished!

-----
| Analyzer completed, no errors found. |
-----

c:\temp>
```

8.3. ábra - A konzolos alkalmazás működése

Habár az elkészített analízáló nem a kezdeti terveknek megfelelő tudással rendelkezik, feladatát – a tesztek által bizonyítottan – jól végzi. Amennyiben pedig a kódgenerálás fázisban célnyelvként a C#-ot választjuk, feltehetően az a fázis is zökkenőmentesen implementálható.

A szakdolgozat során egy olyan szoftvert készítettem el, amelyre a későbbiekben büszke leszek, és rengeteg ismeretet szereztem a több féléven át tartó munka során.

A szakdolgozat teljes fejlődési ciklusa nyomon követhető a következő weboldalon:

<https://github.com/arphox/BScThesis-PseudocodeRecognizer>

10.TARTALMI ÖSSZEFOGLALÓ

Jelen szakdolgozat célja egy nyelv definiálása, a nyelvhez tartozó fordítóprogram analízis fázisának megvalósítása, és a szintézis fázis kódgenerálás lépésének elemzése. A fordítás forrásnyelve egy pszeudokód-szerű nyelv, aminek alapját az Óbudai Egyetem Neumann János Informatikai Karán tanított pszeudokód-nyelv képezi. Az analízis fázis mindhárom részének (lexikális elemzés, szintaktikai elemzés, szemantikai elemzés) implementálása kézzel, C# nyelven történik, és a helyes működést automatizált tesztek biztosítják.

11.ABSTRACT

The goal of this thesis is to define a language, and implement the analysis phase of the corresponding compiler as well as provide an overview of the code generation step of the synthesis phase. The source language of the compiling process is a pseudocode-like language, which is based on the pseudocode-language taught at the John von Neumann Faculty of Informatics of Óbuda University. All three parts of the analysis phase (lexical analysis, syntax analysis, semantic analysis) is written by hand in C# language, and the correct implementation is proven by automatized tests.

12.IRODALOMJEGYZÉK

- [1] Csörnyei, Z.: Fordítóprogramok. *Typotex kiadó*, 2006
- [2] Aiken, A.: Structure of a Compiler,
(<https://www.youtube.com/watch?v=OcDAv-N9Zjs>), utoljára megtekintve: 2016.12.04.
- [3] Aho A. V., Lam M. S., Sethi R., Ullman J. D.: Compilers – Principles, Techniques, Tools (Second Edition). Pearson Addison Wesley, 2007
- [4] Cooper K. D., Torczon L.: Engineering a compiler (second edition). Morgan Kaufmann, 2012
- [5] Wirth, N.: The Programming Language Oberon,
(<http://people.inf.ethz.ch/wirth/Oberon/Oberon07.Report.pdf>), utoljára megtekintve: 2016.12.05.
- [6] Sergyán, Sz.: Algoritmusok, adatszerkezetek I., verzió: 2.0.4.
(<http://users.nik.uni-obuda.hu/sergyan/ProgramozasI/Jegyzet.pdf>), utoljára megtekintve: 2016.12.05.
- [7] IEEE SPECTRUM – The 2016 Top Programming Languages,
(<http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages>),
utoljára megtekintve: 2016.12.07.

13. MELLÉKLETEK

13.1. I. melléklet - A pszeudonyelv definíciója

Pszeudonyelv-definíció

1. BEVEZETÉS

A nyelv a C# nyelvet veszi alapjául. Ami e dokumentumból nem derül ki egyértelműen, arra a C# nyelv szabályai érvényesek. Ez a nyelv funkcionalitására nem vonatkozik, azaz attól, hogy nincs jelezve, hogy nincsenek a C#-ban használatos lambda operátorok, még nem igaz, hogy léteznek.

A nyelv **nem** kisbetű-nagybetű érzékeny (non-case-sensitive).

Minden program

- elejét a program_kezd sor,
- végét a program_vége sor jelzi.

Definíció (whitespace karakter): Tabulátor vagy szóköz karakter.

Egy sorba csak egy utasítás írható. Minden nyelvi elem, azonosító, operátor, stb. előtt és után szerepelnie kell legalább egy whitespace karakternek; ellenkező esetben nem garantált a fordítóprogram hibásan működhet.

A nyelv nem definiál függvényeket, eljárásokat, osztályokat vagy más objektumorientált elvekben használt struktúrákat.

A kilép illetve kilépés utasítás használható a program futásának megszakítására.

2. LEFOGLALT SZAVAK

Lefoglalt szavak listája:

a) Kulcsszavak:

program_kezd
program_vége
kilép
kilépés
ha
akkor
különben
elágazás_vége
ciklus_amíg
ciklus_vége
beolvas
beolvas:
kiír
kiír:
létrehoz
egész
tört
logikai
szöveg

b) Logikai literálok:

igaz
hamis

c) Operátorok

Lásd [Operátorok](#) fejezet.

3. VÁLTOZÓHASZNÁLAT, TÍPUSOK

A változókat használat előtt deklarálni kell és minden változónak meg kell adni a típusát és kezdőértékét. A tömbök létrehozása és kezelése eltérő módon történik, ezt megtalálható a [Tömbkezelés](#) fejezetben.

Létrehozás módja: típus név = kezdőérték

Példa: egész x = 5

Értékadás:

Az értékadás az egyenlőségjel operátorral történik:

x = 3

Tömb típusok esetén létrehozáskor a tömb elemei az adott típus alapértelmezett értékét veszik fel, ez a következő táblázatban található.

Típusok:

Típus neve	Alapértelmezett érték
egész	0
tört	0,0
logikai	hamis
szöveg	"" (üres szöveg)

Literálok:

Legyen $D \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ és jelöljük az üres karaktersorozatot ϵ -nal. Jelentse + az 1 vagy több darab karaktert, * a 0 vagy több darab karaktert, | a „vagy” kapcsolatot.

Típus neve	Literál szabályok	Példa:
egész	$(- \epsilon)D^+$	-64
tört	$(- \epsilon)(D^+, D^+)$	3,14
logikai	hamis igaz	hamis
szöveg	Két idézőjel ("") között tetszőleges, idézőjelet nem tartalmazó karaktersorozat.	”Tetszőleges szöveg...”

Elnevezési konvenció:

A lefoglalt szavak nem használhatók változónévként.

Egy változónév

- **első karaktere** magyar betű,
- **minden további karaktere** magyar betű, arab számjegy vagy aláhúzás karakter (_) lehet

Elfogadott betűk listája (egykarakteres magyar betűk): a, á, b, c, d, e, é, f, g, h, i, í, j, k, l, m, n, o, ó, ö, ő, p, q, r, s, t, u, ú, ü, ű, v, w, x, y, z, A, Á, B, C, D, E, É, F, G, H, I, Í, J, K, L, M, N, O, Ó, Ö, Ő, P, Q, R, S, T, U, Ú, Ü, Ű, V, W, X, Y, Z.

Elfogadott számjegyek listája: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Escape-szekvenciák:

Minden szöveg típusú értékbe beilleszthetők ún. escape-szekvenciák, amelyeknek speciális jelentése van az adott szövegen belül. Ezek az alábbi táblázatban találhatók:

Jel	Név
\n	Új sor
\t	Vízszintes tabulátor
\”	Idézőjel
\\	Fordított perjel

4. TÍPUSKONVERZIÓK

A nyelv erősen típusos, implicit típuskonverzió nem létezik. Explicit típuskonverzió a beépített konverziós függvények használatával végezhető el.

Példa konverziós függvény használatára:

```
egész y = törtből_egészbe(5,4)           // y értéke 5 lesz
```

Minden két különböző alaptípus között (egész, tört, szöveg, logikai) értelmezett konverziós függvény, ezek elnevezési sémája a következő:

forrástípusból_céltípusba

Azaz ha a forrástípus egész, a céltípus logikai, akkor a függvény neve:

egészből_logikaiba

Így a következő függvények alakulnak ki:

- egészből_logikaiba
- egészből_törtbe
- egészből_szövegbe
- törtből_egészbe
- törtből_logikaiba
- törtből_szövegbe
- logikaiból_egészbe
- logikaiból_törtbe
- logikaiból_szövegbe
- szövegből_egészbe
- szövegből_törtbe
- szövegből_logikaiba

5. OPERÁTOROK

A használható típusok oszlopban ha egy adott operátornál egy típust jelöltem meg, akkor adott típusú literálon és változón (illetve többoperandusú operátorok között azok között) használható, vagy azok egy kifejezésén.

Jel	Név	Példa	Használható típusok
[]	tömbindexelő	t[x] (t tömb x. elemére való hivatkozás)	Tömbindex csak egész lehet.
-	numerikus negáció	-x	
!	logikai negáció	!d	logikai
()	kerek zárójelpár		bármilyen

Jel	Név	Példa	Használható típusok
=	értékadás	x = 1	bármilyen két azonos típus között
==	egyenlőség-vizsgálat	y == 4	
!=	nem-egyenlőség vizsgálat	s != 8	
és	feltételes és művelet	z és y	logikai
vagy	feltételes vagy művelet	k vagy s	logikai
> >= < <=	relációs operátorok	x > 3 y >= 3 x < 4 y <= 4	szám típusok: egész, tört
+	összeadás	g + h	
-	kivonás	i - j	
*	szorzás	k * l	
/	osztás	m / n	
mod	maradékos osztás	o mod p	
.	szöveg összefűzés	"alma"."körte"	szöveg

6. KIFEJEZÉSEK

A nyelv korlátozott kifejezésfelismerési képességeket definiál. Ez a nyelv kifejezőerejét nem csökkenti, de a használat módját kényelmetlenebbé teszi.

Példák:

Ha a következő kifejezést szeretnénk leírni a nyelvben:

$$\text{egész } a = 2 * (3 + 4) - 2$$

Akkor azt a fenti forma helyett így kell leírnunk:

$$\text{egész } a = 3 + 4$$

$$a = 2 * a$$

$$a = a - 2$$

Ezek pontosabb szabályai kiolvashatóak a szintaktikus nyelvtan definícióból.

7. KOMMENTEZÉS

A nyelv kommentezési lehetőségei megegyeznek a C# nyelvben lévőkkel:

Jel	Név	Példa
//	egysoros komment	egész x = 4 //x értéke 4
/* */	többsoros komment	/* A sorozatszámítás egyszerű progra- mozási tétel. */

// után az adott sorban minden karaktert kommentként értelmez.

/* és */ karakterpárok között minden karaktert kommentként értelmez.

8. VEZÉRLÉSI SZERKEZETEK

Szelekció:

ha feltétel **akkor**

utasítás(ok)

elágazás_vége

ha feltétel **akkor**

utasítás(ok)

különben

utasítás(ok)

elágazás_vége

Iteráció:

ciklus_amíg bennmaradási_feltétel

utasítás(ok)

ciklus_vége

Vezérlés folyamata:

1. Bennmaradási feltétel megvizsgálása. Ha hamis, ugorj a ciklus_vége utáni sorra.
 2. Ciklusmag utasításainak végrehajtása. Ugorj az 1. pontra.
-

9. I/O KEZELÉS

Az input és output kezelésére a következő utasítások használhatók:

Beolvasás:

Utasítás: beolvas

Példa: beolvas x

Leírás: beolvassa az aktuális input streamről a következő sort, majd a beolvasott szöveget megpróbálja átalakítani szöveg típusra. Ha sikertelen, akkor futási idejű hiba keletkezik.

Megjegyzés: x-et a használat (beolvasás) előtt deklarálni kell!

Kiírás:

Utasítás: kiír

Példa: kiír x

Leírás: kiírja az aktuális output streamre az x változó tartalmát.

„Összetett” kiírás:

Egy változó értékét hozzáfűzhetjük a kiírandó szöveghez.

Példa: kiír "x értéke: " . x

Magyarázat: Összefűzi az "x értéke: " szöveget az x változó értékének szövegbeli reprezentációjával, majd kiírja a konzolra.

Működési példa: ha x értéke 12, akkor ez kerül az aktuális output streamre:

x értéke: 12

10.TÖMBKEZELÉS

A tömbindexelés nullától történik. Tömbindexként csak egész literálok illetve egész típusú változók használhatók.

A nyelvben csak az egydimenziós tömbök támogatottak.

Tömblétrehozás:

egész[] x = létrehoz[N]

ahol:

N a tömb mérete (pozitív egész szám)

Egy tömb deklarációja és definíciója egy sorba is vonható:

Példa: egész[] tömb = létrehoz[6]

A tömb elemeire való hivatkozás:

x[0] = 5 Beállítja az x tömb 0. elemének értékét az 5 értékre.

11. NYELVTAN DEFINÍCIÓ

Itt található a nyelvtan szintaktikájának definíciója Backus-Naur formában.

Kezdő mondat szimbólum: <Program>

<Program> ::= "program_kezd" "újsor" <Állítások> "program_vége"

<Állítások> ::= <Állítás> "újsor" <Állítások>
| <Állítás> "újsor"

<Állítás> ::= <VáltozóDeklaráció>
| <Értékadás>
| <IoParancs>
| "kilép"
| "ha" <NemTömbLétrehozóKifejezés> "akkor" "újsor" <Állítások>
"különben" "újsor" <Állítások> "elágazás_vége"
| "ha" <NemTömbLétrehozóKifejezés> "akkor" "újsor" <Állítások>
"elágazás_vége"
| "ciklus_amíg" <NemTömbLétrehozóKifejezés> "újsor" <Állítások>
"ciklus_vége"

<VáltozóDeklaráció> ::= <AlapTípus> "azonosító" "="
<NemTömbLétrehozóKifejezés>
| <TömbTípus> "azonosító" "=" "azonosító"
| <TömbTípus> "azonosító" "=" <TömbLétrehozóKifejezés>
| <AlapTípus> "azonosító" "=" <BelsőFüggvény> "("
<NemTömbLétrehozóKifejezés> ")"

<Értékadás> ::= "azonosító" "=" <NemTömbLétrehozóKifejezés>
| "azonosító" "=" <TömbLétrehozóKifejezés>
| "azonosító" "=" <BelsőFüggvény> "("
<NemTömbLétrehozóKifejezés> ")"
| "azonosító" "[" <NemTömbLétrehozóKifejezés> "]" "="
<NemTömbLétrehozóKifejezés>

<Operandus> ::= <UnárisOperátor> "azonosító"
| <UnárisOperátor> "literál"
| "azonosító" "[" <Operandus> "]"
| "azonosító"
| "literál"

<NemTömbLétrehozóKifejezés> ::= <BinárisKifejezés>
| <Operandus>

<TömbLétrehozóKifejezés> ::= "létrehoz" "["
<NemTömbLétrehozóKifejezés> "]"

<BinárisKifejezés> ::= <Operandus> <BinárisOperátor> <Operandus>

<BinárisOperátor> ::= "=="
| "!="
| "és"
| "vagy"
| ">"
| ">="
| "<"
| "<="
| "+"
| "-"
| "*"
| "/"
| "mod"
| "."

<AlapTípus> ::= "egész"
| "tört"
| "szöveg"
| "logikai"

<TömbTípus> ::= "egész tömb"
| "tört tömb"
| "szöveg tömb"
| "logikai tömb"

<IoParancs> ::= "beolvas" "azonosító"
| "kiír" "azonosító"

<BelsőFüggvény> ::= "egészből_logikaiba"
| "egészből_törtbe"
| "egészből_szövegbe"
| "törtből_egészbe"

```
| "törtből_logikaiba"  
| "törtből_szövegbe"  
| "logikaiból_egészbe"  
| "logikaiból_törtbe"  
| "logikaiból_szövegbe"  
| "szövegből_egészbe"  
| "szövegből_törtbe"  
| "szövegből_logikaiba"
```

```
<UnárisOperátor> ::=  "-"  
|  "!"
```