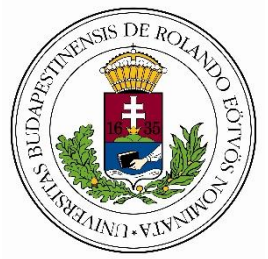


Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

Fordítóprogram optimalizációk implementációja

Pataki Norbert

Adjunktus

Kolozsvári Dániel László

Programtervező informatikus BSc

Budapest, 2016

Tartalom

Bevezetés

Szakdolgozatom célja fordítóprogramok által használt alapvető optimalizációs algoritmusok megvalósítása, eredményük vizualizációja. A téma alapötletét az Eötvös Loránd Tudományegyetem Informatikai Karának Programtervező informatikus BSc. képzésén oktatott Fordítóprogramok című tantárgy keretein belül bemutatott módszerek, eszközök adták. A hallgató a kurzuson való részvétel során megismerkedik a fordítóprogramok működési elvének alapvető elemeivel, a lexikális, szintaktikus és szemantikus elemzők megvalósításával, valamint az assembly nyelv felépítésével. A tárgy hallgatása során bepillantást nyerhetünk a fordítóprogram-optimalizációk világába is, ezt azonban csak néhány konkrét példán keresztül tehetjük meg, nem szerzünk mélyebb gyakorlati ismereteket az adott algoritmusok elvi és működési hátteréről. Ahhoz azonban, hogy átláthassuk miért is fontosak ezek az optimalizációk, véleményem szerint nagy segítség lehetne egy olyan eszköz, amelynek segítségével a gyakorlatban is kipróbálhatjuk, és láthatjuk működés közben ezeket a technikákat.

Az általam készített alkalmazás feladata tehát annak megmutatása, hogy – elvi szinten – hogyan működnek a mindenki által napi szinten használt fordítóprogramok optimalizációs algoritmusai, milyen módosításokat végeznek el a feldolgozott kódon a bemenethez képest a sebesség, vagy épp a kisebb elfoglalt tárhely érdekében. Igyekeztem a különbségek megmutatását a lehető leglátványosabban elvégezni, ehhez több eszköz – melyek a későbbiekben bemutatásra kerülnek – segítségemre volt. Reményeim szerint a szakdolgozatom elkészítése során született program segítségül szolgálhat a hallgatóknak a tárgy elvégzésében, a fordítóprogramok működésének megértésében (hiszen láthatjuk, hogy a megadott forráskódunkból milyen absztrakt szintaxisfa generálódik, melyet majd az algoritmusaink bejárnak, az elkészült optimalizált kódból készült futtatható állományt futtathatjuk), valamint a téma iránt érdeklődőknek.

A valóságban természetesen minden fordítóprogram saját implementációval rendelkezik az adott algoritmusokhoz, számos eszközzel

biztosítanak lehetőséget fordítási és futási idejű, méret és sebességbeli optimalizációra is.

Felhasználói dokumentáció

A feladat

Célunk egy fordítóprogram megvalósítása, melynek szerves részét képezik optimalizációs algoritmusok (konstans tömörítés/változó továbbterjesztés, ciklusmagok kigörgetése, függvények beszúrása, nem elérhető kódok eliminálása). Ehhez a fordítóprogramhoz egy grafikus felületet készítek, mely a felhasználó tevékenységének hatására automatizáltan elvégzi a fordítást, a felépített absztrakt szintaxisfákat grafikusan megjeleníthető formátumban létrehozza, assembly kódot generálja és végül az elkészült assembly kódot futtathatóvá alakítja. A felületen megtekinthetőek az elkészült gráfok, valamint lehetőség van (a felhasználó által megadott) összehasonlító eszköz segítségével a generált assembly kódok összehasonlítására. Alapértelmezett eszközként a **kdiff3** alkalmazást használjuk.

A környezet

Az alkalmazás futtatásához Linux környezet szükséges (ajánlott Linux disztribúció: Ubuntu 12.04 és újabb). A szükséges állományok: **compile**, **io.c**, **Optimizer, Optimizer.sh**. A program futtatása az **Optimizer.sh** shell script futtatásával történik. A script beállítja a futtatáshoz szükséges jogosultságokat, illetve a shared library-k (dinamikusan linkelt, futási időben felhasznált „megosztott” könyvtár) eléréséhez szükséges elérési utat. Erre azért van szükség, mert futtató környezettől függően szükség lehet az **xcb-sync** könyvtár egy adott verziójára, melyet mi biztosítunk. Az **XCB** könyvtár implementálja az **X11** szabvány [1] használatához szükséges kliens oldalt (az **X11** az **X Window System core protocol** jelen verziója, ezt használják grafikus felhasználói felületek megjelenítésére többek között a Unix-alapú és Unix-szerű operációs rendszerek, melyben a szerver kezeli a különböző ki-, és bemeneti perifériákat, például a képernyőt, billentyűzetet, míg a klienst a különböző alkalmazások teszik ki). Előfordulhat azonban, hogy a teljes fordítás valamely lépése nem

hajtható végre valamely könyvtár hiánya miatt (például mikor a futtatható állományt készítjük el a **gcc**-vel), ekkor a könyvtárak biztosítása, mely az esetek túlnyomó részében megtörténik a használt eszközök legújabb verzióra való frissítésével, a felhasználó felelőssége.

A felhasznált eszközök, melyek biztosítása szintén a felhasználó felelőssége:

- **NASM** – az assembly kód feldolgozására szolgáló eszköz, ennek segítségével készítünk elf típusú bináris fájlt.
- **Graphviz** – Több eszközből álló csomag, segítségével vizualizálunk az általa várt **DOT** nyelvű leírásból gráfot

Az eszközök telepítéséhez és a futtatáshoz feltétlen szükséges, általunk nem biztosított könyvtárak beszerzéséhez szükséges információkat a **README.txt** szöveges állományban találjuk meg. Eszközök telepítése Ubuntu operációs rendszeren:

```
sudo apt-get install libc6-dev-i386
sudo apt-get install nasm
sudo apt-get install graphviz
sudo apt-get install kdiff3
```

Használat

A fordítóprogramot lehetőségünk van parancssorból, vagy az erre a célra létrehozott grafikus felhasználói felületen keresztül futtatni, ügyelnünk kell azonban arra, hogy a program olyan környezetben kell fusson, melyre van a felhasználónak írási engedélye (ez a generált fájlok létrehozásához fontos). Nézzük előbb az első esetet:

A fordítóprogram használata az **Optimizer** könyvtárban kiadott **./compile „fájlnév" „paraméterek"** parancs formájában történik, ahol a fájlnév a fordítandó forrásfájl neve, a paraméterek pedig (szóközzel elválasztva) a következők lehetnek:

- u – ciklusmagok kigörgetése
- c – konstans tömörítés és változó továbbterjesztés
- d – nem elérhető kódok eliminálása
- i – függvények beszúrása

- --verbose – plusz információk kiírása (szintaxisfák karakteres kirajzolása, annak méretének kiírása)

Ezen listát a program –help kapcsolójával is elérhetjük.

Hibás fájlmegnyitás, vagy sikertelen fordítás esetén hibaüzenetet kapunk, például lexikális, szintaktikus vagy szemantikus hiba esetén. A nyelvről, annak elemeiről és a pontos szintaxisról a **Fejlesztői dokumentáció** „**A nyelv**” című fejezetében olvashatunk bővebben. Sikeres futtatás után elkészült az optimalizálás előtti és utáni állapot alapján generált assembly állományunk (**output_old.asm** és **output.asm**), valamint a Graphviz **dot** parancsának inputjául szolgáló, szintén a két állapot alapján készült **output_graph_old.gv** és **output_graph.gv** fájlok. A következő lépés a **jpg** formátumú gráfábrázolások elkészítése, a következő parancsokkal:

- dot -Tjpg output_graph_old.gv -ooutput_graph_old.jpg
- dot -Tjpg output_graph.gv -ooutput_graph.jpg

Ekkor sikeres futás esetén az **output_graph_old.jpg** és **output_graph.jpg** képfájlok tartalmazzák az absztrakt szintaxisfánk megfelelő állapotainak reprezentációit.

Hogy futás közben is láthassuk létrehozott assembly kód működését, készítsünk futtatható állományt belőle, a következő parancs futtatásával:

- nasm -felf output.asm

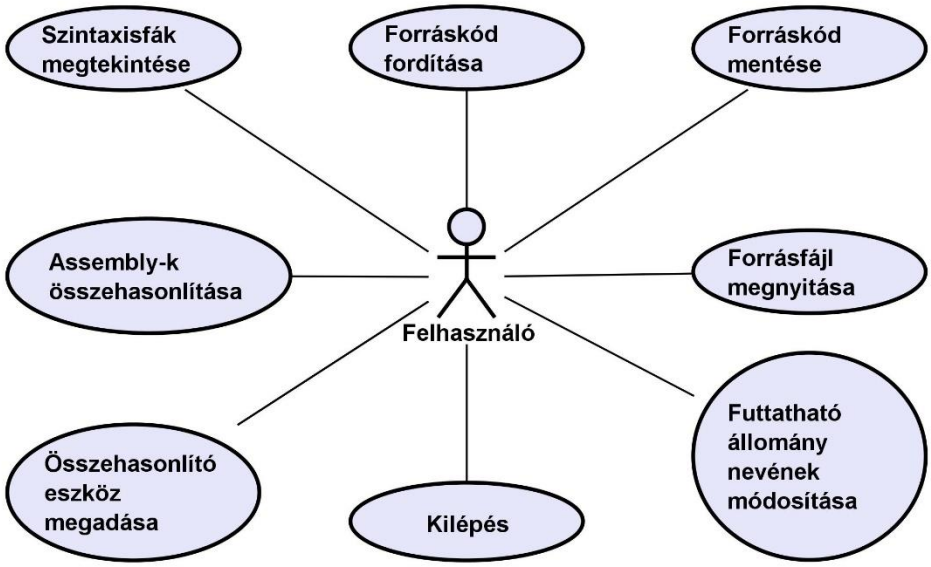
Ekkor **output.o** néven elkészül az elf típusú bináris állományunk. A következő lépésben a beolvasást és kiírást végző függvények C nyelvű kódját tartalmazó **io.c** állományt fordítjuk le, majd linkeljük össze az előbb elkészült **output.o** objectfájllal:

- gcc -m32 io.c output.o –ocompiled_o

Ezzel el is készült **compiled_o** néven a bináris futtatható állományunk, melyet a **./compiled_o** paranccsal lehet futtatni.

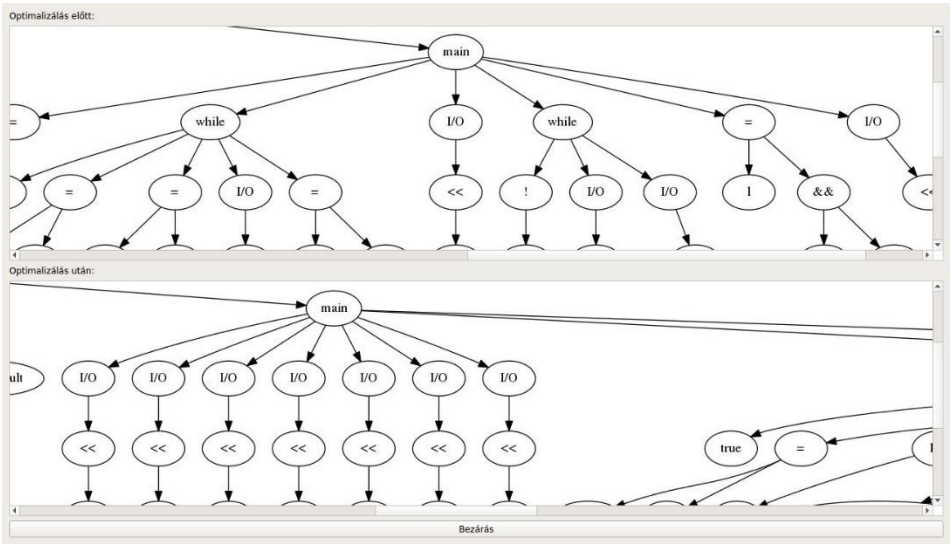
Lehetőségünk van grafikus felhasználói felületről futtatni a fordítóprogramot. Az indítás ebben az esetben is konzolból történik: **sh Optimizer.sh**; ekkor megnyílik a grafikus felület, melyen az erre a célra szolgáló szöveges beviteli mezőben

szerkeszthetjük a forráskódot. Az olvashatóságot sorszámozással és szintaxis kiemeléssel (**syntax-highlight**) segítjük. Az ablak jobboldalán, a beviteli mező mellett helyezkedik el az a szöveges mező, amelyen nyomon követhetjük a fordítás, és a különböző parancsok futtatásának állapotát, mind sikeres, mind hibás futás/fordítás esetén itt adunk visszajelzést a felhasználónak. Az optimalizációs eljárásokat az ablak jobb alsó részén található checkbox-ok segítségével tudjuk ki-, illetve bekapcsolni (alapértelmezetten mindegyiket automatikusan bekapcsoljuk).

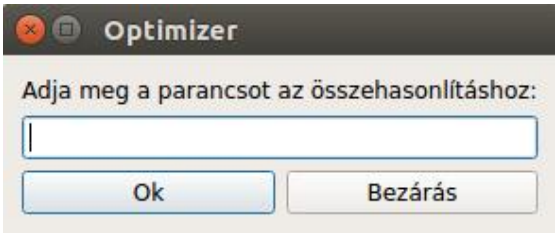


Az alkalmazást az ablak alsó részén lévő gombok segítségével tudjuk vezérelni:

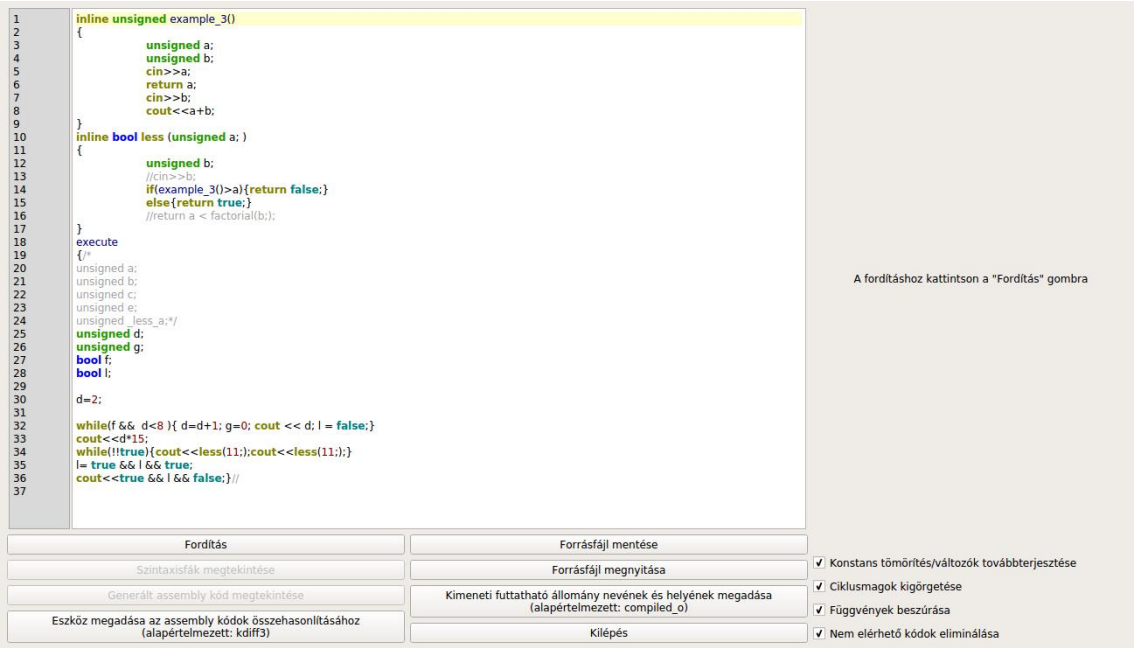
- „**Fordítás**” – A fordítóprogram elindítása a **source.input** állományra (ez az alapértelmezett forrásállomány, a később bemutatott gombbal adhatunk meg másikat), a checkbox-ok értékei alapján felparaméterezve. Lexikális/szintaktikai/szemantikus hiba esetén értesítjük a felhasználót a hibáról az említett szöveges mezőben, ahogy a sikeres fordítás esetén is. Végrehajtjuk mindazon lépéseket is, melyek a parancssori fordításnál a végső futtatható állományig vezettek el.
- „**Szintaxisfák megtekintése**” – Külön ablakban megjeleníti vertikálisan és horizontálisan is görgethető mezőben az optimalizálás előtti és utáni állapotokat tükröző, absztrakt szintaxisfákat vizualizáló gráfokat.



- „**Generált assembly kód megtekintése**” – A megadott (alapértelmezetten **kdiff3**) alkalmazást - az optimalizáció előtt és után generált assembly kódokat tartalmazó állományokat parancssori argumentumként átadva – elindítva az assembly kódok összehasonlítása.
- „**Eszköz megadása az assembly kódok összehasonlításához**” – A kdiff3 cseréje, megadhatjuk, hogy milyen összehasonlító alkalmazást szeretnénk meghívni az imént említett két fájlra.



- „**Forrásfájl mentése**” – Az éppen szerkesztett forráskód fordítás nélküli mentése.
- „**Forrásfájl megnyitása**” – Forrásfájl megnyitása (indításnál a **source.input** állományt nyitjuk meg).
- „**Kimeneti futtatható állomány nevének és helyének megadása**” – Fájlböngésző dialógusablak segítségével megadhatjuk, hogy mi legyen az elkészült futtatható állomány neve, és hova készítse el azt az alkalmazás.
- „**Kilépés**” – Az alkalmazás (és megnyitott új ablakainak) bezárása.



Indításnál a gráfok megtekintésére és az assembly kódok összehasonlítására szolgáló gombok deaktiválva vannak, hiszen ekkor még nincs mit összehasonlítani. Sikeres fordítás után aktiváljuk őket, sikertelen fordítás után azonban megint nem lesz lehetőség kattintani őket.

Fejlesztői dokumentáció

Feladat leírása

Az elkészített alkalmazás célja fordítóprogramokban alkalmazott optimalizációs algoritmusok működésének bemutatása egy saját imperatív nyelven. Ezek az algoritmusok a következők: konstans tömörítés illetve változók értékének továbbterjesztése, függvények beszúrása (inline-osítása), nem elérhető kódok eliminálása, ciklusmagok kigörgetése. Az algoritmusokról később részletesebb leírást kapunk, azonban hogy egy átfogóbb képet kaphassunk a problémáról, néhány szóban összefoglaljuk a lényegüket [2]:

- **Konstans tömörítés illetve változók értékének továbbterjesztése**: feladatunk ekkor két részre bontható. Egyfelől szeretnénk a fordítási időben kiszámítható kifejezéseket kiszámolni, legyen az akár logikai, akár numerikus eredményt adó kifejezés. Előfordulhat olyan eset, hogy nem ismerjük az adott kifejezésben szereplő összes változó értékét, azonban ha van rá lehetőség, a kifejezést olyan formára egyszerűsítjük, amilyennél egyszerűbb alakra hozni a futási időben megismert értékek felhasználása nélkül nem lehet. Másfelől ahhoz, hogy az imént leírtakat a lehető leghatékonyabban elvégezhessük, szükségünk van azon változók értékeire, melyek értéke a program futása során nem fog változni, tehát kiértékelhetjük őket a fordítás során.
- **Függvények beszúrása**: az idegen kifejezéssel „**inline**” nevezett függvények kezelése, vagyis a függvények törzsének beszúrása a programkódban a hívás helyére, abban az esetben, ha ezt megtehetjük, illetve megéri megtennünk. Nem tehetjük meg például, ha a függvényünk rekurzív, hiszen ekkor a törzs beszúrása egy újabb függvényhívást eredményezne, melynek megvizsgálása szintén, és így tovább. Sok esetben azonban nem éri meg beszúrni a függvények törzseit, hiszen azzal, hogy beszúrunk, a program futási ideje csökken, elkerüljük ugyanis a processzor számára az egyszerű mozgatásokhoz képest költségesebb ugrásokat eredményező utasításokat (**call**), a futtatandó kód azonban

jelentősen megnőhet, a beszúrt függvénytörzs mérete függvényében. Ennek érdekében meghatározzuk, hogy maximálisan mekkora függvények másolását engedélyezzük.

- **Nem elérhető kódok eliminálása**: az alapötlet az, hogy távolítsunk el a fordítás során a kódból minden olyan kódrészt, melyet a program a futás során garantáltan soha nem érhet el. Ilyenek az adott alapblokkban **return** utasítás után szereplő utasítások, az olyan elágazások illetve ciklusok törzsei, melyek feltétele soha nem lehet igaz a futás során. Eltávolítjuk ezeken kívül a program futását nem befolyásoló, a futás során feleslegesen végrehajtott utasításokat, műveleteket is. Például ha egy változónk csak értéket kap, melyet azonban soha nem olvasunk ki, mind az értékadások, mind a változó létrehozása olyan plusz költség a processzor és a memória szempontjából a futás során, mely elhagyható. Ugyanezen okból eltávolítjuk az olyan elágazásokat is, melyek törzse üres, hiszen így a feltétel kiértékelése is feleslegesen végrehajtott mozgatásokat, összehasonlításokat és ugrásokat eredményezhet a processzor számára.
- **Ciklusmagok kigörgetése**: a függvények beszúrásához hasonló az alapelv, a cél az ugró utasítások számának csökkentése. Ha már fordítási időben tudjuk, hogy a ciklusunk a futás során csak egy előre meghatározott pozitív egész számnál (**N**) kevesebbszer fog lefutni, megéri a ciklusmagot szekvenciálisan **N**-szer végrehajtani a feltétel újbóli kiértékelése, és a ciklusmag elejére való ugrás helyett. Természetesen az **N**-t úgy kell megválasztani, hogy ha a ciklusmagot szekvenciálisan beszúrjuk a ciklus helyére annyiszor, ahányszor a mag végrehajtódna, a kódunk mérete ne nőjön meg akkora mértékben, hogy emiatt már ne érje meg eliminálni a feltétel-kiértékeléseket és ugrásokat. Mivel fordítási időben mindenképp szükség van a ciklusfeltételben szereplő paramétereknek az értékére, a módszer végrehajtását csak konstans tömörítéssel együtt engedjük meg végrehajtani.

Alkalmazásunkat két nagy egységre lehet bontani. Áll egyrészt a lényegi működést biztosító back-endből, valamint a felhasználóval való kommunikációt biztosító front-endből, ami esetünkben egy asztali grafikus felület. Lehetőségünk

van a grafikus felületet elhagyva parancssorból futtatni az alkalmazást, megfelelően felparaméterezve. A back-end célja, hogy az általunk definiált nyelvben írt forráskódot lexikálisan, szintaktikusan és szemantikusan elemezze, felépítsen belőle egy absztrakt szintaxisfát, a megadott paraméterek alapján ezen végzett műveletek segítségével optimalizálja a kódot, generálja le az optimalizálás előtti valamint utáni állapotoknak megfelelő assembly utasítássorozatot, illetve a két állapotban létező szintaxisfából a **Graphviz** csomag segítségével rajzoljon ki egy-egy gráfot, ezzel könnyítve az összehasonlítást. Természetesen ha a fordítás valamelyik fázisában hiba lép fel, azt jelezzük a felhasználó felé. Sikeres fordítás esetén **Netwide Assembler** (**NASM**) segítségével az elkészült assembly kódból előállítjuk a futtatható állományt, így a kódunk elindíthatóvá válik.

A front-end-et biztosító grafikus felületű alkalmazást **Qt**-ben valósítom meg. A felhasználói felület áll egy ablakból, az abban megjelenített szövegszerkesztőből, valamint a vezérlést biztosító gombokból és checkbox-okból. A kód szerkesztését szintaxis-kiemeléssel (syntax-highlight) és sorszámozással könnyítem. Lehetőségünk van egy felugró ablakban megtekinteni az említett rajzolt gráfokat, valamint megadott (alapértelmezett **kdiff3**) grafikus összehasonlító eszköz segítségével összehasonlítani az optimalizálás előtti és utáni assembly kódot. Megadhatjuk ezenkívül a kimeneti futtatható állomány nevét. Az alapértelmezett input file az alkalmazás mellé biztosított **source.input** nevű szöveges állomány, ez azonban gomb által módosítható, fájlböngésző-ablak segítségével választhatjuk ki a szerkesztendő/fordítandó állományt. Nem létező fájl illetve hibás megnyitás esetén értesítjük erről a felhasználót. Lehetőségünk van ezeken kívül fordítás nélkül elmenteni a szerkesztett szöveges fájlunkat. A fordítás, annak lépéseinek, illetve az egyes parancsok futásának eredményéről a felhasználót folyamatosan, megfelelően színezett üzenetben értesítjük a felületet biztosító főablak egy erre a célra kijelölt mezőjében.

Mindkét részt C++ nyelven készítjük el, természetesen utóbbit a Qt-keretrendszer felhasználásával, míg előbbiben a lexikális elemzőnket működtető Flex és a szintaktikus/szemantikus elemzőnket biztosító Bisonc++ nyújtotta lehetőségek felhasználásával.

A fordítóprogram

Állományok: **compile.l**, **compile.y**, **Parser.ih**, **Parser.h**, **semantics.h**, **compile.cc**, **io.c**, **algorithm.h**, **Makefile**

Használt eszközök

Mielőtt részletesen tárgyalnánk a fordítóprogram, és a lényegi működést biztosító algoritmusok felépítését, ejtünk pár szót a használt eszközökről, melyek a program egyes moduljait biztosítják:

Flex

A Flex (Fast lexical analyzer generator) [3] egy ingyenes, nyílt forráskódú (open-source) lexikális elemző generátor, mely az elemzőt megadott formátumú, a felhasználó által biztosított fájl alapján generálja. Ez a formátum alapvetően szabályoknak nevezett reguláris kifejezés – C nyelvű kód párokat jelent, ahol a kód lehet egy, a reguláris kifejező által felismert token visszaadása, vagy más tetszőleges kód, melyre a későbbiekben láthatunk példát is (pl.: több soros komment felismerése). A megadott szabályok alapján az elemző generál egy **lex.yy.cc** nevű állományt, mely magát az elkészült elemzőt fogja tartalmazni, C++ nyelven. A fájlból generálható futtatható állomány is, a mi esetünkben ezt az állományt a szintaktikus elemző fogja felhasználni az általa használt tokenek felismerésére. Ezt úgy teszi meg, hogy a Flex által definiált osztályból példányosít egy objektumot, melynek a tagfüggvényein (definiálva a **lex.yy.cc** állományban) keresztül dolgozza fel a megadott inputfájlt. Ha a generátorunk reguláris kifejezései közül valamelyik illeszkedik az éppen olvasott szövegrészletre, lefut a hozzá tartozó C-kód. A későbbiekben láthatunk konkrét példát is az elemző generálására.

Bison

A Bison egy szintaktikus elemző (parser) generátor [4], a GNU projekt része. Mielőtt lényegi működését bemutatjuk, szükség van egy fontos definícióra:

Def.: **Környezetfüggetlen nyelvtan (grammatika)** [5]

A $G = (N, T, P, S)$ rendezett négyes környezetfüggetlen nyelvtan vagy grammatika, ha teljesülnek a következők:

- *N: nemterminális szimbólumok véges, nem üres halmaza*
- *T: terminális szimbólumok ábécéje*
- *P: véges átírási szabályhalmaz, mely szabályoknak baloldalán csak egy nemterminális szimbólum, jobboldalán pedig terminálisokból és nemterminálisokból álló szó állhat*
- *SϵN kezdőszimbólum*

A Bison megadott környezetfüggetlen grammatika alapján dolgozik, az adott nyelvtan szabályait próbálja ráilleszteni a beolvasott tokenekre, melyeket esetünkben az imént leírt Flex biztosít. Ha olyan tokensorozatot talál, melyre nem illeszthető egyetlen megadott szabály sem, szintaktikus hibát kapunk. A generátor kimenete alapértelmezetten egy **LALR** elemző [6], mely annyit tesz, hogy a kapott inputban balról jobbra olvasva talált terminális szimbólumokat lecseréljük a megadott szabályok baloldalán álló nemterminális szimbólumokra (ezt nevezük redukciónak), és így szintaktikailag helyes input esetén a végén eljutunk a kezdőszimbólumig.

A Bison a működéséhez szükséges állományokat a ***.y** fájl alapján generálja, mely megadott szintaxissal a nyelvtan leírását tartalmazza, illetve minden szabályhoz egy hozzá tartozó kódot, mely sikeres illesztés esetén lefut. Mivel csak azok a fájlok generálódnak újra, melyek ténylegesen függnek az inputtól (nyelvtan), és a kész generátor lényegi kódját tartalmazzák, az egyszer létrehozott fájlokban lehetőségünk van kiegészíteni a generátorunk kódját; ide kerülnek azok a kódok, melyekre szükségünk van a *.y állományban a szintaktikus elemzés elvégzéséhez. Erre akkor lehet szükség elsősorban, ha valójában nem csak arra vagyunk kíváncsiak, hogy a megírt forráskódunk megfelel-e a megadott nyelvtannak, hanem egyéb műveleteket is végezni akarok, például szintaxisfát akarunk építeni a beolvasott kódból.

NASM

A **Netwide Assembler** (NASM) [7] egy assembler (és disassembler) az Intel X86 architektúrához. Lehetőséget biztosít 32 és 64 bites assembly programok írására

is, a mi esetünkben a 32 bites assemblert használjuk. Ez végzi az optimalizálás előtt és után generált assembly kódunk fordítását bináris kóddá. Használata például:

az output.asm nevű állomány assemblálása (kimenet: output.o): *nasm -felf output.asm*

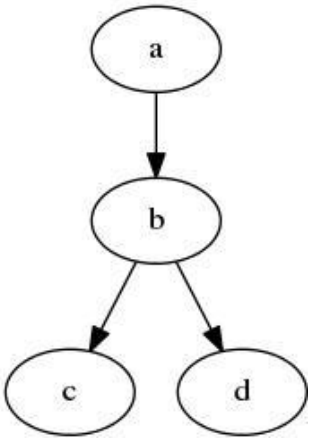
Graphviz

Nyílt forráskódú, DOT nyelven leírt gráfok ábrázolására szolgáló eszközök összessége [8]. A csomagból egy parancs képezi az alkalmazásunk szerves részét, mégpedig a **dot**, mely a hasonló nevű gráfleíró nyelvből képes megadott típusú kimeneti állományba grafikusan megjeleníthető gráfot generálni. Pl.:

dot -Tjpg output_graph.gv -ooutput_graph.jpg

output_graph.gv:

```
digraph G {
a -> b;
b -> c;
b -> d;
}
```



A nyelv

Ahogy korábban említettük, az algoritmusok bemutatásához definiálunk egy saját nyelvet. Ez a nyelv egy C-szerű imperatív nyelv [9], szintaxisában törekedtünk az ahhoz való hasonlóságra.

Felépítése azonban mégis kissé eltérő: a főprogramunkat az **execute** utasítás jelöli, melyet kapcsos zárójelek követnek. Főprogramnak kötelezően szerepelnie kell a kódunkban, törzse azonban – ahogy a függvényeké is – lehet üres. Az execute előtt lehetőségünk van függvények definiálására, ügyelnünk kell azonban arra, hogy a kód egy adott pontján csak azokat a függvényeket tekintjük definiáltnak, melyek az adott függvény (vagy a főprogram) definícióját megelőzik. Hasonló módon a függvénytörzsekben is két főbb szekciót különböztetünk meg: a deklarációs szakasz, melyet a végrehajtási törzs követ. A deklarációs

szakaszban van lehetőségünk változók deklarálására (definiálására), utána ezt azonban nem tehetjük meg, csak olvashatunk vagy írhatunk a változókba, ahogy a deklarációs szakaszban sem fordulhat elő más utasítás (Pl.: kezdeti értékadás a következő módon néz ki, a további deklarációkat *-gal jelölve: * unsigned a *; a = 5;)

Függvényeket a C nyelvhez hasonlóan definiálhatunk: „típus” „függvénynév” („paraméterek”){„törzs”} (ügyeljünk azonban arra, hogy a törzsnek két része van, igaz, mindkettő lehet üres. A paraméterek szintén opcionálisak, típust és azonosítót azonban kötelező megadni). A paraméterek mindegyikét pontosvessző (**;**) követi.

A nyelvünk alapvetően kétféle típust különböztet meg: **unsigned**, és **bool**. Míg előbbi egy 4 bájton tárolt előjel nélküli egész számot ábrázol, addig utóbbi 1 bájtos logikai érték. Változók definíciója a C-nél megszokotthoz hasonlóan történik: „típus” **„**azonosító” (Pl: unsigned a;). Megkülönböztetünk ezeken kívül egy harmadik **void** típust is, mellyel a visszatérési értékkel nem rendelkező függvények definiálására van lehetőségünk. A nyelvünk elemei, magyarázattal:

- számok: pozitív egész számok.
- azonosító: karakterlánc, változók nevét vagy függvényneveket jelöl, számokat, betűket, vagy a ’_’ karaktert tartalmazhatja, azonban nem kezdődhet számmal.
- **execute**: a főprogram, futás során ennek a törzse hajtódik végre
- **unsigned**, **bool**, **void**: Típusok, függvények visszatérési értékének, vagy változók (első kettő) típusát jelölik.
- **true**, **false**: Igaz, illetve hamis értékeket jelölő kulcsszavak, a **bool** típus lehetséges értékei.
- **if**, **while**: Egyirányú elágazás illetve ciklus. A kulcsszót az elágazás/ciklus feltétele követi zárójelben, mely soha nem lehet üres. Utána következik a függvény törzse kapcsos zárójelek között, ez azonban lehet üres.
- **else**: Kétirányú elágazásnál használatos: szintaxisa az egyirányú elágazáshoz hasonló, azonban a kapcsos zárójelet egy **else**

kulcsszó után újabb kapcsos zárójel-páros követi, mely a feltétel hamis kiértékelése esetén kell végrehajtanunk.

- **cout**: A kiíró utasítás első fele, kötelezően a **<<** operátor követi.
- **cin**: A beolvasó utasítás első fele, kötelezően a **>>** operátor követi.
- **<<**: A kiíró utasítás második fele, kötelezően a **cout** kulcsszó előzi meg.
- **>>** A beolvasó utasítás második fele, kötelezően a **cin** kulcsszó előzi meg.
- **return**: Visszatérést jelöl. **execute** törzsében semmi nem szerepelhet utána, ahogy **void** típusúként definiált függvény törzsében sem. **unsigned** vagy **bool** visszatérési értékű függvény esetén ellenben kötelezően szerepelnie kell utána egy megfelelő típusú kifejezésnek.
- **inline**: Függvények definíciójában szerepelhet, a típust megelőzve. Azt jelöli a fordítóprogram számára, hogy lehetőség szerint (ha be van kapcsolva a megfelelő opció) végezze el a függvény beszúrását a függvényhívás helyére. A kulcsszó megadása ellenére a fordítóprogram dönthet úgy, hogy nem éri meg beszúrni a függvénytörzset annak mérete miatt, ilyenkor az **inline** kulcsszót figyelmen kívül hagyja.
- *** / - +**: **unsigned** típusú értékek (változók, konstansok, kifejezések) közti aritmetikai műveletek (szorzás, egészrészű osztás, különbség, összeg).
- **%**: modulo, szintén **unsigned** típusok között.
- **< >**: összehasonlító operátorok **unsigned** típusok között (kisebb mint, nagyobb mint), visszatérési értékük **bool** típusú lesz.
- **==**: összehasonlító operátor (egyenlő-e), azonos típusú értékek összehasonlítását végzi.
- **&& ||**: és, illetve vagy operátorok, **bool** típusú értékek közötti műveleteket végeznek
- **!**: **bool** típusú kifejezés negálása.
- **;**: pontosvessző, utasítások végét jelöli, illetve a paramétereket választja el.

- **=**: értékadás operátora.
- **{ }**: Kapcsos zárójelek. Függvénytörzseket, ciklus és elágazásmagokat határol.
- **( )**: zárójelek. Kifejezések kiértékelésének módosítására, ciklus/elágazásfeltétel megadására, függvényparaméterek felsorolására használható.
- ***//***: egy soros komment. A sor végéig tartó kódrészletet nem vesszük figyelembe.
- ***/* */***: több soros komment. A megadott operátorok közötti kódrészletet nem vesszük figyelembe.

A bemutatott operátorok, precedencia szerint csökkenő sorrendben:

- *** / %**
- **+ -**
- **( )**
- **< >**
- **==**
- **!**
- **&& ||**

A lexikális elemző feladata a forráskód tokenizálása a felsorolt nyelvi elemek szerint. A megfelelő reguláris kifejezés – kód (megfelelő token visszaadása) párokat a **compile.l** állományban adjuk meg, a Flex ez alapján fogja elkészíteni a lexikális elemzőnket. A felismert tokenek fogják jelölni a nyelvet leíró grammatikánkban a terminális szimbólumokat. Hibás input (nem ismert nyelvi elem) esetén lexikális hibát kapunk.

A nyelvtant a **compile.y** fájlban írjuk le, a Bisonnak megfelelő szintaxis alapján. A környezetfüggetlen grammatika, mely alapján a szintaktikus elemzőnket generáljuk (a könnyebb olvashatóság érdekében az egymást követő szimbólumokat szóközzel választjuk el, illetve jelöljük a nemterminális szimbólumokat csupa nagybetűvel, a terminálisokat pedig kis betűkkel, vagy a megfelelő szimbólummal, pl. zárójel), a következő:

START -> FUNCTIONS EXECUTE

FUNCTIONS -> ε | FUNC FUNCTIONS
FUNC -> INLINE FUNCTION_TYPE AZON (PARAMETERS) { DEKLARACIO
POSTTORZS }
INLINE -> ε | inline
AZON -> azonosito
FUNCTION_TYPE -> void | TIPUS
PARAMETERS -> ε | PARAM PARAMETERS
PARAM -> TIPUS azonosito ;
EXECUTE -> execute { DEKLARACIO POSTTORZS }
DEKLARACIO -> ε | DEKL DEKLARACIO
DEKL -> TIPUS azonosito ;
TIPUS -> unsigned | bool
TORZS -> IO ; POSTTORZS | ERTEKADAS ; POSTTORZS |
while (KIF) { POSTTORZS } POSTTORZS |
if (KIF) { POSTTORZS } POSTTORZS |
if (KIF) { POSTTORZS } else { POSTTORZS } POSTTORZS |
FUNCTION_CALL ; POSTTORZS | return RET_VAL ; POSTTORZS
RET_VAL -> ε | KIF
POSTTORZS -> ε | TORZS
IO -> cout << KIF | cin >> azonosito
ERTEKADAS -> azonosito = KIF
IN_PARAM -> ε | IN_P IN_PARAM
IN_P -> KIF ;
FUNCTION_CALL -> azonosito (IN_PARAM)
KIF -> FUNCTION_CALL | (KIF) | KIF + KIF | KIF – KIF | KIF * KIF | KIF / KIF |
KIF % KIF | KIF == KIF | KIF < KIF | KIF > KIF | KIF && KIF | KIF || KIF | ! KIF |
azonosito | szam | LOGIKAI
LOGIKAI -> true | false

Az azonosítókat (függvények, változók nevei) az **azonosito**, a számokat pedig a **szam** terminális szimbólumok jelölik. Mint láthatjuk, a nyelvtanban nem szerepelnek a kommentek. Ez azért van így, mert a kommenteket nem tokenizáljuk, hanem bár a lexikális elemző felismeri és elfogadja őket, az átugrásukon kívül nem végez más műveletet.

Az Absztrakt szintaxisfa (AST)

A Bison segítségével tehát az iménti nyelvtan alapján generálunk egy szintaktikus elemzőt, a szemantikus elemzéshez azonban másra is szükségünk van. Célunk a szemantikus helyesség ellenőrzésén kívül egy absztrakt

szintaxisfa felépítése is, hogy lehetőségünk legyen ezt a fát bejárva később az optimalizációkat elvégezni. Az AST egy fa gráf, mely a kód szerkezetét hivatott absztrakt módon bemutatni, tehát szülő-gyerek viszonyok meghatározásával mutatja meg, hogy épül fel a kódunk, például mik a kódban szereplő alapblokkjaink.

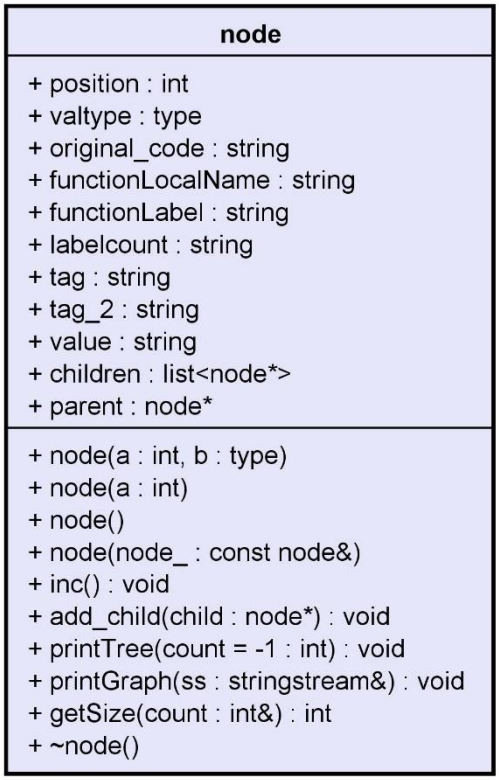
Def.: **Fa** [10]

Egy gráf fa, ha összefüggő, és nincs benne kör. Az összefüggőség annyit tesz, hogy bármely csúcsából bármely csúcsába vezet út vagy séta (tehát komponenseinek száma 1). Nincs benne kör, tehát nincs benne olyan legalább 1 hosszú zárt vonal, melynek kezdő és végpontja megegyezik, a többi csúcs azonban különböző.

Def.: **Alapblokk** [11]

Egymást követő utasítások sorozata alapblokkot alkot, ha csak az első utasításra lehet kívülről átadni a vezérlést, illetve az utolsó utasítás kivételével nincs benne vezérlést átadó utasítás (pl.: visszatérés, ugrások, függvényhívás).

A következő lépés tehát, hogy megismerjük azt az osztályt, melynek példányai alkotják majd a fa csúcsait. Ez az osztály a **node** osztály, definíciója a **semantics.h** állományban található.



Az input szövegben való elhelyezkedést (sorszám) a **position** mező tárolja. Célja a felhasználó tájékoztatása arról, hogy szemantikus hiba (pl: újradeklaráció, típushibás műveletek) hol történt a kódban. Emellett az elemző szintaktikus hiba estén is közli a megfelelő sorszámot, de ez a mező nem azt az értéket tárolja, hiszen szintaktikus hiba esetén az elemző az éppen vizsgált pozíciót adja vissza, míg újradeklaráció esetén szeretnénk értesíteni a felhasználót, hogy a problémás változó vagy függvény hol volt először deklarálva. A **valtype** tárolja az aktuális csúcsban szereplő kifejezés típusát (ha szükség van rá). Értékei a következők lehetnek: **unsignedtype**, **boolean**, **notype**. Az értékek rendre az unsigned, bool, és void típusok megfelelői. Előbbi kettőt változók, függvények, kifejezések esetén is használjuk, míg utóbbit csak függvények és azok visszatérési értékei esetén.

A lényegi információt az **original_code** mező tartalmazza. Itt tároljuk, hogy az adott csúcs milyen utasításhoz, kifejezéshez tartozik. Ez alapján végezzük az optimalizációt. A konkrét algoritmusokon kívül számos helyen felhasználjuk, mindenhol, ahol információra van szükségünk a kód valamely részéről, mivel itt tároljuk a deklarált változók neveit is, egyszóval a szintaxis minden fontosabb elemét, mely információ nélkülözhetetlen a kóddal kapcsolatos műveletek során. A **functionLocalName** az assembly generálásnál használatos,

itt tároljuk az adott függvényekhez tartozó változók egyedi, a függvénynevet is tartalmazó neveit. Erre azért van szükség, hogy egy függvény meghívása esetén a hívott függvény lokális változója ne ütközzön a hívó függvény lokális változójával, vagyis a különböző scope-ok elkülönítését szolgálja. **functionLabel**: Szintén az assembly kód generálásánál fontos, azt tárolja, hol végződött egy beszúrt függvény törzse, illetve visszatérés esetén egy adott függvényből pontosan melyik utasításnál kell folytatnunk a végrehajtást. A **labelcount** egy segédmező, segítségével különböztetjük meg az összetartozó assembly-címkéket. Például el tudjuk különíteni, hogy egy ciklust tartalmazó elágazás esetén mely ugró utasítások tartoztak az elágazáshoz, melyek a benne lévő ciklushoz. A mező értékét konkatenáljuk a megfelelő címkenévvel, így kapunk egyedi elnevezéseket. A **tag**-ben tárolok minden plusz információt, mely a node példány jellegétől függően szükséges. Ez alapján döntöm el többek között, hogy egy adott változó értéke az adott ponton ismert-e fordítási időben, hogy milyen típusú kifejezésről beszélünk, vagy hogy függvény-beszúrás esetén az abban szereplő return utasítás mely függvényhez is tartozott. Ehhez hasonló a **tag_2** is, vagyis ez is adott helyzetekben fontos plusz információt hordoz, ennek egyetlen célja azonban az, hogy a nem elérhető kódok eliminálásának algoritmusa figyelmen kívül hagyja az adott változót, ha a mező értéke alapján erre van szükség. Gyakorlatban ez a függvények beszúrásánál fontos, azt az újonnan létrehozott csúcsot jelöljük meg ugyanis, mely a beszúrt törzs után kerül a kódba, jelölve a függvény visszatérési értékét, ez ugyanis a hívó függvény lokális változói közé újonnan bekerült visszatérési változó által valósul meg, melyet semmilyen esetben nem törölhetünk. A **value**-ben tároljuk a csúcshoz tartozó tényleges értéket, ha az már fordítási időben ismert. Olyan csúcsoknál fordul elő, melyek változót tartalmaznak, a **node** példány ebben az esetben (melyhez az említett **value** mező tartozik) nem része a fának, erről azonban később ejtünk szót.

A **children**-nel valósítjuk meg a tényleges fa szerkezetet, ez a lista ugyanis az adott csúcs gyerekeinek a címeit tárolja. A fa bejárása során ezen a listán fogunk végigiterálni, lefedve így az egész gráfot. Hasonló célt szolgál a **parent**, itt tároljuk a csúcs szülőjének a címét, melyből természetesen mindig pontosan egy van, kivéve a fa gyökerét, melynél ez a mező a 0 értéket veszi fel.

Most, hogy már ismerjük az osztály mezőit, melyik milyen célt szolgál, szóljunk pár szót az osztály tagfüggvényeiről is, hiszen ezek is fontos szerepet töltenek be a működés során:

A konstruktorok célja hogy a megadott mezőkön kívül a **labelCount** és a **parent** mezőket 0-ra inicializálják (előbbit string formájában, tehát „0” kezdőértékkel indul). Rendelkezünk egy copy-konstruktorral is, mely a megfelelő mezők értékeinek másolásán kívül a létrehozott objektum **children** adattagjában felsorolt gyerekeit is újonnan létrehozza, majd azoknak is rekurzívan. Erre azért van szükség, mert előfordul hogy a fa egy adott részfájára van szükségünk, nem tehetjük meg azonban, hogy módosítjuk a kész AST csúcsainak értékét, szükségünk van tehát a részgráf egy temporális másolatára. Az **inc** a **labelCount** értékének karbantartását végzi, szükség esetén ezzel tudjuk növelni a tárolt számot. Hasonlóan kényelmi funkciót valósít meg az **add_child** függvény, az új gyerek hozzáadásán kívül beállítja szülőnek az aktuális csúcsot is. A fejlesztés, és a parancssori használatot segítő metódus a **printTree**, a megadott csúcstól kezdve (amelyikre meghívjuk a függvényt) a teljes fát rekurzívan bejárva „kirajzolja” nekünk a gráfot olyan formában, hogy minden csúcsnál jelöli annak szülőjének sorszámát (a számlálót a gyökérnél indítjuk 0-ról), így látjuk azt, hogy egy adott csúcs a fa melyik szintjéhez tartozik. Szintén az AST rajzolása a célja a **printGraph**-nak is, ez azonban a Graphviz számára megfelelő DOT nyelvű inputot generálja, erre fogjuk meghívni a működés során a dot parancsot. A fejlesztés során elsősorban hibakeresésre használt függvény a **getSize**, a meghívott csúcstól kezdve bejárva a fát megadja nekünk a részgráf méretét (a csúcsok darabszámát). Segítségével könnyen láthatjuk például hogy egy adott optimalizációs eljárás változtatott-e a szintaxisfa méretén, avagy nem. Egy másik fontos felhasználási módja a **getSize**-nak a ciklusmagok kigörgetése és a függvények beszúrása esetén jelenik meg, ugyanis ahogy azt korábban írtuk, a műveletek végrehajtása függ a vizsgált részfa méretétől, melyet ez a függvény ad meg nekünk. Megvalósítottuk ezeken kívül a destruktort is, melyre a gyerekek megfelelő felszabadítása miatt volt szükség.

A szemantikus elemző

A szintaktikus elemzőnk sikeres illesztés esetén létrehozza a megfelelő **node** típusú objektumot, hozzáadja a megfelelő gyerekeket, elvégzi a szükséges összehasonlításokat (létezik-e már a változó, műveletek részkifejezései megfelelő típusúak-e, stb.). A fa gyökerét a **Parser** osztály egy tagja tárolja, sikeres szintaktikus elemzés esetén kap értéket. Ez az osztály a **ParserBase** leszármazottja, definíciója a **Parser.h** fájlban kap helyett. Mind ez, mind a tagfüggvényeit megvalósító **Parser.ih** állomány generált fájlok, melyeket csak egyszer hoz létre a Bison, utána szükség esetén módosíthatjuk, és nem generálódik újra. A **Parser** példányosítja a lexikális elemzéshez szükséges objektumot is, mi pedig ezen osztály egy példányán keresztül érjük el az elemzőnket, ennek adjuk át a vizsgálni kívánt kódot, valamit az AST gyökerét (**root** adattag) is ezen keresztül érjük el.

Ahhoz azonban, hogy a változó-, és függvénydeklarációkat kezelni tudjuk, szükség van még egy eddig nem említett lényeges dologra, mégpedig a szimbólumtáblára. Ahogy korábban említettük, szükség van olyan speciális node-okra, melyek nem tartoznak a fához. Ezek azok, melyek a változók megfelelő értékeit (**value**) is tárolják. A szimbólumtáblát a **Standard Template Library** (**STL**) egy tárolójával, a **map**-pel valósítjuk meg, mely string-ekhez rendel node* típusú objektumokat. Amikor deklarálunk egy változót, felveszünk egy új elemet a szimbólumtáblába. Az azonosító a létrehozott változó neve lesz, a létrehozott új objektum pedig a változóról tárolt információkat fogja tartalmazni (kapott-e már értéket, mi az értéke, stb.). Mivel nem egyetlen scope-hoz tartozó változókat szeretnénk csak kezelni, szükség van még egy **map** típusú tárolóra, hiszen meg szeretnénk engedni, hogy több függvényben is definiálhassunk azonos nevű változókat. Ehhez string-ekhez rendelünk korábban leírt map típusú objektumokat. Minden definiált függvényhez fog tehát tartozni egy megfelelő szimbólumtábla, mely az adott függvényhez tartozó változókról hordoz információkat. Mind a „külső”, mind a „belső” map példány a **Parser** osztályban szerepel, hogy azokat szükség esetén a gyökérhez hasonlóan könnyen elérhessük. Míg a „bővebb”-et referencia szerint átadva kezeljük mindenhol, így közvetlenül ezt az objektumot módosítjuk, addig a „szűkebb” értéke attól függ,

melyik függvényt vizsgáljuk éppen. Szükség lesz magukról a függvényekről tárolt speciális információkra is: hogy gyorsan elérjük, hogy az adott függvény inline-ként volt-e definiálva, mik a paraméterei, vagy épp a típusa, felveszünk egy „**1a**” stringhez rendelt csúcsot a szimbólumtáblába. Az **original_code** mező fogja tartalmazni, hogy szükség lehet-e függvénybeszúrásra („inline” vagy „no_inline” értéket vesz fel), a **valtype** tárolja a függvény típusát, illetve könnyítésképpen a függvényparamétereket felvesszük a csúcs gyerekei közé, így nem kell mindig bejárnunk az adott részfát, mikor szükség lenne ezekre az információkra (például a függvényhívások vagy függvénybeszúrás esetén). A másik ilyen speciális csúcs a „**2a**”-hoz tartozó eleme lesz a szimbólumtáblának, ez magára a függvény fában elhelyezkedő gyökerére mutat. Ezen keresztül érjük el a függvény nevét (**original_code**), vagy szükség esetén az egész részfát. Láthattuk a nyelv leírásánál, hogy nem engedjük azt meg, hogy azonosító számmal kezdődjön, így soha nem lehet ütközés ezekkel a speciális elemekkel.

A főprogram

Főprogramunk a **compiler.cc** fájlban helyezkedik el. A forrásfájl megnyitása (parancssori argumentumként nem megadott fájlnév, vagy sikertelen megnyitás esetén hibát adunk) után példányosítjuk a **Parser**-t, majd a **parse** metódus meghívásával megtörténik a lexikális, szintaktikus és szemantikus elemzés, valamint felépül az AST-nk, és feltöltődnek a szimbólumtábláink. A generált assembly kódot (a generálásról később ejtünk szót) az **output_old.asm** állományba mentjük, a DOT nyelvű gráfleírást pedig a **output_graph_old.gv**-be. Hogy milyen optimalizációt szeretnénk alkalmazni, illetve szeretnénk-e plusz információkat a standard output-ra írni (fa soros kirajzolása, méret kiírása), parancssori argumentumként kapjuk meg:

- u – ciklusmagok kigörgetése
- c – konstans tömörítés és változó továbbterjesztés
- d – nem elérhető kódok eliminálása
- i – függvények beszúrása
- --verbose – plusz információk kiírása

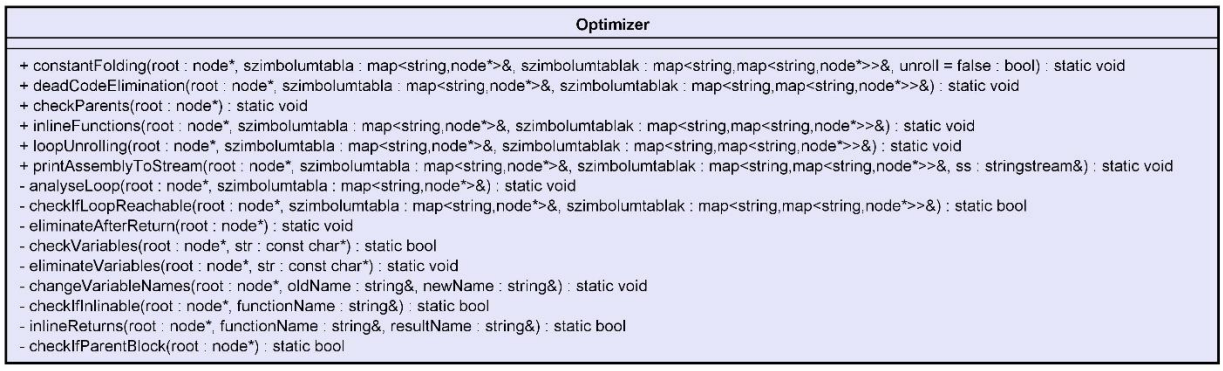
Ezen listát a program –help kapcsolójával is elérhetjük.

Mivel egy, a szintaxisfára meghívott eljárás után a fa olyan helyzetbe kerülhet, melyen korábban el nem végezhető optimalizációt is el lehet végezni (például: kiértékelt ciklusfeltétel után tudjuk, hogy szükség van-e eliminálásra, és ha igen, már végezhetünk műveleteket a ciklusmagban szereplő változók ismert értékeivel a ciklus után is), annyiszor hívjuk meg egymás után a megfelelő algoritmusokat, amíg a gráf mérete nem változik, ez alapján állapítjuk meg, hogy már nem tudunk változtatni az AST-n.

Miután már nem változik a fánk, létrehozzuk az optimálizálás utáni állapotot képviselő **output.asm** állományt, illetve a DOT nyelvű **output_graph.gv** állományt. Ezután lehetőségünk van a **Felhasználói dokumentációban** leírtak szerint összehasonlítani a kész assembly kódokat, futtatható állományt létrehozni vagy gráfot rajzolni.

Az optimalizációs algoritmusok

Alkalmazásunk lényegi részét az optimalizációk megvalósítása adja. Az algoritmusokat az Optimizer osztály statikus függvényeként implementáltam. A könnyebb kezelhetőség érdekében, mivel az osztálynak egyetlen adattagja sincs, nem is példányosítjuk soha, az átadott függvényargumentumokon végezzük el a módosításokat:



Megkülönböztetünk lokális és globális optimalizációkat. Lokálisnak nevezünk egy optimalizációt akkor, ha egy adott alapblokkon belül történik (definíciót lsd.: **Az Absztrakt szintaxisfa (AST)** című fejezet). A mi esetünkben lokális például a konstans tömörítés, mely során ugye egy adott kifejezést vizsgálunk, globális a változók továbbterjesztése. Bár ha eszerint szeretnénk tagolni a műveleteinket,

külön algoritmus végezhetné őket, a könnyebb megvalósítás érdekében azonban ugyanaz a függvény végzi mindkettőt.

Konstans tömörítés, változók értékének továbbterjesztése

Az eljárást a **constantFolding** rekurzív (ahogy az összes többi, AST-t bejáró algoritmusunk is rekurzív) függvény valósítja meg, mely paraméterként a vizsgálni kívánt (rész)fa gyökerét, a szimbólumtáblákat (ezeket várják egyéb, hasonló algoritmusaink is argumentumként), és egy logikai értéket vár. Utóbbi hordozza magában azt, hogy szeretnék-e ciklusmag-kigörgetést is végezni, hiszen ahogy korábban is írtuk, ez nem lehetséges konstans tömörítés nélkül. Az algoritmus kezdetén megvizsgáljuk, szükség van-e egyáltalán folytatni a működést, illetve megengedett-e (van-e az adott csúcsnak gyereke, deklarációt vizsgálunk-e, stb.), illetve szükség esetén aktualizáljuk a szimbólumtáblát (melyik függvényhez tartozóra van szükségünk?) A továbbiakban a függvény viselkedését az esetek túlnyomó részében az határozza meg, hogy mi szerepel az aktuálisan vizsgált csúcs **original_code** mezőjében. Ha ciklust vizsgálunk, a **checkIfLoopReachable** metódussal ellenőrizzük, hogy jelen ismereteink szerint végrehajthatjuk-e legalább egyszer a ciklusmagot a futás során. Ha igen, a szimbólumtábla megfelelő értékeit módosítva „elfelejtjük” az eddigi ismereteinket (**analyseLoop**) a ciklusmagban szereplő változók értékéről, hiszen azzal sem a magban, sem a feltételben, sem a ciklus utáni kódban nem számolhatunk, illetve ha szükséges, meghívjuk a **loopUnrolling** függvényt, mely a mag-kigörgetést végzi majd. Hasonlóan járunk el a változók értékei szempontjából elágazások esetén is, hiszen ha nem tudjuk biztosra hogy soha nem teljesül egy adott feltétel, nem tudunk mit mondani az elágazásban szereplő változók értékeiről az azt követő kódban. Szintén nem használjuk a továbbiakban az olyan változók értékeit, melyek értékadás baloldalán szerepelnek úgy, hogy a jobboldalon álló kifejezés értéke ismeretlen, vagy ha a beolvasó (**cin >>**) kifejezés jobboldalán álnak (ekkor biztos, hogy a kapott érték csak futási időben lesz ismert). Algoritmusunk eddig a változók továbbterjesztésének előkészítésével foglalkozott, miután azokat a műveleteket elvégezte, következhet a lényegi konstans tömörítés. Két nagy esetet különböztetünk meg: a kifejezés, melyet vizsgálunk unsigned, vagy boolean típusú. Ha unsigned, szintén két eset áll fenn; az első esetben az adott kifejezésünk minden tagja (mivel algoritmusunk

rekurzívan működik, ekkor az adott levelet megelőző szinten vagyunk a fában) ismert értékű változó vagy konstans. Ekkor a megfelelő műveletet (vagy összehasonlítást, pl.: **<**) elvégezve lecseréljük a kifejezés gyökerét képviselő csúcsot az eredményt tartalmazó csúcsra. Láthatjuk, hogy összetettebb kifejezés esetén is ugyanígy kell eljárnunk, hiszen a legszűkebb részkifejezéseket kiértékelve, egyre nagyobb részt feldolgozva a végén az egész kifejezésünk egyetlen konstans értékre lesz lecserélve. A második esetben a kifejezésnek legfeljebb az egyik tagját ismerjük, például egy összeadás esetén vagy az egyik, vagy mindkét operandus ismeretlen értékű változó. Mivel ekkor nincs lehetőségünk műveletvégzésre, más technikát alkalmazunk: aszerint járunk el, hogy milyen információink vannak a vizsgált kifejezés gyökerének testvéréről (a szülő másik ágáról a fában), ha van ilyen. Ha a másik ág konstans vagy értékelt változó, és a vizsgált művelet felcserélhető a szülő műveletével a végrehajtás során (összeadás-kivonás, kivonás-összeadás, szorzás-szorzás, osztás-osztás), elvégezzük a megfelelő műveletet a szülő másik ága és a vizsgált kifejezés ismert tagja között, az eredmény a szülő másik ágába kerül, a szülő ezen ága pedig az ismeretlen értékű változó lesz. Ha a szülő másik ága ismeretlen értékű változó, tehát nem tudunk műveletet végezni, és teljesülnek az imént leírt műveleti követelmények, megcseréljük az ismert értékű csúcsot a szülő másik ágával, így „görgetve fel” az ismert értéket a fában, hogy ha később egy másik kifejezés során tudnánk vele egyszerűsíteni, ezt megtehessük. Erre példa az a 1 + a + b + 2 eset, ahol az „a”, „b” változók értékeit nem ismerjük. Sem az 1 + a, sem a b + 2 kifejezés nem egyszerűsíthető, ha az ismert konstansokat a szabályok megtartásával áthelyezzük a fában azonban, ezt a kifejezést kapjuk: 1 + 2 + b + a. Itt már van lehetőségünk összevonásra, az 1 + 2-t 3-ra cserélhetjük, így az egyszerűsített kifejezés a következő lesz: 3 + b + a.

Eddig arról az esetről beszéltünk, ha unsigned típusú értékekkel foglalkoztunk, ejtsünk pár szót hát a bool típus esetéről is. Itt összesen 4 operátorral kell foglalkozunk: az összehasonlítás és negálás során ha nem ismert értékekkel dolgozunk, nem teszünk semmit, érdekesebb azonban a logikai és (**&&**) és logikai vagy (**||**) esete. Mindkettőnél hasonlóan járunk el: logikai és esetén ha mindkét értékünk ismert, és igazra kiértékelt (logikai vagy esetén hamisra kiértékelt), akkor a művelet cserélhető az igaz (hamis) értékre. Ha

mindkét érték ismert, és legalább az egyik operandus értéke hamis (vagy esetén igaz), akkor a művelet hamis értéket kap (vagy esetén igazat). Ezek után mindkét műveletnél egyetlen eset marad: és esetén az egyik operandus értéke igaz, a másik azonban ismeretlen, logikai vagy esetén pedig az egyik biztosan hamis, a másikat szintén nem ismerjük. Ekkor az ismert érték elhagyható, így a műveletet az ismeretlen értékű csúcsra cseréljük (Pl.: a && true = a illetve a || false = a).

Nem elérhető kódok eliminálása

Az algoritmust a **deadCodeElimination** metódussal valósítjuk meg, paraméterei a **constantFolding**-hoz hasonlóan a vizsgálandó gyökér és a szimbólumtáblák. Első lépésként ellenőrizzük, hogy ciklust vizsgálunk-e, és ha igen, a korábban leírtakhoz hasonlóan kezelnünk kell a ciklusmagban szereplő változókat, vagyis sem a magban, sem a ciklus után nem állíthatunk semmit az értékükről. Ezután elkezdjük a tényleges kód-eliminációt; Ha olyan elágazást észlelek, melynek üres a törzse, azt törölöm, hiszen a kész program futását csak lassítja a felesleges feltétel kiértékelés. Ugyanezen okból kifolyólag eltávolítom az olyan kétirányú elágazásokat is, melyek törzse mindkét ág (igaz, illetve hamis) esetén is üres. Ha egy ilyen utasításnak csak az első (igaz) ága üres, akkor az egész utasítást egyirányú elágazásra cseréljük, az eredeti feltételt pedig egy plusz művelet hozzáadásával negáljuk. Eltávolítom a fából azokat a részfákat is, melyek gyökere olyan egyirányú elágazást vagy ciklust képvisel, amelyek feltétele már fordítási időben ismert, és hamis (false) értékű, hiszen ebben az esetben biztosan tudjuk, hogy a törzsben olyan kód szerepel, melyet futás során nem érhetünk el, így csak növeli a futtatható állomány (és a kész assembly kód) méretét és komplexitását. Ha egy egyirányú elágazás feltétele biztosan igaz (true), az elágazás csúcsát eltávolítjuk, a törzset pedig beszúrjuk az eltávolított csúcs helyére, hiszen ez az eset épp az ellentéte az előzőnek, vagyis olyan kódrészlettel van dolgunk, melyet minden esetben biztosan elérünk. Nem ejtettünk még szót arról, mi történik akkor, ha a feltétel fordítási időben ismert, a vizsgált kódrész azonban kétirányú elágazás. Ha a feltétel igaz, akkor az elágazást egy egyirányú elágazással helyettesítjük, tehát elhagyjuk a hamis ág törzsét (mivel ahogy azt korábban írtuk, az algoritmusok többszöri alkalmazása szükséges a lehető legjobb eredményű optimalizáció eléréséhez, a **deadCodeElimination** következő alkalmazásánál ennek az elágazásnak is

kibontjuk a törzsét, az elágazást pedig elhagyjuk). Ha a feltétel hamis, beszúrjuk a hamis ág törzsét a kétirányú elágazás helyére.

Természetesen nem csak elágazások és ciklus törlésére van szükség a kód méretének hatékony csökkentéséhez. Szükség van például az olyan változók eltávolítására is (**checkVariables**), melyekre a futás során alapvetően nincs szükség. Ez akkor fordul elő, ha egy változó deklarálva van, azonban sehol nem használjuk a kódban, vagy ha használjuk, akkor is csak értéket kap, tehát az általa hordozott információra soha nincs szükség. Eliminálom még az AST azon részfáit is, melyek **return** (visszatérés) utasítást követnek (**eliminateAfterReturn**), ügyelve arra, hogy az eliminációt csak a return utasítás csúcsa utána következő testvérein végezzük el. Így nem fordul elő, hogy például egy elágazásban lévő return utasítás után az elágazás után szereplő kódot is eltávolítjuk.

Függvények beszúrása

Az **inlineFunctions** függvény az olyan csúcsokra koncentrál, melyek **original_code** mezőjének értéke „function_call”, vagyis függvényhívást jelölnek, és a függvényt az **inline** kulcsszóval definiáltuk. Fontos kritérium még azonban az is, hogy a függvény nem lehet rekurzív, ezt a **checkIfInlinable** végzi. Ekkor leellenőrzöm, hogy a beszúrandó függvénytörzset képviselő részfa az általunk megadott határ alatt van. Ezt a határt 50-nek választottuk meg, hogy megfelelően lehessen szemléltetni az elég kicsi függvények beszúrását, illetve a túl nagyon beszúrásának ignorálását. Ha szükség van a beszúrásra, felépítek két temporális fát. Az egyik a függvény paramétereit, és lokális változóit fogja tartalmazni (**tempParameters**), a másik pedig a beszúrandó törzset (**tempBody**). Az előbbi felépítése a következőképpen történik: végigiterálunk a függvény paraméterein, majd a deklarációs szakaszban létrehozott lokális változóin, és mindegyiket eltároljuk egy ideiglenes csúcsban (**tempVar**). Megváltoztatjuk a nevüket oly módon, hogy beszúrjuk elé a „függvénynév”+’_’ stringet, ahol a „függvénynév” az őt tartalmazó függvény neve (Pl.: meghívjuk az **f**() függvényt, és f-nek van egy **variable** nevű változója. Ekkor az új név **f_variable lesz**). Megkeressük a hívó függvény deklarációs szekcióját, majd ellenőrizzük, hogy az újonnan létrehozott változónév szerepel-e már a deklarált változók között. Ha igen, ’_’ karaktereket

szúrunk be elé mindaddig, amíg ez nem teljesül (Pl.: tegyük fel, hogy az előző esetben a hívó függvénynek már volt **f_variable** nevű változója. Ekkor az új név **_f_variable** lesz, és így tovább). Ha végeztünk az átnevezésekkel, a **changeVariableNames** metódus segítségével elvégezzük a névcserét a vizsgált változó minden előfordulásánál az ideiglenes fában, mely a függvénytörzs másolatát tartalmazza.

A változók egyedivé tétele és hozzáadása a deklarációkhoz után a következő lépés a függvény visszatérési értékének, illetve az ezt visszaadó **return** utasítások kezelése. Ehhez szintén létrehozok egy új csúcsot, a **resultNode**-ot. Erre azért van szükség, mert a visszatérési értéket egy új változóban fogjuk tárolni, melyet a hívó függvény deklarációi között hozunk létre. A változó a **return** utasításoknál kap értéket. Az új változó neve a következő alakú: '_' + „függvénynév" + '_result', ahol „függvénynév" ebben az esetben is a hívott függvény azonosítója (Pl.: meghívjuk az **f()** függvényt, ekkor egy **_f_result** nevű új változót hozunk létre). Természetesen az ütközések elkerülése végett itt is alkalmazzuk az iménti technikát, vagyis '_' karaktereket szúrunk az azonosító elejére addig, amíg az azonosító egyedi nem lesz.

Az **inlineReturns** ellenőrzi, hogy szerepelnek-e az (ideiglenes) függvénytörzsben **return** utasítások, és ha igen, **jump** nevű csúcsokra cseréli őket, **tag**-nek pedig beállítja a hívott függvény nevét. A visszatérési értéket a **return** csúcs gyereke tartalmazta, illetve művelet esetén az a részfa, melynek gyökere az utasítás gyereke volt, ezért a **jump** csúcs elé beszúrunk egy új értékadást, mely során az imén létrehozott eredményváltozónak (**resultNode**) adjuk értékül a visszatérési értéket. Ezután a **jump** csúcs gyerekét töröljük (a korábbi visszatérési értéket), majd hozzáadunk egy új gyereket, mely az eredményváltozó nevét tartalmazza. Erre azért van szükség, hogy ha kódeliminálást is alkalmazunk, ne töröljük az eredményváltozót akkor sem, ha azt máshol nem használnánk, hiszen mivel ekkor csak értékadásban szerepelne, algoritmusunk szerint eliminációra lenne szükség. Ha az **inlineReturns** igazat ad vissza, szükségünk lesz még egy új csúcsra, az **endNode**-ra, mely a **function_end** nevet kapja, **tag**-nek pedig beállítjuk a hívott függvény nevét, majd ezt a csúcsot hozzáfűzzük a törzs részfájához. Ez a csúcs fogja meghatározni,

hogy a korábbi **return** utasítások helyetti értékadások után honnan kell folytatnunk a működést (**jump**).

Miután kezeltük a visszatéréseket is, és létrehoztuk az eredményváltozót, beszúrjuk a függvény törzsét. Ehhez a **checkIfParentBlock** segítségével megkeressük a fában a gyökér felé haladva az első olyan csúcsot, mely szülője alapblokkot képviselne, ha nem fordulnának elő benne **return** utasítások, majd ide, a talált utasítás elé beszúrjuk a függvénytörzset, a függvényhívás csúcsában tárolt információt pedig lecseréljük az eredményváltozóra. A **tag** mezőbe „ignore” kerül akkor, ha a hívott függvény típusa **void**, egyébként „”. Erre az assembly kód generálásánál lesz szükség, hogy ne próbáljuk meg kiértékelni az értéket nem tartalmazó változót. Láthatjuk, hogy ebben az esetben fontos például az, hogy az eredményváltozót akkor sem töröljük, ha az csak értékadás baloldalán szerepel. Az utolsó két lépés némi magyarázatra szorul: a cserére azért van szükség, hogy ezzel eltávolítsuk a kódból a korábbi hívást, helyét pedig a visszatérési érték vegye át. Az első lépésben elkövetett visszalépések magyarázatához nézzük meg, milyen formában fordulhatnak elő függvényhívások: lehet önálló utasítás, ekkor elég lenne egyszerűen beszúrni a függvénytörzset a hívás elé. Szerepelhet például egy értékadás jobboldalán, ekkor már a szülő csúcs (az értékadás) elé kéne beszúrni a törzset, a hívás helyett pedig csak az eredményt kiértékelni. A visszalépésekre az utóbbi, és annál bonyolultabb, de hasonló esetekben van szükség; a cout << a + b + f() * c; esetén meg kell keresnünk, hogy hova kell beszúrnunk a függvénytörzset úgy, hogy semmit ne rontsunk el. Jelen példánknál ez a cout-ot tartalmazó csúcs lesz, ez elé kell beszúrni a törzset, a függvényhívást pedig lecserélni az eredményre, így a kapott kifejezés: …függvénytörzs…cout << a + b + _f_result * c; lesz.

A ciklusmagok kigörgetése

A műveletet a **loopUnrolling** metódus végzi abban az esetben, ha a **constantFolding** meghívja. Ez függ egyrészt attól, hogy megadtuk-e a programnak a megfelelő paramétert (ahogy a többi optimalizációs algoritmusunk esetén is), illetve hogy a ciklus magja elérhető-e. Ha nem elérhető, tehát a feltétele fordítási időben is ismerten hamis, akkor felesleges megpróbálnunk a kigörgetést.

Létrehozunk egy új fát, mely a kigörgetendő ciklus feltételét és magját fogja tartalmazni, az utasítás neve azonban **while**-ról **analysed**-ra változik, hogy ne vizsgáljuk meg többször ugyanazt a ciklust. Létrehozunk egy új, temporális szimbólumtáblát, melybe lemásoljuk az aktuális függvény szimbólumtábláját, ezzel elkerüljük azt, hogy miközben szimuláljuk a ciklus működését, elrontsuk az ismert változók értékeit. Ahogy említettük, ez az algoritmus is egy meghatározott korláttól függően dönti el, hogy elvégezhető-e az optimalizáció avagy sem. Ezt az értéket 10-nek választottuk meg, vagyis ha a ciklusmagot 10-nél kevesebbszer hajtanánk végre a futás során, engedélyezzük a kigörgetést. A szimuláció során egy végrehajtjuk a magot annak egy másolatán (konstans tömörítéssel és változók továbbterjesztésével együtt), majd megnézzük, hogy a végrehajtás után teljesül-e a ciklusfeltétel. Ha igen, folytatjuk a szimulációt egészen addig, míg el nem érjük a megadott korlátot. Miután véget ért ez a lépés, megvizsgáljuk, hogy a szimuláció során hamissá vált-e a ciklusfeltétel. Ha igen, az azt jelenti, hogy a magot a korlátnál kevesebbszer kell végrehajtanunk, és elvégezzük a kigörgetést; beszúrjuk a ciklus elé a ciklusmagot szekvenciálisan annyiszor, ahányszor a mag a futás során végrehajtódott volna, az eredeti **while** utasítás kódját (**original_code**) pedig **unrolled**-ra cseréljük. Ezzel jelezzük a **constantFolding**-nak, hogy itt sikeres optimalizáció történt, és az elvégzi majd az eredeti ciklus törlését a fából.

Igaz, az algoritmusok önmagukban is elvégzik a kívánt optimalizációs eljárásokat, hatékonyságuk egymástól is jelentősen függ, a kiváltott hatást akkor tudjuk maximalizálni, ha valamennyi algoritmus futását engedélyezzük. Láthattuk, hogy van, amelyik a futási időt csökkenti a fordítási időben elvégezhető műveletek fordítási időben való elvégzésével, vagy épp az elkerülhető processzorműveletek kihagyásával, illetve van, amelyik a kódméretet csökkenti, redukálva ezzel a futtatható állomány méretét. Bár az eljárások nagyon hasonlítanak a való életben, fordítóprogramok által használt algoritmusokhoz, ahogy korábban is említettük, a célunk a szemléltetés, a nyelv komplexitása is ennek megfelelő, így az eredmény nem minden esetben hordoz szignifikáns különbséget a kiindulási állapothoz képest (a futtatható állomány méretét és a futási időt tekintve), az elkészült gráfok és assembly kódok azonban megfelelően szemléltetik célunkat.

Az Assembly

A generált assembly kód binárissá történő feldolgozását a **NASM** végzi, arról azonban még nem ejtettünk szót, hogy mi felelős maga a kód elkészültéért. Mind optimalizálás előtt, mind utána kiírom a korábban említett állományokba a kész kódot, melyet a fájlba írás előtt egy **stringstream**-be építünk fel, a **printAssemblyToStream** metódus segítségével. A függvény alapvetően bejárja az AST-t, és a csúcsok által képviselt kód (**original_code**) alapján beírja a megfelelő utasításokat a streambe, a már megszokott és széles körben használt mintákat (pattern) felhasználva. Korábban említettük a **node** osztály leírásánál a **labelcount** adattagot, ez most fog jelentős szerepet kapni, ennek segítségével különböztetjük meg az összetartozó címkéket az összes többitől. Például egy ciklus vagy elágazás esetén jelölnünk kell, hogy milyen feltételek teljesülése esetén honnan kell folytatnunk a végrehajtást (**jump**), a konkrét helyet pedig az egyedi címke jelöli. Az egyedi címkék label_+ „labelcount” felépítésűek, kivéve a vizsgált elágazás/ciklus végét jelölő címkék, ezek end_label_ + „labelcount” formában jelennek meg. Külön kezeljük a beszúrt függvényekhez tartozó címkéket, ezekbe magának a hívott függvénynek is belefűzzük a nevét. Ezek azok a címkék, melyeket a korábbi **return** utasítások helyett elvégzett ugrások során felhasználunk.

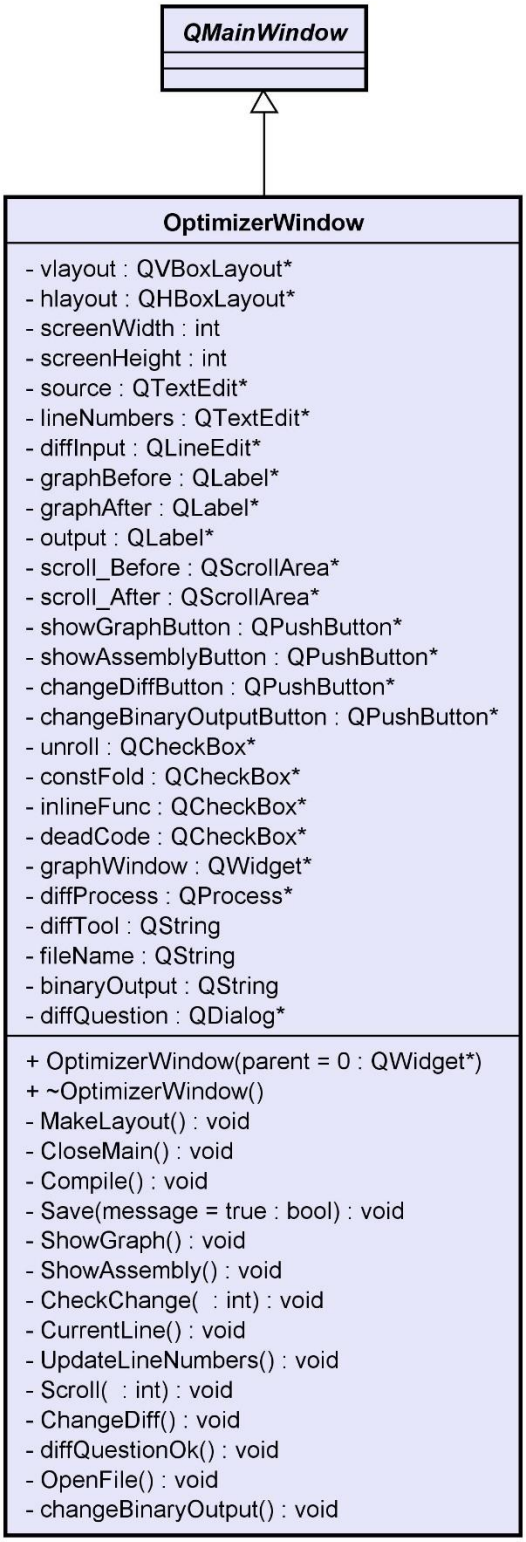
A kiírást és beolvasást egy másik állományban (**io.c**) definiált C függvények adják, ezeket az assemblyből létrehozott object-fájllal linkeljük össze, elkészítve így a végleges futtatható állományt. Példákat a kódgenerálásra a **Tesztelés** fejezetben láthatunk. A végső futtatható állomány létrehozása a generált kódból a **Felhasználói dokumentáció** fejezetben lett részletesen bemutatva.

A felhasználói felület

A grafikus felhasználói felületet **Qt**-ban valósítom meg [12], az **Optimizer** projekttel. Mivel a felület elrendezését, és a megjelenést a kódból közvetlenül szabályozom, nem használjuk például a Qt adta lehetőségeket a grafikus elemek szerkesztésére, az ehhez kötődő fájlokat és kódrészleteket (kommentelve)

megtartottam, a későbbi fejlesztések lehetőségét szem előtt tartva. A lényegi működést megvalósító állományok tehát a következők: **Optimizer.pro**, **main.cpp**, **optimizerwindow.h**, **optimizerwindow.cpp**.

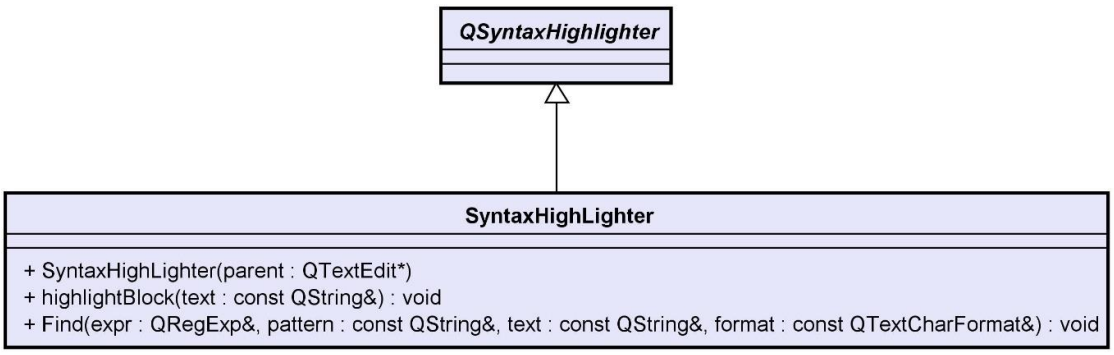
Az **Optimizer.pro** a projekt alapvető beállításait tartalmazza, többek között hogy mely Qt keretrendszerbeli modulokra lesz szükségünk, vagy milyen forrásfájlokat használunk fel. A **main.cpp**-ben példányosítjuk az **OptimizerWindow** osztályt, létrehozva így a megjelenítendő főablakot, valamint indítjuk el magát az alkalmazást, mely meg is jeleníti azt. Az **OptimizerWindow**-t az **optimizerwindow.h** header-fájlban definiáljuk:



Az osztály a **QMainWindow** leszármazottja, így az alapvető funkcionalitást nem nekünk kell megvalósítanunk, a főablakot egy ablak-sablonra építve hozhatjuk létre. Az elrendezésért a **vlayout** és **hlayout** adattagok a felelősek, értékeiket futás során dinamikusan változtatjuk, attól függően, hogy melyik ablak (a főablak, vagy a gráfok megjelenítéséért felelős) mely részének megjelenítésével

foglalkozunk. Hogy az elrendezés mind a hely kihasználtságát, mind a felhasználói élményt tekintve a lehető legoptimálisabb legyen, a konstruktorban eltároljuk a **screenWidth** és **screenHeight** attribútumokba az aktuális képernyő szélességét és magasságát, a megjelenített elemek mérete és pozíciója ezen változók értékétől függően alakul majd.

A könnyű kezelés érdekében a fordítandó forráskód szerkesztésére megjelenítek egy szövegszerkesztő mezőt (**source**), melyet a szövegbeviteli mezők alapvető tulajdonságain felül plusz funkciókkal ruházok fel. Megvalósítunk például sorszámozást (**lineNumbers**), mely az előzőhöz hasonlóan egy **QTextEdit** típusú mezőben jelenik meg. A **source** mező **textChanged** jelének (**signal**) hatására az **UpdateLineNumbers** segítségével változtatom a sorszámokat tartalmazó mező tartalmát. Szintén a szinkront valósítja meg a **Scroll** függvény, mely a beviteli, és sorszámokat tartalmazó mezők **QScrollBar** típusú, **verticalScrollBar** függvényével elért objektum **valueChanged** jelére reagál és az aktuális értékre állítja be a szövegszerkesztő mezők görgetősávjának értékét, elérve így, hogy az egyiket görgetve a másik vele együtt változzon. A használat kényelmesebbé tétele érdekében szöveg-, és sorkiemelést (syntax-highlight) is alkalmazok. A sorkiemelést a **cursorPositionChanged** jel hatására a **CurrentLine** végzi, mely a teljes sorra kiterjedően megváltoztatja a kurzor pozíciójának hátterét. A szövegkiemelés megvalósításához származtatok egy új osztályt (**SyntaxHighLighter**) a **QSyntaxHighlighter** osztályból:



Az osztályból példányosítok egy objektumot (**highlighter**), oly módon, hogy a konstruktornak átadjuk a **source**-t paraméternek, mint szülő objektum, innentől kezdve a szöveges mező minden változásánál megpróbálja ráilleszteni (**Find**) a

szövegre a megadott reguláris kifejezéseket (**QRegExp**), egyezés esetén pedig az adott formázást (**QTextCharFormat**) a vizsgált blokkra illesztve színezi át a kódot. A kiemelő azokat a nyelvi elemeket ismeri fel, melyeket korábban felsoroltunk, például konstansok, kulcsszavak vagy kommentek.

A felhasználó felé való visszajelzésekért a **QLabel** típusú **output** objektum a felelős. A fájldialógusokat leszámítva minden esetben itt értesítem a felhasználót a fordítás, illetve az egyes parancsok (**dot**, **nasm**, **gcc**) futtatásának sikerességéről. Fordítási hiba esetén itt írom ki, hogy a fordítás mely fázisa volt sikertelen, szintaktikai hiba esetén az mely sorban történt, szemantikus hiba esetén pontosan mi volt a hiba. A figyelemfelkeltés érdekében a hibát jelző kiírásokat pirosra színeztem, míg a pozitív/semlegeseket zöldre (sikeres fordítás, diff-tool futtatása, stb.).

A vezérlést gombokkal és **checkbox**-okkal valósítom meg. A checkboxok jelzik azt, hogy mely optimalizációs eljárásokat kívánjuk elvégezni, ezek a program indulásánál alapesetben ki vannak pipálva. Ennek az az oka, hogy a korábban leírtak szerint az algoritmusok akkor a leghatékonyabbak, ha azokat együtt (egymás után többször) hajtjuk végre. A megfelelő gombra kattintva (**compileButton**) a fordítás megkezdése előtt ellenőrizzük, hogy mely optimalizációs eljárásokat választotta ki a felhasználó, eszerint hozza létre a parancssori argumentumokat, mellyel külön folyamatként (**QProcess**) futtatjuk magát a fordítóprogramot. A **Compile** függvény deaktiválja az AST-k megjelenítésére, és az assembly-k összehasonlítására szolgáló gombokat (**setDisabled**), majd megpróbálja elmenteni a szerkesztett kódot a megnyitott forrásfájlba. Ha ez nem sikerül, vagy bármilyen későbbi művelet során hiba lép fel, értesítjük a felhasználót a hibáról, majd visszatérünk a függvényből, megszakítva ezzel a fordítás folyamatát. A fordító sikeres futtatása után a **dot** paranccsal generáltatom a gráfot a megfelelő **jpg** formátumú fájlokba, majd lefordítom az assembly-t elf típusú object-fájllá, melyet a lefordított **io.c** állománnyal összelinkelve elkészítem a végső futtatható állományt. Sikeres végrehajtás esetén újra engedélyezzük a korábban deaktivált gombokra való kattintást. A futtatható kimenet nevét a felhasználó meghatározhatja, szintén gombra kattintva (**changeBinaryOutputButton**), melynek **clicked()** jelzésére reagálva a **changeBinaryOutput** párbeszédablak segítségével megváltoztatja a

binaryOutput változó értékét, melyet a fordítás során felhasználunk a futtatható állomány nevének meghatározására. A megfelelő gomb szövegét az új névhez igazodva megváltoztatjuk. Hasonló elvvel van lehetőségünk az alapértelmezett **source.input** állomány helyett másik forrásfájl megnyitására (**openFileButton**) is, ekkor a megfelelő adatot szintén fájldialógus felhasználásával kérjük be (sikeres megnyitás esetén az ablak címét is megfelelően frissítjük). Természetesen minden fájlművelet sikertelensége esetén (mentés, megnyitás) értesítjük a felhasználót a hibáról egy felugró ablak (**QMessageBox**) formájában.

Megadhatunk új összehasonlító eszközt is az assembly kódok összehasonlításához (**changeDiffButton**). A **showAssemblyButton** kattintásának hatására lefutó **ShowAssembly** a **diffTool** mező értékét olvasva hívja meg a külső összehasonlító alkalmazást, mely mező értékét az imént említett gombra kattintva megváltoztathatjuk, egy **QDialog** felugró ablakban megjelenített **QLineEdit** típusú egysoros szerkesztésre szolgáló szöveges beviteli mező (**diffInput**) segítségével.

A szintaxisfák megjelenítését **showGraphButton** kattintására reagáló **ShowGraph** végzi egy új ablakot létrehozva (**graphWindow**), a korábban **dot** paranccsal generált képeket az ablak felületére helyezve. Mivel a képek (és a megjelenítendő gráf) mérete a futás során ismeretlenek, **QScrollArea** típusú mezőben helyezem el azokat a **QLabel** típusú objektumokat, melyek a képek megjelenítését (**setPixmap**) végzik majd. Így ha a megjelenítendő kép a mérete miatt túlnyúlna az őt megjelenítő objektum határain, vertikális és horizontális görgetés segítségével lehetőségünk van az AST-k egészét megtekinteni.

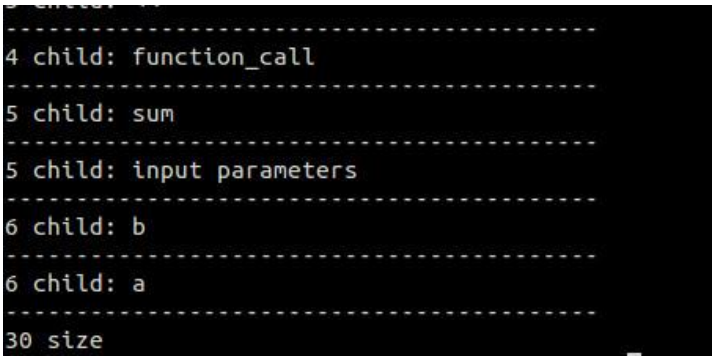
Tesztelés

Vizsgáljuk ebben az esetben a fordítóprogram parancssorból történő indítását, hisz a grafikus felhasználói felület is ezt teszi, új folyamat indításának formájában.

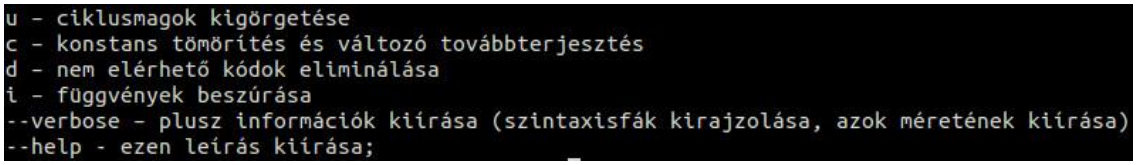
Futtassuk a **compile** állományt fájl megadása nélkül, ekkor a következő hibaüzenetet kapjuk: „A forditando fajl nevet parancssori parameterben kell megadni.”

Hasonló üzenetet kapunk akkor is, ha nem létező állományt adunk meg: futtassuk a compile állományt a „not_exists.cc” fájl megadásával. Ekkor a kapott hibaüzenet: „A not_exists.cc fajlt nem sikerult megnyitni.”

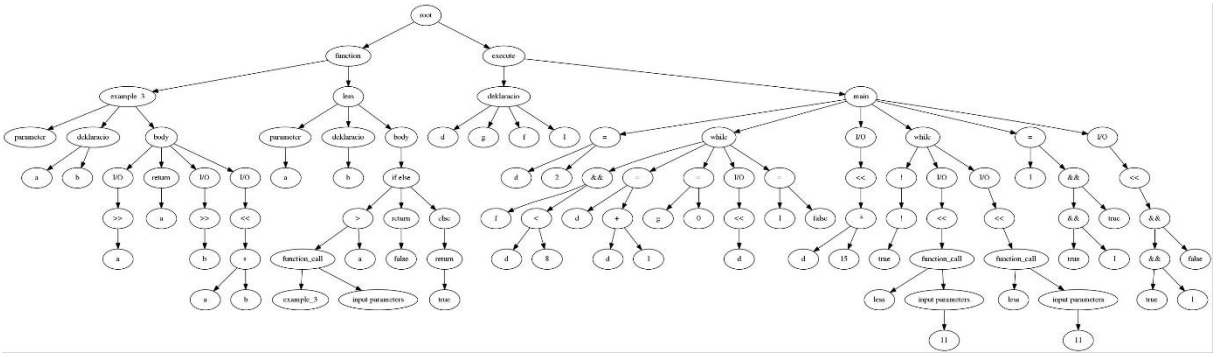
A használni kívánt optimalizációs eljárásokat kapcsolókkal aktiválhatjuk, ahogy a segítség kérése (**--help**) és fordítás eredményének kiírása (AST soros kiírása) is ilyen módon történik. Futtassuk a compile állományt a **source.input** előre biztosított forrásfájllal. Helyes futás esetén ekkor semmilyen visszajelzést nem kapunk. Tegyünk meg most ugyanezt, de a **–verbose** kapcsoló alkalmazásával, ekkor a megfelelő output:



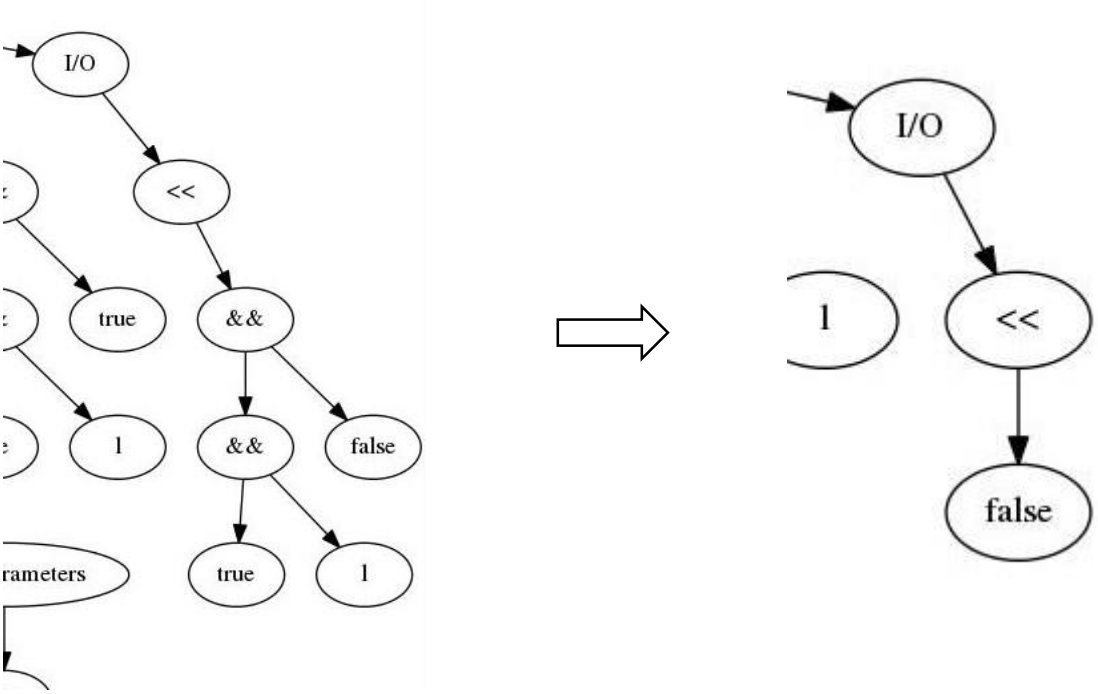
Ha megadjuk a **–help** kapcsolót, a program a segítség kiírása után nem végez más műveletet, és terminál:



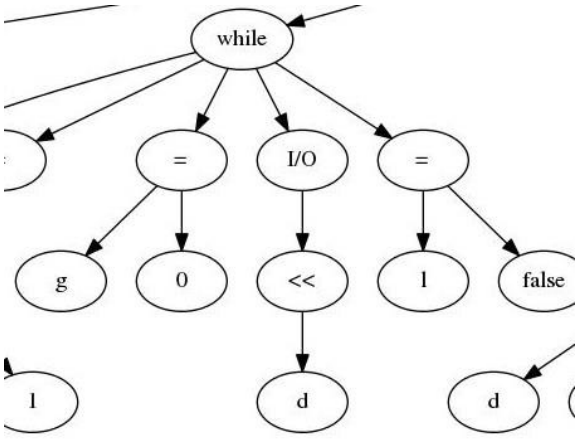
Magához a fordításhoz és optimalizáláshoz kapcsolódó teszteket a grafikus felhasználói felületen végezzük, hiszen az is az előbbi esethez hasonlóan indítja a fordítóprogramot, a megfelelő parancsok futtatásával azonban le is generálja a megfelelő gráfokat (**output_graph.jpg**), így könnyebb szemléltetni az eredményeket. Először kapcsoljuk be egyenként az optimalizációs eljárásokat (egyszerre csak egyet, kivéve a ciklusmagok kigörgetésének esetét, hiszen ezt nem végezhetjük változók továbbterjesztése/konstans tömörítés nélkül), és figyeljük meg, hogyan változik a szintaxisfa a kiinduló állapothoz (töröljük az összes pipát) képest:

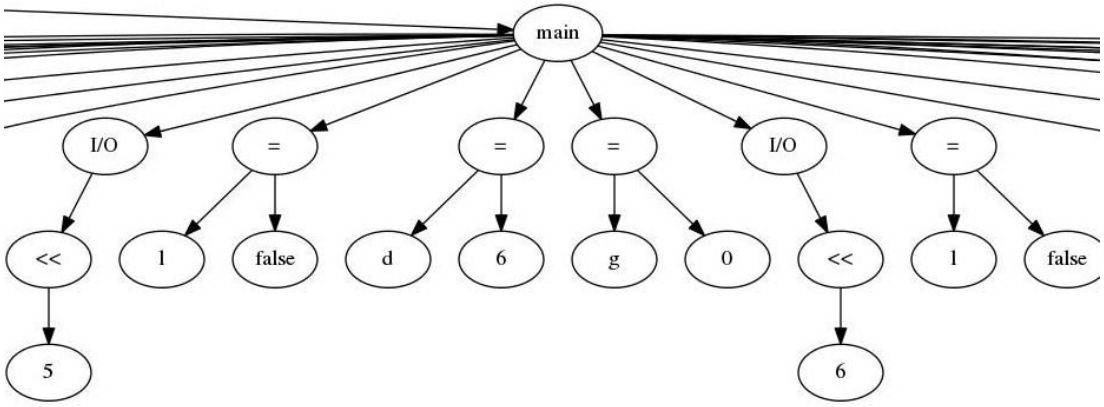


- **Konstans tömörítés/változók továbbterjesztése**:

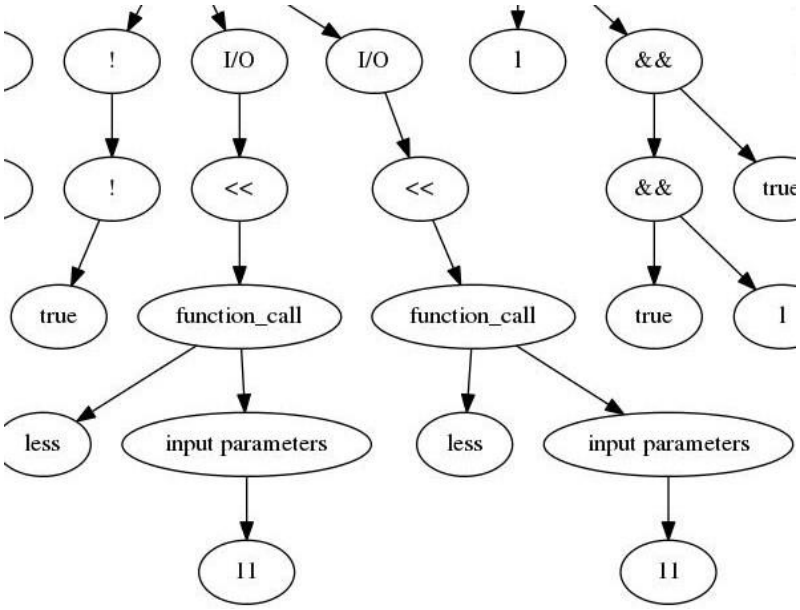


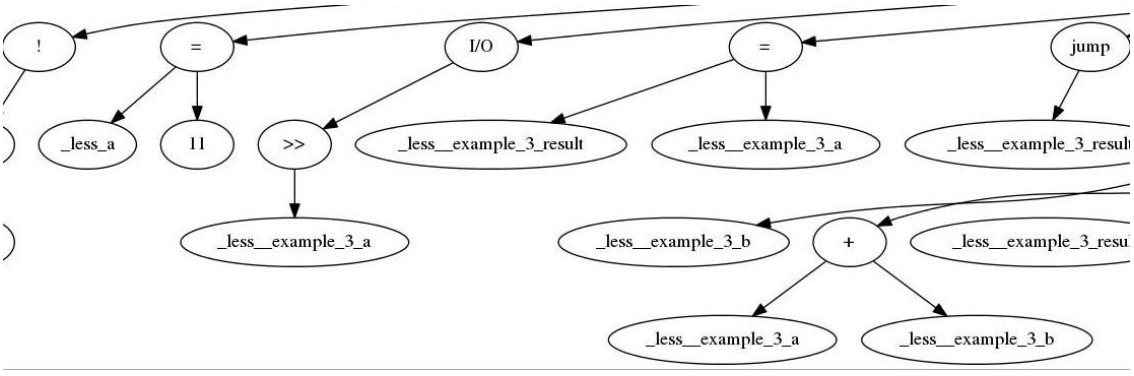
- **Ciklusmagok kigörgetése**:



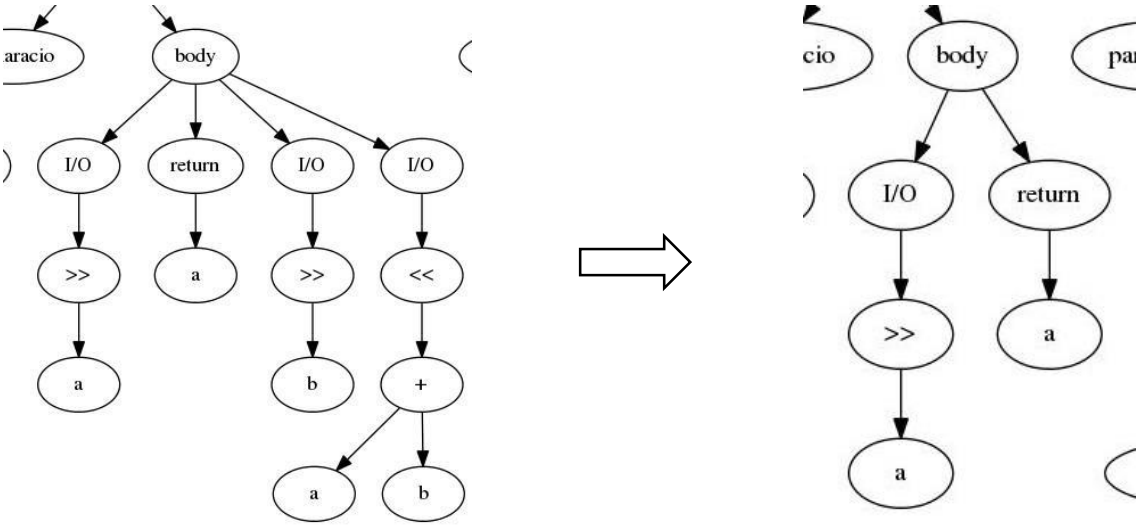


- **Függvények beszúrása**:

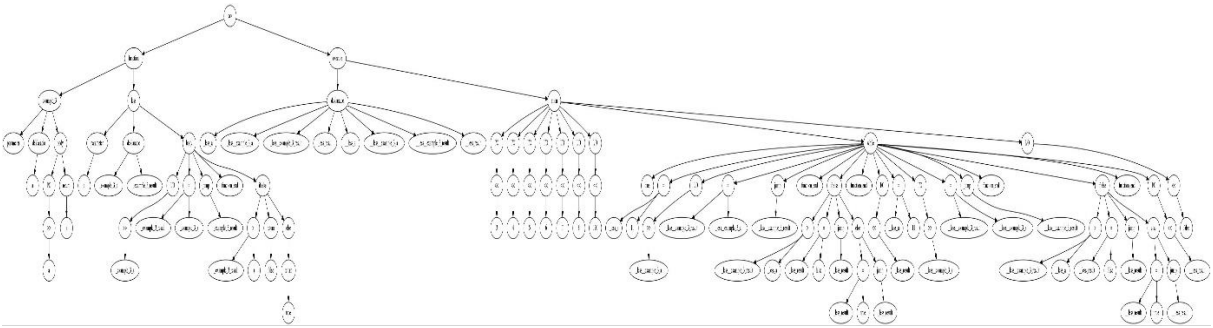




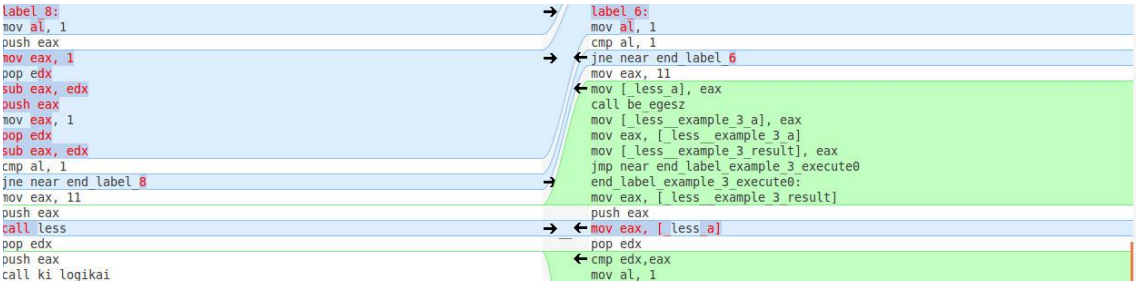
- **Nem elérhető kódok eliminálása**:



Láthatjuk, hogy minden eljárás a leírtaknak megfelelően változtatja meg a fát, ahogy a kész assembly kódot is, melynek szemléltetéséhez kapcsoljuk be valamennyi eljárást:



Ekkor a generált optimalizálatlan kód egy részlete, a generált optimalizált kód megfelelő részletével összehasonlítva (használt eszköz: meld):



Teszteljük most a felhasználói felület egyéb funkcióit. Adjunk meg egy új összehasonlító eszközt, ekkor láthatjuk, hogy a gomb felirata megváltozik. Kattintsunk **Generált assembly kód megtekintése** gombra, és láthatjuk, hogy az új eszköz indul el. Ha olyan nevet alkalmazást adunk meg, amely nem elérhető (pl.: nincs telepítve), vagy üresen hagyjuk a mezőt, és elfogadtatjuk, majd futtatni

akarjuk, a következő hibaüzenetet kapjuk: ’Nem sikerült futtatni a „parancsnév” parancsot!’. Sikeres futtatás esetén a felhasználó értesítésére szolgáló mezőbe a ’A „parancsnév”… futtatása’ szöveget írjuk.

Hasonlóan megváltozik a megfelelő gomb szövege, ha a kimeneti futtatható állomány nevét, illetve helyét szeretnénk megadni. Sikeres megadás esetén az új fájlnév kerül a gomb feliratára, ha üresen hagyjuk a mezőt, nem változik, ahogy a kész állomány elérési útja sem. Szintén nem történik változás, ha új fájlt szeretnénk megnyitni, azonban elfogadás nélkül bezárjuk a fájldialógus-ablakot, vagy nem létező fájlt szeretnénk megnyitni.

Fordítási (lexikális, szintaktikai, szemantikai) hiba esetén értesítjük a felhasználót. Pl.:

Végezetül futtassuk is a **source.input** alapján elkészült programot (a kimeneti állomány nevét az alapértelmezett **compiled_o**-n hagytuk):

Összefoglalás

Célunk egy olyan alkalmazás elkészítése volt, mely betekintést enged az adott eljárások, módszerek működésébe, szemlélteti, hogyan változik a kódunk az optimalizációk hatására. Véleményem szerint ezt sikerült elérnem, az elkészült nagyprogram teljesíti a fejlesztés kezdetén meghatározott kritériumokat. Az alkalmazás elkészítése során alapvetően a Fordítóprogramok tantárgy elvégzése során szerzett ismereteimet használtam fel, azokat továbbgondolva építettem fel a fordítóprogramot. Sokszor ez azonban nem volt elég a célom eléréséhez, kutatómunkára volt szükség – többek között például az optimalizációs eljárások mögötti gondolatvilág megismeréséhez és megértéséhez - így én is sok új ismeretre tettem szert a fejlesztés során.

Mivel a szakmai fejlődésemhez jelentősen hozzájárult a szakdolgozatom elkészítése, sikeresnek tekintem a folyamatot, mely a kész programhoz vezetett. Reményeim szerint a siker abban is megnyilvánul majd, hogy a téma iránt érdeklődő hallgatóknak segédeszközként szolgálhat majd az elkészült alkalmazás.

Hivatkozások

[1] X11 szabvány, https://en.wikipedia.org/wiki/X_Window_System_core_protocol (2016.03.11.)

[2] Lokális optimalizáció, Csörnyei Zoltán: Bevezetés a fordítóprogramok elméletébe 2. rész, ELTE, 2004, [250]

[3] Flex, http://flex.sourceforge.net/manual/ (2016.03.20.)

[4] Bison, https://www.gnu.org/software/bison/ (2016.04.02.)

[5] Dr. Hunyadvári László, Manhertz Tamás: Automaták és formális nyelvek, ELTE IK, 2006, [78]

[6] LALR elemző, https://en.wikipedia.org/wiki/LALR_parser (2016.04.05.)

[7] NASM, http://www.nasm.us/doc/nasmdoc0.html (2016.04.12.)

[8] Graphviz, http://www.graphviz.org/Documentation.php (2016.04.12.)

[9] C szintaxis, https://en.wikipedia.org/wiki/C_syntax (2016.04.14.)

[10] Járai Antal: Bevezetés a matematikába, ELTE Eötvös Kiadó, 2006, [242], ISBN-9789634637295

[11] Alapblokk, https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html (2016.04.23.)

[12] Qt, http://doc.qt.io/ (2016.04.23.)