|  |  |  |
| --- | --- | --- |
| oe_cimer_szines_print_res | **NEUMANN JÁNOS**  **INFORMATIKAI KAR** | NIK_cimer.jpg |

|  |  |  |
| --- | --- | --- |
| **OE-NIK**  **2017** | Hallgató neve:  Hallgató törzskönyvi száma: | **Ozsvárt Károly**  **T/003699/FI12904/N** |

**SZAKDOLGOZAT**

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:

**Pszeudokód 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 7](#_Toc476558626)

[Rövid tartalmi összefoglaló a téma területéről, a feladatról 8](#_Toc476558627)

[A megoldandó probléma és motivációi 9](#_Toc476558628)

[1. A fordítóprogramok vázlatos működése 10](#_Toc476558629)

[1.1. Áttekintés 10](#_Toc476558630)

[1.2. Lexikális elemző 11](#_Toc476558631)

[1.2.1. Előreolvasás 12](#_Toc476558632)

[1.2.2. Formális felírás és a kimenet 13](#_Toc476558633)

[1.2.3. Szimbólumtábla 13](#_Toc476558634)

[1.1. Szintaktikus elemző 15](#_Toc476558635)

[1.2. Szemantikus elemző 15](#_Toc476558636)

[1.3. Kódgeneráló 15](#_Toc476558637)

[1.4. Kódoptimalizáló 15](#_Toc476558638)

[1.5. A fordítási fázisok komplexitásának összehasonlítása [2] 16](#_Toc476558639)

[2. A feladat elemzése 17](#_Toc476558640)

[2.1. Lexikális elemző 17](#_Toc476558641)

[2.2. Szintaktikus elemző 17](#_Toc476558642)

[2.3. Szemantikus elemző 17](#_Toc476558643)

[2.4. Kódgeneráló 17](#_Toc476558644)

[2.5. Kódoptimalizáló 17](#_Toc476558645)

[3. Fejlesztőeszközök kiválasztása 18](#_Toc476558646)

[3.1. Programozási nyelv választás 18](#_Toc476558647)

[3.2. Operációs rendszer és fejlesztői környezet választás 18](#_Toc476558648)

[4. A felhasznált pszeudonyelv definiálása 20](#_Toc476558649)

[4.1. Áttekintés 20](#_Toc476558650)

[4.2. Bevezetés 21](#_Toc476558651)

[4.3. Lefoglalt szavak 21](#_Toc476558652)

[4.4. Változóhasználat, típusok 22](#_Toc476558653)

[4.5. Típuskonverziók 23](#_Toc476558654)

[4.6. Operátorok 23](#_Toc476558655)

[4.7. Kommentezés 23](#_Toc476558656)

[4.8. Vezérlési szerkezetek 24](#_Toc476558657)

[4.9. I/O kezelés 24](#_Toc476558658)

[4.10. Tömbkezelés 24](#_Toc476558659)

[5. Lexikális elemző 25](#_Toc476558660)

[5.1. Tervezés és implementálás 25](#_Toc476558661)

[5.1.1. Lexikális elem kódok 25](#_Toc476558662)

[5.1.2. Kimeneti szimbólumsorozat 25](#_Toc476558663)

[5.1.3. Szimbólumtábla 26](#_Toc476558664)

[5.1.4. Elemző 27](#_Toc476558665)

[5.2. Tesztelés 29](#_Toc476558666)

[5.3. Fejlesztési tapasztalatok 30](#_Toc476558667)

[6. Szintaktikus elemző 32](#_Toc476558668)

[6.1. Tervezés 32](#_Toc476558669)

[6.2. Implementálás 32](#_Toc476558670)

[6.3. Tesztelés 32](#_Toc476558671)

[6.4. Fejlesztési tapasztalatok 32](#_Toc476558672)

[7. Szemantikus elemző 33](#_Toc476558673)

[7.1. Tervezés 33](#_Toc476558674)

[7.2. Implementálás 33](#_Toc476558675)

[7.3. Tesztelés 33](#_Toc476558676)

[7.4. Fejlesztési tapasztalatok 33](#_Toc476558677)

[8. Irodalomjegyzék 34](#_Toc476558678)

[9. Mellékletek 35](#_Toc476558679)

[9.1. I. melléklet - A pszeudonyelv definíciója 35](#_Toc476558680)

# Köszönetnyilvánítás

Szeretném megköszönni Dr. Sergyán Szabolcs, az Óbudai Egyetem Neumann János Informatikai Karának, azon belül az Alkalmazott Informatikai Intézet igazgatójának segítségét, aki a nyelvdefiníciómra adott véleményeként hasznos tanácsokkal látott el; néhány helyen rávilágított pontatlanságokra és hiányosságokra. Valamint köszönöm ugyanezen karról 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) 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 tanáraimat, 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, megértetéséhez az oktatók gyakran használnak pszeudokódot, amely segítségével szövegesen definiálható egy program 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 pszeudonyelvet majd megad hozzá egy, a nyelvben írt programok ellenőrzésére használható, a fordítás analízis fázisát elvégző programot. A definiált pszeudonyelv számos elemében szándékosan hasonlít az Óbudai Egyetem Neumann János Informatikai Karán, Mérnökinformatikus BSc. szakon tanított Programozás I. tárgy keretein belül használt pszeudonyelvre, 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.

# A fordítóprogramok vázlatos működése

## 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 fordítson.

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.

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

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

* 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 két folyamat 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 előbbiek átlátását segíti az 1.2. ábra.



1.2. ábra - A fordítóprogramok szerkezete

## 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 karaktersorozathoz fűzött éppen aktuális karakter értelmes lexikális elemet ad, akkor azt tároljuk el és lépjünk 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 lépjünk 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 résszöveg 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.

## Előreolvasás

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: „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.

Röviden összefoglalva: a lexikális elemzőt úgy kell megalkotni, hogy kezelje azon eseteket, amikor két lexikális elem úgy fedi egymást, hogy az egyikből másikba való átmenet hibásan felismert lexikális elemen keresztül vezet.

## 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:

1. Paraméter nélküli lexikális elem
2. 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 mindegyik 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.

## 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 tárolhatóak 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

(a felsorolás forrása: [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 detektálható 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.

## 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]

## Szemantikus elemző

A szemantikus elemző a szintaxisfa és a szimbólumtábla felhasználásával ellenőrzi a programkód szemantikai helyességét. Fontos részei:

* típus ellenőrzés:
  + egy kifejezés két oldalán összehasonlítható típusok szerepelnek-e
  + az operátorok és operandusok kompatibilitásának ellenőrzése
  + 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

([1] alapján)

## Kódgeneráló

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

## 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]

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.

## A fordítási fázisok komplexitásának összehasonlítása [2]

A korai fordítóprogramok (pl. FORTRAN) esetén a fázisok komplexitását szemlélteti az 1.3. ábra.



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

A modern fordítóprogramok esetén a fázisok komplexitását szemlélteti az 1.4. ábra.



1.4. 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 új terület volt és nehezebb volt elkészíteni). A kódoptimalizáló rész növekedését pedig a programok bonyolultságának növekedése vonta maga után (nagyobb kódbázison fontos minél többet optimalizálni).

# A feladat elemzése

Ebben a fejezetben a megoldandó feladatokat elemzem.

## Lexikális elemző

Habár a lexikális elemzéshez rendelkezésre állnak automatizált eszközök (pl. Lex, Yacc), úgy döntöttem, hogy ezt a részt én implementálom. Döntésemet azzal indoklom, hogy szeretném alaposan megismerni a lexikális elemzés folyamatát, és ezt úgy tudom megtenni, ha saját magam írom meg az elemző programkódját. A lexikális elemző implementálását szakdolgozatom írásának első félévében végzem el.

## Szintaktikus elemző

A szintaktikus elemző implementálása lesz a legnehezebb feladat, ugyanis e fázis megértéséhez és implementálásához megfelelő szintű ismeretre lesz szükségem *formális nyelvek és automaták* témakörben. A szintaktikus és szemantikus elemző implementálása a szakdolgozat írásának második félévében történik meg.

## Szemantikus elemző

A szemantikus elemző jóval kisebb komplexitásúnak ígérkezik, mint a szintaktikus elemző, mivel tervezem már a lexikális elemzőben ellenőrizni, hogy egy változót deklarálni kell használat előtt és egy változót pontosan egyszer lehet deklarálni.

## Kódgeneráló

Szakdolgozatomban vizsgálom a kódgenerálás lehetőségeit, azonban erre csak akkor lesz lehetőségem, ha a program analizátor (analízis fázist megvalósító) része már implementálásra került, tehát ennek részletezésére a következő félévben kerül sor.

## Kódoptimalizáló

A kódoptimalizálás fázis nem része a szakdolgozatnak, és nem is érzem szükségesnek, ugyanis, ha a célból készül a fordítóprogram, hogy a kezdő programozók képesek legyenek összehasonlítani bizonyos algoritmusok futási idejét, akkor a kódoptimalizáló ennek eredményét módosíthatja. Például, ha a felhasználó azt szeretné kipróbálni, hogy két hasonló algoritmus közül melyik a gyorsabb, akkor azonos teszteredményt kaphat, ha a kódoptimalizáló optimalizálja mindkét algoritmust, miközben optimalizálás nélkül különböző lenne a kettő futási ideje.

# Fejlesztőeszközök kiválasztása

## Programozási nyelv választás

Az implementáció elkészítéséhez számos nyelv rendelkezésemre állt. Egyetemi tanulmányaim során (a teljesség igénye nélkül) a következő nyelveket ismertették meg velünk: C#, Java, C++, SQL, PHP, MATLAB. Ezek közül fordítóprogramok írására kevésbé használhatóak 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 minőségi integrált fejlesztői környezet nem 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, codeproject, stb.) legtöbbször találok hasznos információt vagy kódrészletet a témával kapcsolatban. A fentiek alapján ez a nyelv kitűnően használható a szakdolgozatomhoz.

## Operációs rendszer és fejlesztői környezet választás

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

* SharpDevelop (Windowsra)
* QuickSharp (Windowsra)
* C# Studio (Windowsra)
* Microsoft Visual Studio (Windowsra)
* MonoDevelop (Linuxra)

A Microsoft Visual Studio, számos hasznos eszközt biztosít arra, hogy a kódolás, hibakeresés és optimalizálás könnyebb legyen (lépésenkénti 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 csak a Windows operációs rendszer családot támogatja, azt kell használnom. Ez nem okoz problémát, ugyanis eddigi pályafutásom során is mindig Windows rendszereken dolgoztam. Végül a Windows 10 Pro operációs rendszer mellett döntöttem.

# A felhasznált pszeudonyelv definiálása

Ebben a fejezetben részletezem a pszeudonyelv definiálásának folyamatát. A dokumentum jelenlegi legfrissebb verziója a dokumentum [I. mellékleteként](#_I._melléklet_-) található meg.

## 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 nyelvdefiníciómat azonban szándékosan úgy fogom elkészíteni, hogy azt egy kezdő programozó is meg tudja érteni. Emiatt a fent említett módokat kevésbé fogom használni; amikor pedig használom, 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 folyamodok.

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, előredefiniált (beépített) eljárások, modulok)
9. Függelék (ahol megtekinthető a nyelv szintaxisa EBNF formában)

A fentiek áttanulmányozása és átgondolása alapján saját nyelvdefiníciómat 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.

## Bevezetés

Itt általános információkat adok a nyelvről, pl. hogy nem kisbetű 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.

## 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 szavak. Erre azért van szükség, mert a nyelv nem definiál függvényeket, és emiatt fontosnak tartom, hogy a programozónak explicit módon[[1]](#footnote-1) 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 jelentősen 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 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 „átirányításokat” 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 (kivéve „különben ha” szerkezet)
* különben
* elágazás\_vége (azonos az elágazás vége utasítással)
* ciklus\_amíg (azonos a ciklus amíg utasítással)
* ciklus\_vége (azonos a ciklus vége utasítással).

A számlálós ciklus is azonos módon működik, azonban mivel a nyelv típusos, a ciklusváltozó típusát is meg kell adni annak neve előtt. Ez a következő utasításokat fedi le: ciklus, -tól, -től, -ig. A –tól és –től utasítások azonos funkciójúak; nyelvtani elemzést nem végez a nyelv, azaz szintaktikailag teljesen elfogadható például a ciklus egész i = 4-tól kezdetű programkód is.

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

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

## 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 implementálhatóak 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 kezelése valamint a fájlírás során (erre a nyelvben nincs lehetőség) van nagyobb jelentősége. A nyelvdefiníció korábbi változataiban szerepelt 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.

## Típuskonverziók

Tekintettel arra, hogy a nyelv erősen típusos, szükség van konverziós függvényekre. Ezek beépített függvény formájában léteznek, és a típusok közötti átjáráshoz használhatóak. A programozók munkájának megkönnyítése érdekében létezik automatikus implicit típuskonverzió. Ez abban nyilvánul meg, hogy az *egyértelmű* konverziókat a nyelv értékadás esetén automatikusan elvégzi. Véleményem szerint az egyes konverziós függvények nem igényelnek indoklást, leírásuk megtekinthető a nyelvdefinícióban.

## Operátorok

A nyelv operátorainak megtervezése során több nyelv is megihletett. A legtöbb operátort a C# nyelvből vettem át. Kivételt képeznek ezelől 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

## Kommentezés

Ahogyan azt a nyelvdefinícióban is leírom, *„A nyelv kommentezési lehetőségei megegyeznek a C# nyelvben lévőkkel (…)”*. A fejezet nem igényel további részletezést.

## 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 tesztelős ciklust a programozás során ritkán használjuk (pl. a *Sergyán Szabolcs – Algoritmusok, adatszerkezetek I.* című irodalom is [6]), illetve én sem kedvelem, így kihagytam a nyelvből.

Az iterációs lehetőségek (elöl tesztelős és számlálós ciklus) esetén feltüntettem a vezérlés folyamatát is, hogy működésük könnyebben megérthető legyen.

## 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 kideríthetőek.

## Tömbkezelés

A nyelvben definiálhatóak 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 közzétett *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).

# Lexikális elemző

Ebben a fejezetben részletezem a lexikális elemző elkészítésének folyamatát.

## Tervezés és implementálás

## Lexikális elem kódok

A program működéséhez szükséges egy egységes adatszerkezet, amelyből futásidőben 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. Ez a gyorsaság jól megfelel a céljaimnak. A kódokat tároló hash táblát mindenhonnan el kell tudni érni, és csak egy példány szükséges belőle, ennek megfelelő helyen és módon kell majd tárolni.

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* (a feldolgozást könnyíti meg). A nyelvdefinícióból következőleg bizonyos kulcsszavak azonos jelentést hordoznak, ezeket úgy kezeltem le, hogy minden jelentés azonos számkódra mutat (pl. –tól, -ig kulcsszavak).

A kódokat egy erre szolgáló osztály (LexicalElementCodes) tartalmazza, a konkrét adatszerkezet pedig:

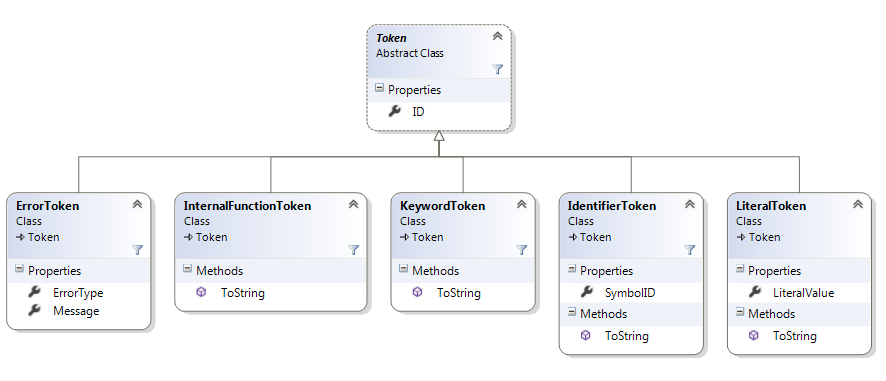
private static Dictionary<string, int> LexicalElementToCode

Az adatszerkezet privát, mert beszúrási műveletre csak az inicializálás során van szükség. Osztályon kívülről csak lekérdezni lehet, ezekre elkészítettem a megfelelő publikus metódusokat.

## 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 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 ősosztályt és hozzá tartozóan megfelelő leszármazott osztályokat, ezek (vázlatos) osztálydiagramját az 5.1. ábra mutatja.



5.1. ábra - Token osztály és leszármazottjai

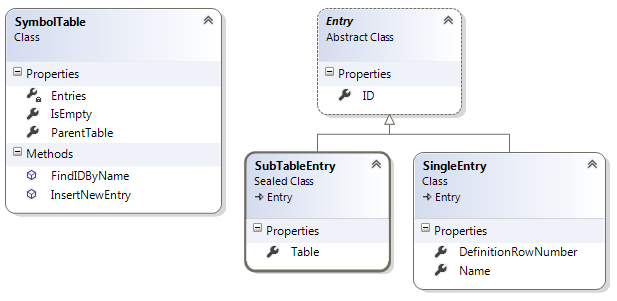
Az ErrorToken azt jelzi, ha egy részszövegből nem sikerült felismerni egy elemet sem.

Így az elemző kimenete egy Token típusú láncolt lista adatszerkezet lesz.

## Szimbólumtábla

A szimbólumtábla tárolásra számos lehetőség közül választhattam (e lehetőségek megtekinthetőek 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 Entry absztrakt osztályt, amely jelképez egy bejegyzést. Ennek két leszármazottja van, SubTableEntry és SingleEntry, A SubTableEntry egy altáblát jelképez és egy SymbolTable referenciát tárol, a SingleEntry pedig egy változót jelképez. A SymbolTable osztály pedig tartalmaz egy Entry típusú láncolt listát. Így egy „gyökér” SymbolTable objektumban eltárolható az egész szimbólumtábla. A szerkezet megértését segíti a fentiek vázlatos osztálydiagramja (5.2. ábra).



5.2. ábra - A szimbólumtábla vázlatos osztályszerkezete

A szimbólumtáblát elegendő egyszer, egy mindenhonnan elérhető változóban tárolni. Az ábrán látható egy ParentTable nevű tulajdonság. Ez egy olyan SymbolTable referencia, amely az adott tábla közvetlen ősre mutat; azért van rá szükség, mert így könnyebben bejárható a fa, akár rekurzió használata nélkül is. Az egyetlen gyökér szimbólumtáblában ez az érték null, ezzel jelezve azt, hogy ez a fa gyökere. A SingleEntry-ben található DefinitionRowNumber tulajdonság megadja, hogy a változót a forrásfájl hányadik sorában definiálták.

## Elemző

Az eddigi alfejezetek az elemző adatstruktúráját taglalták, ez pedig a működést mutatja be. A működés részt öt osztály írja le. A könnyebb megértés érdekében ezek vázlatos felépítése a következő:

* LexicalAnalyzer: Az elemző fő része; ez végzi az elemzést, ad hozzá elemeket a szimbólumtáblához és a kimeneti tokenlistához. Ezen osztály bizonyos metódusai alkotják a lexikális elemző publikus interfészét.
* LexicalAnalyzerState: Az elemző által megvalósított állapotgép állapotait tároló felsorolás típus.
* LexicalElementIdentifier: Megadja, hogy egy karakterlánc milyen lexikális elemnek felel meg.
* OutputTokenListHandler: A kimeneti listát kezelő osztály, példányként létezik a LexicalAnalyzer osztályban.
* SymbolTableHandler: A szimbólumtáblát kezelő osztály, példányként létezik a LexicalAnalyzer osztályban.

A két utóbbi osztály azért lett kibontva a fő osztályból, mert így jelentősen egyszerűsödik a fő osztály és jobban átlátható.

Az elemzés folyamata a következő:

1. Kívülről meghívják egy LexicalAnalyzer példány egyik publikus metódusát (PerformLexicalAnalysisOnFile(string path) vagy PerformLexicalAnalysisOnString(string sourceCode)), amely majd az elemzés végeztével visszatér a token listával (List<Token> objektummal).
2. Némi bemeneti ellenőrzés után megtörténik a lényegi elemzés (DoLexicalAnalysis()), majd a szimbólumtábla tisztítása (üres levél szimbólumtáblák törlése) és frissítése zajlik le a visszatérés előtt.
3. DoLexicalAnalysis(): kiválasztja a kezdeti állapotot, majd amíg az állapot nem a végállapot (Final), végrehajtja az állapotnak megfelelő metódust, majd új állapotot keres.

Az elemző a következő állapotokat definiálja:

* **Initial**: kezdeti állapot
* **Comment1Row**: egysoros kommentet lekezelő állapot
* **CommentNRow**: 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

1. Az állapottól függően a következő metódusok futhatnak le, leírásuk:
   1. Whitespace(): továbbhalad a bemeneti szövegen, és elmegy a következő nem üres karakterhez.
   2. StringLiteral(): idézőjel (”) észlelése esetén lép érvénybe. Elmegy az adott szöveg végéig, majd hozzáfűzi a kimeneti token listához a literálhoz tartozó tokent.
   3. Comment1Row(): két darab perjel (//) észlelése esetén lép érvénybe. Elmegy az adott sor végéig.
   4. Comment1Row(): egy perjel és egy csillag (/\*) észlelése esetén lép érvénybe. Elmegy az adott többsoros komment végéig (amíg nem észlel \*/ sorozatot).
   5. NonWhitespace(): a legbonyolultabb metódus, ez kezeli az összes többi lexikális elemet; szöveg (ami nem szöveg literál) érzékelése 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 részszövegeket). Kezeli az ütközéseket a következő lexikális elemek között:
      * egész literál és tört literál kulcsszavak
      * - (mínusz) operátor és -tól, -től, -ig kulcsszavak.

Eközben építi a szimbólumtáblát és bővíti a kimeneti token listát. A metódus részletesebb megértése a kód és működés mélyebb ismeretét igényli, így ezt jobban itt nem részletezem.

## Tesztelés

A lexikális elemző teszteléséhez egy egyszerű (Tester nevű) konzolos projektet készítettem az eddigi LexicalAnalysis nevű projekt mellé. A tesztprojekt lefuttatja a megadott fájlon a lexikális elemzést, majd a token listát és a szimbólumtábla tartalmát kiírja egy fájlba.

A teszteléshez számos tesztesetet készítettem. A konkrét teszteseteket és az azokra kigenerált kimeneti fájlokat nem közlöm, ugyanis akkor a szakdolgozat terjedelme számos (20-30) oldallal megnőne, és ez nem kívánatos. A tesztesetek elnevezését, rövid leírását, eredményét az 5.1. táblázat közli. A tesztelési fázis közben, ha egy teszteset hibásan futott le, akkor a hibát okozó kódrészletet javítottam; a táblázat utolsó oszlopa azt jelzi, hogy az adott teszteset derített-e fel hibát a kódban.

Néhány tesztesetnél az elemző sebességét teszteltem. A sebességmérés egy viszonylag régi számítógépen történt, melynek specifikációja megtekinthető itt[[2]](#footnote-2). A mérés idejét nem tartalmazza a bemeneti fájl háttértárról való beolvasását és a kimeneti fájl oda való kiírását (ugyanis az a háttértártól is függővé teszi a sebességet), csak a lexikális elemzéssel eltöltött tényleges időt.

|  |  |  |
| --- | --- | --- |
| **Tesztfájl neve** | **Mit tesztel?** | **Derített fel hibát? Ha igen, milyet?** |
| **Helyes bemeneti fájlok** | | |
| 01\_empty.opl | üres bemeneti fájlt | igen, szépséghibát |
| 02\_nostartend.opl | program\_kezd és program\_vége utasításokat nem tartalmazó fájlt | nem |
| 03\_noend.opl | program\_vége utasítást nem tartalmazó fájlt | nem |
| 04\_nostart.opl | program\_kezd utasítást nem tartalmazó fájlt | nem |
| 05\_notonlycode.opl | a program\_kezd és program\_vége sorok előtt és után egyéb karaktereket is tartalmazó fájlt | nem |
| 06\_comments.opl | főleg kommenteket | nem |
| 07\_declarations.opl | deklarációkat | igen, szépséghibát |
| 08\_masodfoku.opl | deklarációkat, beolvasást, szelekciót | nem |
| 09\_for.opl | számlálós ciklust | nem |
| 10\_arrayForIf.opl | deklarációt, szelekciót, számlálós ciklust, tömbkezelést | nem |
| 11\_expressions.opl | összetett aritmetikai kifejezéseket | nem |
| 12\_simpleTheorems.opl | egymásba épített blokkokat, szelekciót, tömbkezelést | nem |
| 13\_deepBlocks.opl | mélyen egymásba épített blokkokat, azokon belül deklarációkat (szimbólumtáblát) | nem |
| 14\_internalFunctions.opl | beépített függvényeket | nem |
| 15\_loadtest1k.opl | hosszú fájlokat (1000 sor) | nem idő: 130 ms |
| 16\_loadtest10k.opl | hosszú fájlokat (10000 sor) | nem idő: 622 ms |
| **Hibás bemeneti fájlok** | | |
| 101\_redeclaration.opl | változó újradeklarálását | nem |
| 102\_multipleStart.opl | többszörös program\_kezd sorokat egymás után | igen, kisebb hibát |
| 103\_notype.opl | nem deklarált változóhasználatot | igen, közepes hibát |
| 104\_unknownSymbol.opl | a nyelvben nem létező lexikális elemet (;) | nem |

5.1. táblázat - A lexikális elemző tesztelése

## Fejlesztési tapasztalatok

Az alfázis implementálása az első félév első felében történt. Már az implementálás során is sikerült bizonyos gyorsításokat eszközölni a programban:

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

A programkód szerkezete akkori állapotában azonban nem volt jól áttekinthető.

A félév közepe-vége felé elkezdtem olvasni egy könyvet a tiszta kódolásról [[3]](#footnote-3). Tanulva a könyvből, elhatároztam, hogy refaktorálom a kódomat, és (viszonylag) tiszta, jól átlátható kóddá alakítom. Így az első félév végén a tesztelési fázis előtt becslésem szerint 20 órát töltöttem a kód refaktorálásával. Az eredmény sokkal jobban átlátható és érthető kód: beszédesebb metódusnevek, rövidebb metódusok, eltávolított nem használt kódrészletek és kommentek, több osztály, rövidebb fájlok.

Elvégeztem egy tesztet is a LexicalAnalysis projekten a nagy refaktorálás előtti és utáni verzióval, amelyben a kód metrikáit elemeztem. A tesztet a Microsoft Visual Studio 2015 Enterprise Edition fejlesztői környezettel végeztettem el, Debug módban. A teszt eredményét tartalmazza az 5.2. táblázat; a színek a változás irányát és mértékét jelzik.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Mikor? | Maintainability Index | Cyclomatic Complexity | Depth of  Inheritance | Class Coupling | Lines of Code |
| előtte | 83 | 344 | 2 | 34 | 515 |
| utána | 85 | 239 | 2 | 39 | 470 |

5.2. táblázat - A lexikális elemző refaktorálásának eredménye kód metrikák szempontjából

# Szintaktikus elemző

## Tervezés

Megírandó.

## Implementálás

Megírandó.

## Tesztelés

Megírandó.

## Fejlesztési tapasztalatok

Megírandó.

# Szemantikus elemző

## Tervezés

Megírandó.

## Implementálás

Megírandó.

## Tesztelés

Megírandó.

## Fejlesztési tapasztalatok

Megírandó.

# 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/Programozas1Jegyzet.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.

# Mellékletek

## I. melléklet - A pszeudonyelv definíciója

1. Explicit módon: egyértelműen, külön kihangsúlyozva a jelentést [↑](#footnote-ref-1)
2. A teszteléshez használt számítógép:  
   Alaplap: GIGABYTE EP45-UD3L.  
   Memória: BUFFALO 2x2GB DDR2 800MHz.  
   Processzor: Intel Dual Core E5200. [↑](#footnote-ref-2)
3. Robert C. Martin – Clean Code – A Handbook of Agile Software Craftmanship [↑](#footnote-ref-3)