Diplomamunka

Általános forráskód elemző kiegészítése a GCC fordítóprogram kódgenerálójával

Nagy Csaba
közgazdasági programozó matematikus hallgató

Témavezető: Dr. Beszédes Árpád, adjunktus

Szoftverfejlesztés Tanszék
Szegedi Tudományegyetem, Természettudományi Kar

Szeged, 2007

Tartalomjegyzék

Feladatkiírás

A Columbus forráskód elemző eszköz és keretrendszer nyelvi elemzője egy általános célú, könnyen újrafelhasználható komponens (ún. front-end), amely számos alkalmazásban bizonyított a szoftverkarbantartás területén. Az elemző feladata a forráskód átalakítása egy közbülső formára, amely kellőképpen általános, ezért minden információt tartalmaz, továbbá kitűnően alkalmas a további feldolgozásra. Emiatt fordítóprogram szintaktikus elemzőjeként is működhet, viszont ezidáig nem készültek hozzá megfelelő optimalizáló és kódgeneráló modulok.

A fordítóprogramként üzemelő Columbus elemzőnek több haszna is lehet. Először, mivel a szintaktikus elemző nagyfokú flexibilitással rendelkezik a kigyűjtött adatok tárolása és manipulálása terén, lehetőség nyílik különböző programtranszformátorok (pl. refaktoring eszközök) írására, amelyek közvetlenül képesek a tárgykódú programot előállítani a módosított programból. Erre eddig csak forráskód generálás révén volt lehetőség. Továbbá, a szintaktikus elemző pontosságának ellenőrzésére is kiválóan alkalmas egy ilyen házasítás, hiszen egy bizonyítottan jó kódgenerálóval előállított és helyesen futó program nagy megbízhatósággal jelenti a szintaktikus elemző helyességét is.

A diplomamunka keretében el kell végezni a Columbus C/C++ szintaktikus elemző illesztését a GCC fordítóprogram kódoptimalizáló és kódgeneráló részeihez. A Columbus részéről a szabványos Absztrakt Szintaxis Gráf programozói felületét kell használni, míg a GCC-ben meg kell találni a leginkább megfelelő csatlakozási pontot. A megvalósításnak fednie kell a C nyelvet azaz, eltekintve a front end esetleges hibáitól, C nyelvű programok elemzését sikeresen kell elvégezni.

Tartalmi összefoglaló

A feladatkiírásban szereplő feladat, azaz a Columbus/CAN *front end* alkalmazása és a GCC fordító program összekapcsolása, két szemszögből is megközelíthető. Az egyik szemszög szerint a *reverse engineering* forráselemzésekre tervezett Columbus keretrendszert ruházzuk fel egy compiler képességeivel és illesztjük hozzá a GCC fordítót. A másik szemszög szerint pedig a GCC fordító programot egészítjük ki egy új, jól kezelhető *reverse engineering front end* alkalmazással.

Diplomamunkámban a Columbus felőli megközelítést választottam, azaz olyan megoldási módszert mutatok be a felvetett problémára, amelyben a Columbus forráselemzési folyamatához hozzáillesztem a GCC fordítási folyamatának kódoptimalizáló és kódgeneráló fázisait.

Mivel mindkét alkalmazás igen összetett mind használatában, mind felépítésében, ezért a megoldási módszer megtervezéséhez nem elég pusztán a két program működését ismerni, hanem felépítésük szerkezetét is meg kell érteni. Az alkalmazott megoldási módszer bevezetéséhez, ezért diplomamunkámban a Columbus és a GCC ismertetése mellett olyan témákat járok körbe, mint a fordító programok működése és felépítése, vagy a C/C++ alkalmazások forráselemzése.

A Columbus és GCC házasításához egy olyan C/C++ alkalmazást készítettem, ami a Columbus és GCC köztes reprezentációs nyelvei közötti transzformáció segítségével képessé teszi a fordítót a CAN *schema instance* állományának a lefordítására. A transzformáció implementálásában a C és C++ nyelvi eszközök mellett olyan alkalmazott technikák is ötvöződnek, mint a *Visitor*, *Factory* vagy *Iterator* tervezési minták. A két összekapcsolandó alkalmazás bonyolultsága és a köztük lévő nagy különbségek eredményezték a kiterjesztés forrásának összetettségét, amit az szemléltet, hogy az implementálás végeztével egy 95 (.cpp és .h) fájlból álló, közel 7700 soros forrás keletkezett.

Az implementálás helyességének ellenőrzéséhez a kiterjesztett fordítóval valós alkalmazásként sikeresen fordítottam le Debian/GNU Linux rendszeren a *bzip2*-t, ami közel 8000 soros forrásával összetett nyelvtani elemekkel verifikálta az implementációt. Teszteltem a kiterjesztést továbbá több kisebb C és C++ forrásfájlon is, valamint a GCC hivatalos *Code-Size Benchmark Environment* (CSiBE) tesztrendszerének egyes részein is.

Diplomamunkám eredményeként azzal, hogy megoldást mutattam a feladatkiírásban kitűzött feladatra, egy széles körben használt forráskód elemző eszközt egészíthettem ki új és hasznos értékekkel. Ez a kiegészítés pedig további, újabb lehetőségek előtti kapukat tár fel.

Kulcsszavak - forráselemzés, Columbus, fordító programok, GCC, reverese engineering, front-end, optimalizálás

Bevezetés

Nagyobb méretű alkalmazások esetén – különösen ha kereskedelmi szoftverről van szó nagyon fontos –, hogy már teljesen leoptimalizált bináris kód kerüljön a felhasználók elé. Ezt fejlesztés közben elérni gyakran igen nehéz feladat, ugyanis a programozók általában nem tudnak minden optimalizálási lehetőségre külön odafigyelni. Gyakran az ismert lehetőségeket szándékosan sem egyszerűsítik, hogy megérthető maradjon a forrás. Nem is várhatnánk viszont el ezt tőlük másként, hiszen több ezer sorokat tartalmazó alkalmazásokat, főleg ha több programozó készítette, szinte lehetetlen átlátni. Léteznek ezért olyan alkalmazások, amelyek megpróbálják segíteni a fejlesztők dolgát. Segíthetnek a végső binárist egy adott szempont (pl. futási idő vagy kódméret) alapján optimalizálni, vagy segíthetnek rámutatni az optimalizálható forrásrészekre, az esetleges hibák forrásaira. Azok az alkalmazások, amik a végső binárist optimalizálják, általában maguk a fordító programok, utóbbiak pedig úgynevezett *reverse engineering* alkalmazások.

Diplomamunkámban egy forráselemző programcsomag (Columbus) és egy fordító program (GCC) összekapcsolásával a két különböző programcsalád között teremtek kapcsolatot. Az összekapcsolásnak köszönhetően egy olyan fordítói képeségekkel felruházott forráselemzőt kaphatunk, ami képes a fordítás előtt *reverse engineering* elemzések elvégzése mellett, közvetlenül a tárgyprogram előállítására is. Mivel a Columbus olyan saját reprezentációs formát épít fel a forráskódról, ami egészen magas szinten ad átfogó információt az egész projekt forrásáról (nem csak egy fordítási egységéről, mint a fordító programok általában), az így kapott fordítási folyamatba egészen speciális transzformációk elvégzésére is lehetőségünk adódik.

GCC fordítót nagyon sokáig kritizálták például azért, mert képtelen *interprocedurális elemzések* [25] elvégzésére. Ezek olyan eljárások, amelyek több függvény között vagy esetleg a forrásprogram több fordítási egységén átfogóan végeznek különböző műveleteket. A GCC fordítás közben azonban fordítási egységként a forrásprogram függvényeit tekinti, és azokat transzformálja alacsonyabb szintű nyelvekre. A Columbus és a GCC képességeit egyesítve olyan alkalmazást kaphatunk, ami az interprocedurális optimalizálásokat is képes fordítási folyamatában elvégezni, akár forrásfájlok között is, sok fordítót felülmúlva ebben.

A bemutatott kiterjesztéshez a GCC forrásába ágyazok bele egy olyan modult, ami a Columbus köztes reprezentációs kódját alakítja át a GCC legmagasabb szintű reprezentációs nyelvére (arra a nyelvre, ami alapján a GCC építi fel a saját Absztrakt Szintaxis Fáját). Ha ugyanis a Columbus forráselemzési fázisai után, ezt a transzformációt el tudjuk végezni, akkor egy olyan szerkezetű struktúrát tudunk felépíteni, amit a GCC parsere helyet átadhatunk a fordító parsert követő fázisainak. Így pontosan azt érjük el, hogy a Columbus *front end* részét a fordítási folyamat elejére illesztjük, és a két alkalmazást ezzel összekapcsoljuk egy fordítási folyamatba.

Diplomamunkám eredményét ennek a kiterjesztésnek az implementálása adja. Ebben a modulban ugyanis rengeteg tanult technikát kell alkalmazni azért, hogy kapcsolatot tudjunk teremteni a két igen összetett alkalmazásnak a C++ teljes nyelvtanát leíró köztes nyelvei között. Ezt jellemzi a modul forrásának mérete is ugyanis az implementálás végeztével egy 95 (.cpp és .h) fájlból álló, közel 7700 soros forrás keletkezett.

A projektet a jövőben több olyan irányba is tovább lehet fejleszteni, ami napjainkban a forráselemzők és fordítóprogramok világában is újdonságnak minősül. Az egyik, hogy a két a program összekapcsolásával kapott fordítási folyamatba belevigyük azokat az említett transzformációkat, amelyekre a Columbus a jól strukturált, modern, objektum orientált szemléletű köztes reprezentációja miatt alkalmas (ellenben sok fordító programmal). A másik pedig, hogy az alkalmazott technikák általánosításával egy általános modult készítsünk a GCC-hez, aminek a segítségével a Columbushoz hasonlóan bármilyen más *front end* alkalmazást is össze lehessen kapcsolni a fordítóval. Mivel a diplomamunkámban bemutatott kiterjesztés egyben első próbálkozásnak is minősül, amelyben a GCC-t új *front end*-del egészítik ki, ezért későbbiekben ennek az általánosításnak is az alapjául szolgálhat.

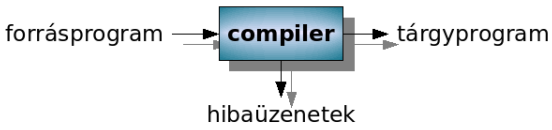
1. fejezet

A GCC fordító

Ebben a fejezetben egy mélyebb leírást adok a GCC-ről, hogy jobban megismerhessük a fordító programot, amely diplomamunkám témájának alapjául szolgál. Egy általános bevezető után röviden bemutatom az alkalmazás történetét, majd részletesebben jellemzem működését és felépítését. Utóbbi különösen fontos ahhoz, hogy később pontosan láthassuk, hol és hogyan illeszthetjük ezt a nagyon összetett fordítót egy új *front end* alkalmazáshoz, a Columbus keretrendszer CAN elemzőjéhez.

1.1. Bevezetés

A *compilerek* (*fordító programok*) olyan számítógépes programok, amelyek valamely számítógépes nyelven – a *forrásnyelven* – írt szöveget fordítanak le egy másik számítógépes nyelvre – a *tárgynyelvre* – (1.1. ábra) [1]. Tipikusan a fordító bemenete, a *forrásprogram*, egy (esetleg több) szöveges fájl, aminek tartalma valamely ismert, magasabb szintű programozási nyelven íródott. A kimenet, a *tárgyprogram*, pedig általában egy futtatható bináris, vagy egy *object file*, ami már úgynevezett *gépi kódot* tartalmaz. Az outputot a számítógépen futó operációs rendszer segítségével gyakran azonnal futtathatjuk is, vagy egy *linker* segítségével később is felhasználhatjuk több object például futtatható állománnyá való összefűzéséhez.



1.1. ábra. Egy fordító program.

A fordító programokat több szempont alapján lehet csoportosítani. Az egyik ilyen szempont, hogy milyen szintű köztes nyelvek között végzi a compiler a fordítást. Eszerint léteznek úgynevezett *decompilerek*, azaz „visszafordító programok” is, amelyek a *compilerek* ellentétjeként a gépi kódot próbálják visszafejteni és valamilyen magasabb szintű programozási nyelvre átalakítani. Illetve léteznek úgynevezett *nyelv fordítók* (*language translator*) is, amelyek egyik magasabb szintű nyelvből a másikba végeznek átalakítást.

Egy másik szempont a csoportosításhoz, hogy hány lépésben végzi a compiler a fordítást. Eszerint egy menetes (*single-pass*), illetve több menetes (*multi-pass*) fordítókat

különböztethetünk meg. Az egy menetes fordításnak az előnye lehet a gyorsaság és az egyszerű implementálás. Ma viszont már a modern fordítók általában az összetettségük miatt több menetre osztják a fordítási folyamatot. Több menetes fordító például a *GNU Compiler Collection* (GCC) [13] is.

A GCC, ahogy a teljes neve is mutatja, valójában nem egy fordító program, hanem többnek is a gyűjteménye, amelyekkel több programozási nyelvet rengeteg külön architektúrára fordíthatunk. A *GNU projekt* részeként a GCC-t is ingyenes szoftverként a *GNU General Public License* (GNU GPL) és *GNU Lesser General Public License* (GNU LGPL) licenszekkel adja közre a *Free Software Foundation* alapítvány.

1.2. A GCC története

A GCC fejlesztését 1985-ben kezdte Richard Stallman, aki egyben a *GNU projekt* megalapítója is volt 1983 szeptemberében [4]. Ennek a projektnek a keretében egy ingyenes *Unix-szerű* operációs rendszert, a *GNU Operációs Rendszer*-t akarták megvalósítani, hiszen a 80-as években, még minden operációs rendszerért igen nagy összegeket kértek a fejlesztők. Erre utal maga a GNU név is, ami egy frappáns, önmagára mutató, rekurzív rövidítése a „GNU's Not Unix", azaz a „GNU Nem Unix" kifejezésnek. A fejlesztők céljukat először 1992-ben, a „Linux" kernel megjelenésével érték el, amikorra a kernellel már minden fontos összetevő elkészült egy operációs rendszer kiadásához.

A rendszernek a részét képezte az először 1.0-s verzióként, 1987 májusában kiadott GCC is, ami akkor még a „GNU C Compiler" rövidítését jelentette. Stallman kezdetnek egy korábban megírt *Pastel* (a *Pascal* nyelv egy dialektusa) fordítót írt át C fordítóvá. Eredetileg pedig maga a fordító program is *Pastel* nyelven íródott, de még az első kiadás előtt Stallman, Len Tower kollégája segítségével teljesen átírta C nyelvre a forrást is [40]. Még ugyanabban az évben, 1987 decemberében kiadták a fordítónak azt a verzióját is, amiben már kezdetleges C++ kezelés is volt. Az igazi C++ támogatás viszont később, az 1992-ben kiadott 2.0-s verzióban jelent meg.

A következő nagy fordulópont a projekt életében 1997-ben volt, amikor a fejlesztők egy csapata „megelégelte", hogy lassan halad a GCC fejlesztése és elindította az *Experimental/Enhanced GNU Compiler System* (EGCS) projektet. Ennek a „*branch*"[1]-nek a célja az optimalizálási fázisoknak és a C++ támogatásnak a továbbfejlesztése volt, több GCC-ből kinőtt fejlesztői szál összefűzése mellett. Mivel az EGCS fejlesztése sokkal dinamikusabban haladt, mint a „*mainline*"[2] vonal, ezért később, 2001-ben az EGCS lényegében átvette a *mainline* GCC helyét, és összeolvasztva a két projektet, kiadták a GCC 2.95-ös verzióját [16]. Ebben a verzióban, már a „GNU Compiler Collection" nevet használták a projektnek utalva a támogatott programozási nyelvek sokaságára (1.1. táblázat).

A jelenleg legfrissebb, 2007. február 13-án kiadott, 4.1.2-es verzióban már világszerte több száz fejlesztő munkássága van benne, amit talán legjobban a fordító sokoldalúsága bizonyít. A legújabb verzióban már van C, C++, Objective-C, Objective-C++, Java, Fortran, és Ada programozási nyelv támogatás, valamint számos architektúrára fordíthatunk binárisokat, amelyek közül csak néhányat kiemelve létezik *Alpha*, *ARM*, *Blackfin*,

[1] A GCC fő fejlesztői vonaláról több fejlesztői vonal ágazódik le. Ezeket, a kisebb fejlesztői csapatok által karbantartott al-projekteket hívják *branch*-eknek.

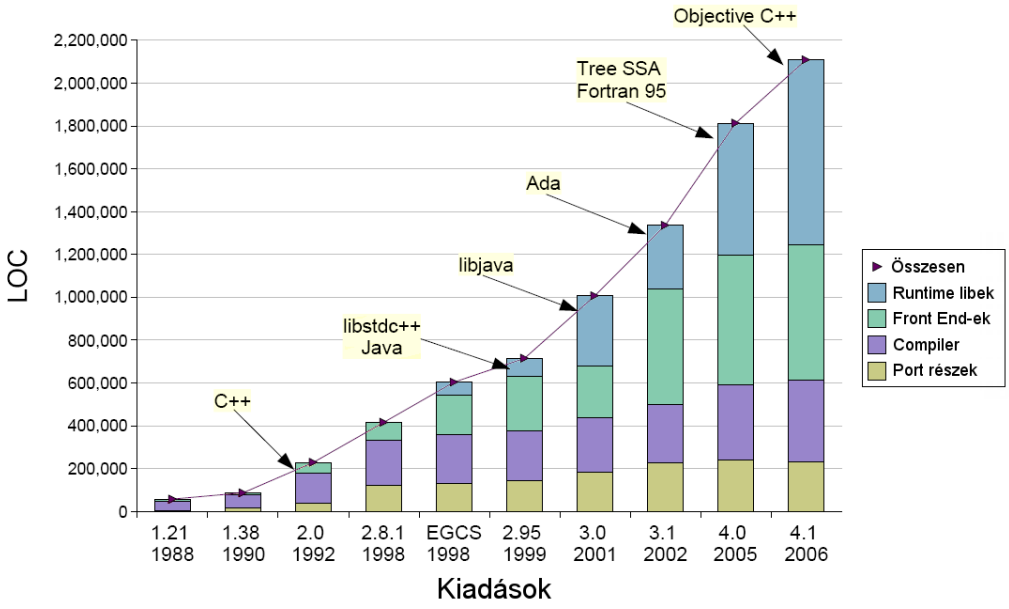
[2] A GCC aktuálisan fejlesztett verzióját hívják *mainline*, fő fejlesztői vonalnak.

Verzió	Megjelenés	Megjegyzés
0.9	1987. március 22.	az első béta kiadása a GCC-nek
1.0	1987. május 23.	az első stabil kiadás, C nyelv fordítására
1.15.3 (g++)	1987. december 18.	kezdetleges C++ front end
2.0	1992. február 22.	forrás revíziója, C++ támogatás
EGCS 1.0	1997. december 3.	az első EGCS verzió megjelenése
EGCS 1.1.2	1999. március 15.	utolsó EGCS verzió
GCC 2.95	1999. július 31.	„GNU Compiler Collection”
GCC 3.0	2001. június 18.	az EGCS beolvasztása a mainline GCC-be
GCC 4.0.0	2005. április 20.	4.0-s sorozat megjelenése
GCC 4.1.2	2007. február 13.	jelenlegi legújabb verzió

1.1. táblázat. Néhány fontosabb GCC verzió megjelenése.

i386, *IA-64*, *M68k*, *MIPS*, *SuperH*, *SPARC* processzorok támogatása [17].

A GCC-t, köszönhetően sokoldalúságának és dinamikus fejlődésének (1.2. ábra [27]), több operációs rendszer is hivatalos fordítójának választotta, ami azt jelenti, hogy az operációs rendszert magát is, és a benne található alkalmazásokat is GCC-vel fordítják és adják ki. Ilyen rendszerek például a különböző „Linux” disztribúciók, amelyeket gyakran „GNU/Linux”rendszereknek is hívnak (pl.: *Debian GNU/Linux*). A GCC a hivatalos fordítója még a BSD rendszereknek is (*FreeBSD*, *OpenBSD*, *NetBSD*, stb.), az *Apple Inc.* által fejlesztett *Mac OS X*-nek, a *BeOS*-nek és az úgynevezett *smartphone*-okról vagy *palm top*-okról közismert *Symbian OS*-nek is.



1.2. ábra. A GCC dinamikus fejlődése a fontosabb verziószámokkal.

1.3. A GCC használata

A GCC több, ingyenes operációs rendszernek is a hivatalos fordítója, ezért ezekre a rendszerekre általában külön telepíteni már nem kell. Az egyes *Linux* disztribúciókon például a disztribútorok (a rendszer fejlesztői) által kiadott csomagok segítségével könnyen elérhető és használható a fordító. A *Minimal GNU for Windows* (MinGW) [32], valamint a Cygwin [37] keretrendszereknek köszönhetően pedig a GNU alapvető programcsomagjaival együtt a GCC is használható Windows rendszereken.

Mivel maga a GCC projekt több programozási nyelvnek a fordítóját foglalja magába, a fejlesztők a külön kisebb fordítókat önálló projektekként külön nevekkel látták el. A *G++* elnevezést kapta a C++, a *GCJ* elnevezést a Java, *GNAT* nevet az Ada és a *GFORTRAN*-t pedig a Fortran fordító. A diplomamunkám szempontjából a C++ és a C nyelvek kapnak kiemelt szerepet, ezért továbbiakban az ezekhez kapcsolódó alkalmazások bemutatásával foglalkozok.

Tekintsük most minta programunknak a klasszikus „Hello World!” program [30] C illetve C++ változatát, a `hello.c` és `hello.cpp` forrásfájlokat (1.3. ábra)! Attól függően, hogy melyik nyelven írt forrást szeretnénk lefordítani, fordításukhoz a `gcc` illetve `g++` programokat használhatjuk az 1.4. ábrán szemléltetett módon.

```c
#include <stdio.h>

int main() {
  printf("Hello World! (C)\n");

  return 0;
}
```

(a) `hello.c`

```cpp
#include <iostream>

using namespace std;

int main() {
  cout << "Hello World! (C++)" << endl;

  return 0;
}
```

(b) `hello.cpp`

1.3. ábra. „Hello World!” mintaforrások.

A `gcc` és `g++` fordítók egyszerűen parancssorból futtathatók, a fordítandó forrásfájlok illetve a kimeneti állomány neve pedig parancssori argumentumként adhatóak meg. A fordítás menete és a működés természetesen rengeteg kapcsolóval módosítható, amelyek közül csupán nehányat (a diplomamunkában alkalmazott módszerekhez fontosabbakat) emel ki a 1.2. táblázat [17].

```
$ gcc -o helloc hello.c
$ g++ -o hellocpp hello.cpp
$ ./helloc
  Hello World! (C)
$ ./hellocpp
Hello World! (C++)
```

1.4. ábra. „Hello World!” programok fordítása és futtatása.

Kapcsoló	Jellemző
-c	a fordítás utolsó fázisában a linkelés kimarad, így a kimeneti állomány egy object fájl lesz (`.o`)
-S	a fordítás az assembler előtt megáll, a kimeneti állomány az assembly forrás lesz (`.s`)
-E	a fordítás az preprocesszor után megáll, a kimeneti állomány egy proprocesszált `.i` fájl
-o <file>	a kimeneti állomány nevének meghatározása
-I <könyvtár>	könyvtár hozzáadása a header fájlok keresési útjaihoz
-L <könyvtár>	könyvtár hozzáadása a library fájlok keresési útjaihoz
-v	„verbouse mód”, fordítás részleteinek kiírása
-Wall	minden fordítási figyelmeztetés bekapcsolása
-Werror	a fordítási figyelmeztetéseket fordítási hibákként kezelje a fordító
-O0	optimalizálás nélküli fordítás
-O1, -O2, -O3	optimalizálási algoritmusok bekapcsolása (minél nagyobb a szám, annál több, hatékonyabb algoritmust futtat a fordító (a fordítási idő rovására))
-Os	méretre optimalizálás
-fdump-tree-all	a *Tree* optimalizálási fázisok dump adatainak kiíratása
-fdump-rtl-all	RTL optimalizálási fázisok dump adatainak kiíratása

1.2. táblázat. Fontosabb *gcc* és *g++* futtatási paraméterek.

1.4. A fordítási folyamat

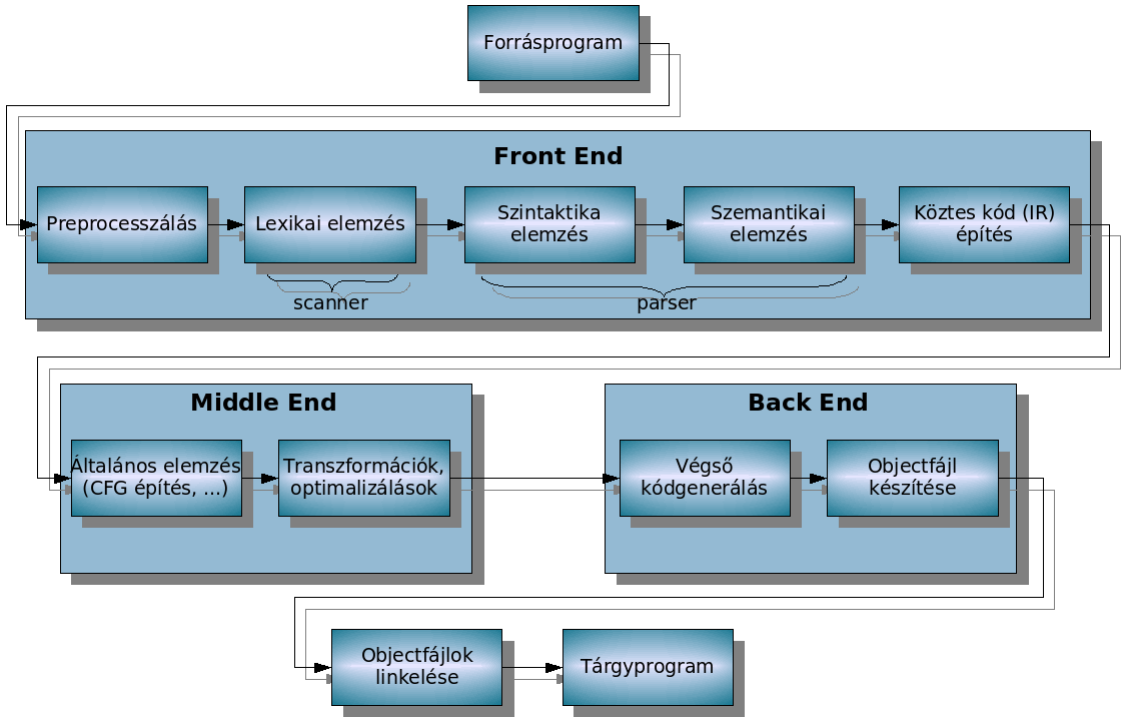
A compilerek fordítási lépéseit általában az *Analízis-Szintézis Modellel* szokás jellemezni [1], ami meghatározza egyben a fordító programok struktúráját is. Eszerint a modell szerint a fordítás menete két fő részre osztható: az „elemzésre” (analízis) és a „felépítésre” (szintézis). Az elemzési fázisban olvassa be a compiler a *forrásprogramot*, ellenőrzi annak helyességét és építi fel egy úgynevezett köztes reprezentációját a forrásnak. Az így felépített reprezentáció alapján a következő fázisban, a felépítési fázisban készíti el a fordító program a *tárgyprogramot*. Az elemzésért a *front end*, a felépítésért pedig a *back end* a felelős.

A modern fordítók, mint a GCC is, ebbe a két lépéses modellbe egy harmadik, köztes lépést is beiktatnak, ami az optimalizálási lépésekért, és köztes reprezentációs nyelven végzett transzformációkért a felelős. Ezt a részt a *middle end*-nek hívják. (1.5. ábra)

1.4.1. Front end

A *front end* tehát a fordító programnak az a része, ami az *Analízis-Szintézis Modellben* definiált elemzési részért felelős. Ebben a szakaszban a fordító az alábbi főbb lépéseket végzi el:

1. preprocesszálás
2. elemzés



1.5. ábra. A fordítás lépései.

3. a köztes kód építése

Még az elemzéseket megelőzően egy fontos feladata a modern fordítóknak a bemeneti állomány *preprocesszálása* is, aminek során a fordító előkészíti a forrást a *parser* (az elemző) számára. A preprocesszálás alatt végzendő feladat lehet például a C nyelvben a makrók behelyettesítése.

Az elemzési fázisokat követően pedig az utolsó és igen fontos feladata a *front end*-nek, az úgynevezett *közbenső kód* (*intermediate representation*, IR[3]) generálása. Ez egy olyan adatstruktúra, ami a fordító számára könnyen kezelhető formában minden fontos adatot eltárol a forrásprogramról a későbbi fázisok futásához.

Maga a fő feladat, az elemzés, pedig további 3 fázisra tagolható:

1. lineáris vagy lexikai elemzés
2. hierarchikus vagy szintaktikai elemzés
3. szemantikai elemzés

A következő alfejezetekben ezeket a fázisokat ismertetem.

Lexikai elemzés

A *lexikai elemzés* (más néven *lineáris elemzés*) során a bemeneti állományt „balról jobbra” karakterenként olvassa a fordító és csoportosítja egységekre, amelyeket *jelkép*eknek

[3] A szakirodalmakban az *intermediate representation* (IR), *intermediate representation language* (IRL) és *intermediate language* (IL) kifejezések keverednek. Én továbbiakban az IR kifejezést használom, amikor a köztes kódra utalok, illetve az IL kifejezést ha ennek a nyelvezetére.

(angolból átvéve *token*-eknek hívnak). A

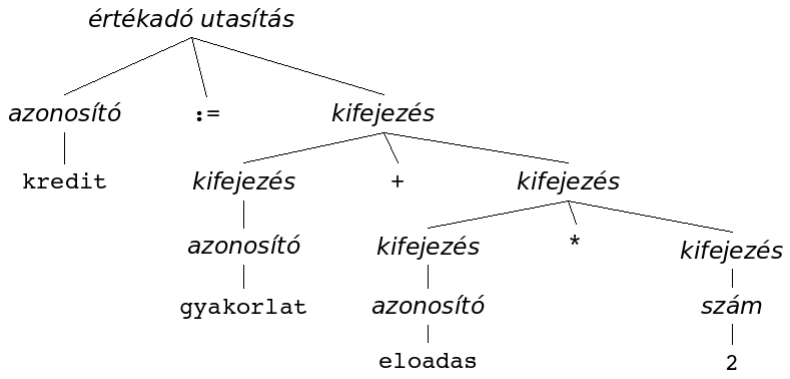
$$kredit := gyakorlat + eloadas * 2$$

kifejezés például az alábbi *token*ekre tagolható:

1. a $kredit$ azonosító
2. az értékadás $:=$ szimbóluma
3. a $gyakorlat$ azonosító
4. a $+$ összeadás jel
5. az $eloadas$ azonosító
6. a $*$ szorzás jele
7. a 2 szám

Szintaktikai elemzés

A *szintaktikai elemzés* (más néven *hierarchikus elemzés*) során a korábbi *lineáris elemzés* alatt keletkezett tokenek lineáris szekvenciáját csoportosítja a fordító az adott forrásnyelv által értelmezhető, összefüggő hierarchikus csoportokba, mondatokba. Ezt a csoportosítást gyakran egy fa struktúra – a *parse tree* – felépítésével hajtják végre (1.6. ábra).



1.6. ábra. A $kredit := gyakorlat + eloadas * 2$ kifejezés elemzőfája.

Szemantikai elemzés

A *szemantika elemzés* során a fordító a korábban felépített elemzőfát látja el a későbbi kódgeneráláshoz szükséges tartalommal, miközben szemantikai ellenőrzést végez a *forrásprogramon*. Az ellenőrzés közben az adott nyelvtől függően egyéb fontos feladatokat is elláthat a *front end*, mint például egy *szimbólum tábla* (a forrásprogramban előforduló azonosítókat tároló adatstruktúra) felépítése, avagy az operátorok típusellenőrzései.

1.4.2. Middle end

A *middle end* szerepe gyakran összemosódik a *back end* szerepével. Egyes irodalmak [1] ezért nem is tárgyalják különvéve, csak a *back end* egységén belül. Az újabb irodalmakban [36, 33] azonban egyre inkább külön egységnek veszik, hiszen a modern fordítóknál egyre nagyobb és egyre fontosabb szerepet kapnak azok a transzformációk, amikkel méretre, futási időre, vagy akár felhasznált energiára is optimalizálhatunk. Az ilyen transzformációk rendszerint nyelvtől független transzformációk, amelyeket a *front end* által létrehozott köztes kódon hajt végre a fordító.

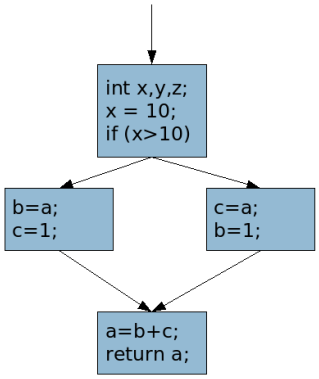
A *middle end* feladatai a következők:

1. általános elemzés (CFG, DFG építés)

2. optimalizálás

Az optimalizálási folyamatokhoz gyakran, a *front end* által, elemzés közben összegyűjtött információ nem elég. Ahhoz, hogy egy kódot jól optimalizálhassunk, a forrás teljes *irányítás folyamatát* (*control flow*), illetve teljes *adatfolyamát* (*data flow*) ismerni kell. Ezeket a folyamokat irányított gráfokkal (*Control Flow Graph* (CFG), *Data Flow Graph* (DFG)) szokták jellemezni.

A CFG csúcsai a *basic block*[4]-ok, élei pedig azt írják le, hogyan léphet át egyik *basic block*-ból a másikba a program (1.7. ábra). A DFG hasonló szerkezettel az egyes utasítások függőségét az adatokon végzett változtatások szempontjából reprezentálja.



1.7. ábra. *Control Flow Graph* (CFG) példa.

Néhány alapvető optimalizálási eljárás, amit a fordító programok általában használnak:

Algebrai egyszerűsítések. Az algebrai műveletek sok kifejezésnél általában könnyen egyszerűsíthetőek. Makrók használatánál keletkezhetnek például olyan kifejezések, mint a $i-1+i$ ($\Rightarrow -1$) kifejezés. Persze ennél bonyolultabb egyszerűsítéseket is végezhet a fordító az asszociativitási, kommutativitási és disztributivitási szabályokat felhasználva.

[4] A *basic block* utasítások összefüggő szekvenciája, pontosan egy bemenettel és kimenettel.

Konstans műveletek. A konstansokat tartalmazó kifejezések gyakran tartalmaznak előre kiszámolható értékeket, amelyeket érdemes fordítási időben előre kiszámolni ($4 + + 3 - 2 * 1 \Rightarrow 5$).

Redundanciák kiküszöbölése. Az ismétlődő utasítások kiküszöbölésére több, egészen hatékony eljárást is alkalmazhatnak a fordítók, mint a „ciklusinvariáns kódmozgatás" (*Loop-invariant code motion*), „közös részkifejezések kiküszöbölése" (*Common sub-expression elimination*) vagy a „részleges redundanciák kiküszöbölése" (*Partial redundancy elimination*) [33].

1.4.3. Back end

Az utolsó fázisa a fordításnak a *végső kódgenerálás*, ami során a *middle end* által módosított IR-ből a végső gépi kód, vagy *assembly* kód készül. Ebben a fázisban már mindent tudnia kell a fordítónak a tárgyprogram nyelvének felépítéséről és a tárgyprogramot futtató rendszerről is (milyen utasításokat ismer, milyen regisztereket használ, stb.), hiszen különböző gépi kódot kell generálni a különböző architektúrákhoz is.

A *back end* feladata tehát, hogy az IR-t előkészítse, majd elvégezze a végső kód generálását. Ez egy igen nehéz és összetett feladat, amit a diplomamunkámban csak érintőlegesen tárgyalok, hiszen diplomamunkámban a GCC-t egy új *front end* alkalmazáshoz illesztem, amihez a fordítónak is a *front end* részeit kell módosítanom, a *back end* részt pedig érintetlenül hagyom. Egy-két érdekesebb feladatot, azonban mégis szeretnék kiemelni:

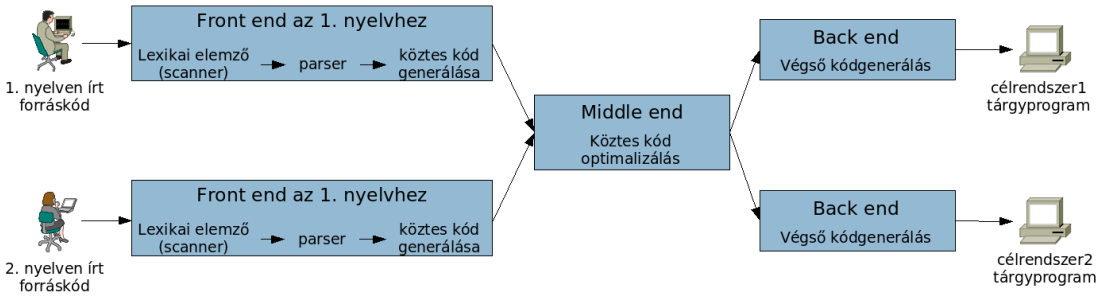
Regiszter allokáció. (*Register Allocation*.) Egyik fontos feladata a fordítónak, hogy a célrendszer által használt regisztereket minél jobban és minél hatékonyabban kihasználja. Ehhez pedig olyan gépi kódot kell generálni, ami maximalizálja azoknak a változóknak a mennyiségét, amelyeket a hardware regisztereibe érdemes betölteni a memóriában való tárolás helyett (a regiszterek elérése ugyanis sokkal gyorsabb).

Kód ütemezés. (*Code Scheduling*.) A modern processzorok is ütemezik a végrehajtandó utasításokat, hogy minél gyorsabban tudjanak futtatni egy-egy kódrészletet. Különösen így van ez a több szálat használó processzoroknál. Hasonló ütemezést érdemes már fordításkor is elvégezni, hogy minél kevésbé terheljük futtatáskor a CPU-t.

A *végső kódgenerálás* után még rendszerint nem egyből futtatható kódot kapunk. Az így kapott gépi kódot még, a célrendszer követelményeinek megfelelően, *object* állományokba ágyazza a fordító, majd általában egy külső *linker*-t meghívva a futtatáshoz szükséges object fájlokat összefűzi. Futtatható bináris állományt végül csak ezek után kapunk.

1.5. A GCC felépítése

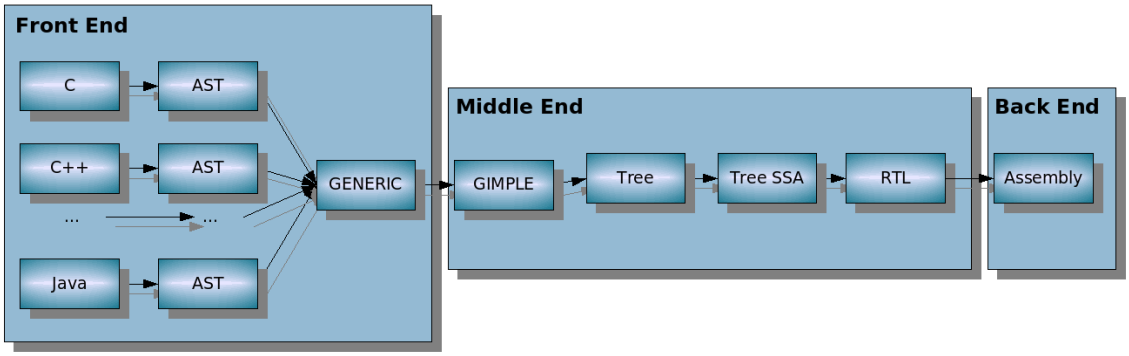
A 1.4. alfejezetben tárgyalt 3-as tagoltságot a GCC is követi, azaz itt is elkülöníthetjük a *front end*, *middle end* és *back end* részeket. Fontos lényegi különbség azonban az, hogy a GCC-ben a több nyelv és architektúra támogatása miatt több *front end* és *back end* részt is meg kell különböztetnünk. Ezek mindegyike önálló *front end*-ként, illetve *back end*-ként



1.8. ábra. Egy fordító program felépítése.

működik, és attól függően hívódnak meg, hogy milyen forrásnyelvű illetve tárgynyelvű programot fordítunk. (1.8. ábra)

Egy másik fontos tulajdonsága a GCC-nek, hogy fordítás közben több köztes nyelvvel dolgozik. A *front end* tehát felépíti az első IR-t, amit a úgynevezett *Abstract Szintaxis Fának* hívnak, majd a *middle end* számára átalakítja egy nyelvfüggetlen formára a GENERIC formára. Ezt azért hívják nyelvfüggetlen formának, mert amíg az szintaxis fa tartalmaz a forrásnyelvtől függő elemeket, addig GENERIC formában már bármilyen nyelvet ábrázolhatunk. A GENERIC az optimalizálási fázisokhoz tovább egyszerűsödik GIMPLE formára, majd később általános *Tree*, illetve a *Static Single Assignment* (SSA) [5] szabályait követő *Tree SSA* alakra [34, 35]. A *Tree-SSA* alak még egy aránylag magas szintű IL, amin nagyon jól tudnak dolgozni az egyes optimalizáló algoritmusok, de a *back end* számára még további egyszerűsítéseket igényel, hogy a köztes nyelvből a végső kód generálás könnyen elvégezhető legyen. Ezért a *Tree-SSA* alak tovább egyszerűsödik egy *Register Transfer Language* (Regiszterárviteli Nyelv, RTL) nyelvre, ami már nagyon alacsony szintű nyelv (gyakorlatilag az assemblynek egy architektúrától független változata) (1.9. ábra) [14, 15].



1.9. ábra. A GCC köztes nyelveinek sorrendje.

1.5.1. AST

Az *Abstract Syntax Tree* (Absztrakt Szintaxis Fa, AST) az első köztes alak a GCC-ben. Ezt a formát a *front end* már a *parser* futása közben építi, hogy az adott input forrásról a *middle end* számára minden fontos adatot eltároljon. Mivel az ezt leíró IL a későbbiekhez

képest egy olyan magas szintű nyelv, ami még a nyelvtani elemeket is tartalmazza, ezért minden különböző fordítási nyelvhez tartozó *front end* különböző szerkezetű, saját AST-vel dolgozik. Diplomamunkám fő témája Columbus keretrendszer kiterjesztése a GCC-vel, a Columbus pedig C és C++ forráskódok elemzésére íródott. A továbbiakban ezért valahányszor AST-re hivatkozok, az alatt a GCC C++ AST nyelvét értem.

Az AST felépítését a GCC szintaktikai analízis közben felépített elemzőfa (*parse tree*) egyszerűsítésével végzi. A *parse tree* szerkezete ugyanis egyértelműen ábrázolja a forrást, és így akár alkalmas is lenne a későbbi optimalizálások elvégzésére is, ha nem tartalmazna nyelvtani elemeket. A GCC ezért gyakorlatilag ugyanazt az adatstruktúrát használja mindkét fa ábrázolására. Sőt később látni fogjuk, hogy a GENERIC, GIMPLE (*Tree*) és *Tree SSA* formák is ugyanezt a struktúrát használják. Ez a `tree` struktúra, amit frappánsan a következőként jellemez egy GCC fejlesztő: „*tree' is the root of all evil. Imagine what the root of 'tree' is!*" [26].

A `tree` által leírt fáknak a csúcsait (és így az AST csúcsait is), egyszerűen *node*[5]-oknak szokás hívni, amelyek az egyes utasítások, kifejezések különböző nyelvtani elemeit jellemzik. A csúcsokat összekötő élek pedig ezeknek a nyelvtani elemeknek a hierarchiáját írják le, mint a korábban (1.4.1. fejezetben) tárgyalt *parse tree*.

A `tree`, mint típus, a GCC-ben egy struktúrára mutató pointerként van definiálva, az általános jellege miatt rengeteg különböző adattaggal, amelyeket általában makró hívásokon keresztül érhetünk el. A struktúra összetettségét jellemzi, hogy magát az adattípust és a hozzá tartozó legalapvetőbb makrókat definiáló `tree.h` fájl több, mint 4600 soros állománya a jelenlegi *mainline* GCC forrásának. Ennek a struktúrának a teljes leírására ezért nem is törekszem diplomamunkámban, csupán néhány fontosabb jellemzőjét emelem ki, amelyeket általában más források is jellemeznek [14, 15, 26, 27, 36].

A legfontosabb adattag, amit a `tree` definiál, az adott elem típusát leíró tag, amit a TREE_CODE makró segítségével érhetünk el. Emellett léteznek makrók az egyes utasítástípusok argumentumainak lekérdezésére, mint például egy összeadás első és második operandusának lekérdezésére a TREE_OPERAND makró. Illetve általában jellemző, hogy minden *node* típusnak a jellemző attribútumait valamilyen egyedi, csak az adott típusú elemeknél használandó makrókon keresztül érhetünk el. Ilyen például az *identifier* (azonosító, pl. változónév) típusoknál az azonosítóhoz tartozó név hosszát (IDENTIFIER_LENGTH), vagy magát a nevet (IDENTIFIER_POINTER) elérő makrók.

A kölönböző *node* típusok közül néhány fontosabbat emeltem ki a függelék 5. táblázatában. A táblázat első oszlopa az adott típus azonosítója (amit a TREE_CODE makró ad vissza lekérdezésnél); a második oszlopa a *node*-hoz tartozó argumentumok számát mutatja; a harmadikban pedig egy rövid leírással szemléltetem milyen funkciója van az adott típusnak.

„n alatt a k" AST példája

A GCC köztes nyelveinek szemléltetéséhez Dr. Dévényi Károly Tanár Úr „Programozás alapjai" című kurzuson tanított „`nalattk.c`" (1.10. ábra) példaprogramját választottam. Ez a program adott n és k pozitív egész számokhoz kiszámolja „n alatt a k" értékét a $\binom{n}{k} = \frac{n\cdot(n-1)\cdot\ldots\cdot(n-k+1)}{k!}$ képlet segítségével. Kis módosításként annyit változtattam rajta, hogy n és k kezdeti értékét a program ne a *standard input*-ról várja, hanem a két válto-

[5] node - az angolból átvett csúcs, csomópont

```
#include <stdio.h>

int main()
{
  int n, k, nak;
  int i;

  n=90;
  k=5;

  if (n < k || k < 0) {
    printf("%d alatt %d nem értelmezett!\n", n, k);
  } else {

    nak = 1;

    for (i = 1; i <= k; i++)
       nak = nak * (n - i + 1) / i;
    printf("%d alatt %d = %d\n", n, k, nak);
  }

  return 0;
}
```

1.10. ábra. Az „n alatt a k” példaprogram forrása.

zó inicializálásakor lehessen megadni. Ezzel egy kicsit egyszerűsítettem a forráson, de a módosítás a program szerkezetén lényegében nem változtat, az ábrák viszont jelentősen leegyszerűsödnek. Ezeket az ábrákat terjedelmük miatt viszont még, leegyszerűsített alakban is a függelékben praktikusabb elhelyezni, ezért későbbiekben az „n alatt a k” példaforrásokra vonatkozó ábrahivatkozásokhoz az ábrák mind a függelékben keresendők. (A példák szempontjából fontosabb részeket néha a szövegbe is beemeltem az egyszerűbb olvashatóság miatt.)

Maga a forrás egyszerű, ezért jól szemlélteti az egyes köztes nyelvek szerkezetét. Ugyanakkor elég összetett ahhoz, hogy megfigyelhessük rajta hogyan épülnek fel az alapvető vezérlési szerkezeteket, mint a feltételes, vagy az ismétléses vezérlés.

Az „`nalattk.c`” program AST alakját a függelékben a 5.1. ábra szemlélteti. Ezt az alakot a GCC-ben a *parser* kis módosításának segítségével írattam ki, fordítás közben. A módosítás az volt, hogy még az AST alacsonyabb szintre (GENERIC szintre) való transzformálásáért felelős *genericizer* meghívása elé beillesztettem egy saját kódrészletet, amivel a „main” függvény törzsét egy úgynevezett `debug_tree` eljárással kiírattam.

Az ábra szépen szemlélteti, hogyan ágazik el az egyes csomópontoknál az „absztrakt szintaxis fa”, illetve mely C++ utasításoknak milyen AST *node*-ok felelnek meg (*var_decl* a változódeklaráció, *if_stmt* az `if` utasítás, *for_stmt* a `for` utasítás, *call_expr* a függvényhívás, *return_expr* a `return` utasítás, stb.).

Az AST utasításai közül külön érdemes kiemelni a *cleanup_point_expr node*-okat, amelyek az AST építés sajátos pontjai. Ezek ugyanis azt jelzik a fordító számára, hogy az ilyen *node* alatti utasítások a későbbi fázisok során további vizsgálatot igényelnek. A példában ilyen utasítások az értékadó utasítások, illetve a `printf` utasítások, mert ezeknél az operandusok típuskonverzióját későbbi fázisban hajtja végre a fordító.

1.5.2. GENERIC, GIMPLE

Az AST-t még a *front end* GENERIC alakra transzformálja [14, 15]. Ez az alak egy nyelvfüggetlen köztes reprezentációs nyelv (IRL), ami a GCC esetében azt jelenti, hogy minden – a fordító által támogatott – forrásnyelv GENERIC alakra hozható. Ezt a transzformációt a *genericizer* végzi, amit *front end* minden függvény fordításakor az AST építésének a végén hív meg. A C és C++ forrásnyelvek esetében ez a transzformáció egészen könnyen végrehajtható ugyanis a GENERIC nyelvet a C/C++ AST-ből általánosították a fejlesztők.

Ugyan maga a GENERIC forma is alkalmas lenne egyes optimalizálások elvégzésére, ezt az alakot még az optimalizálási fázisok előtt egy egyszerűbb alakra, a GIMPLE alakra hozza a „*gimplifier*" [31]. Ez az eljárás még ugyan a *front end* részét képezi, de a GIMPLE alakban lévő fákat már a *middle end* fázisai használják. A GIMPLE alak a GENERIC egy egyszerűsített alakja, amit a McGill Egyetemen fejlesztett McCAT fordító SIMPLE [22] alakja alapján terveztek a GCC fejlesztői.

A leglényegesebb egyszerűsítés, amit a *gimplifier* a GENERIC nyelven végez, hogy minden kifejezést úgynevezett *3-címes* alakra hoz temporális változók segítségével. Ennek az alaknak a lényege, hogy minden utasítás legfeljebb 3 operandusú opA, B, C alakban kerül felírásra, ahol $AopB \longrightarrow C$. Ez az egyszerűsítés főleg az algebrai műveleteknél (mint például az értékadó utasítások vagy az egyenlőségek) jelentős, ezeket a kifejezéseket ugyanis a *3-címes* alaknak köszönhetően könnyebben és hatékonyabban tudják elemezni az optimizerek.

A másik fontos egyszerűsítés a GENERIC-hez képest a GIMPLE alakban a `goto` utasítások használata. Minden magasabb szintű utasításban ugyanis az ugrások `goto` utasításra egyszerűsödnek, ami a ciklusok és az `if` illetve `switch` szerkezetek alacsonyabb szintre egyszerűsödését jelenti a C/C++ nyelveken.

„n alatt a k" GIMPLE példája

GIMPLE példának a korábbi `nalattk.c` program fordítása közben készült „dumpfile"-t használtam, hogy láthassuk az eredeti forrás (1.10. ábra), az AST alak (5.1. ábra) és a GIMPLE (5.2. ábra) közötti különbségeket. A „dumpfile" elkészítéséhez a forrást a következő utasítással fordítottam:

```
gcc -c -fdump-tree-simple nalattk.c
```

A GENERIC formáról külön példát pedig azért nem készítettem, mert mint korábban említettem ez a C/C++ nyelvek esetében gyakorlatilag megegyezik az AST formával. Ennek a formának a fa szerkezetét pedig az előző alfejezetben már bemutattam.

A GIMPLE alakot a függelékben a 5.2. ábrán láthatjuk. Érdemes megfigyelni például, hogy az eredeti

```
nak = nak * (n - i + 1) / i;
```

utasítást *3-címes* alakban, hogyan egyszerűsíti le a fordító:

```
D.1777 = n - i;
D.1778 = D.1777 + 1;
D.1779 = D.1778 * nak;
nak = D.1779 / i;
```

A példán megfigyelhetjük a ciklusok `goto` utasításokra történő egyszerűsítését is. Az eredeti

```
for (i = 1; i <= k; i++)
    nak = nak * (n - i + 1) / i;
```

ciklusból az egyszerűsítés során a következő utasítássorozat lett:

```
<D1771>:;
D.1777 = n - i;
D.1778 = D.1777 + 1;
D.1779 = D.1778 * nak;
nak = D.1779 / i;
i = i + 1;
<D1772>:;
if (i <= k)
  {
    goto <D1771>;
  }
else
  {
    goto <D1773>;
  }
<D1773>:;
```

1.5.3. Tree, Tree-SSA

A GIMPLE *Tree* formája egy viszonylag újnak mondható, a 2003-as *GCC Developers' Summit* konferencián bemutatott struktúra [31]. Mielőtt beépítették volna a fordítóba, a GCC a függvényeket egyből az igen alacsony szintű RTL formába fordította, és a *middle end* ezen a struktúrán futtatta az optimalizáló algoritmusokat. Az új *Tree* struktúra bevezetésével lehetőség nyílt a magasabb szinten futtatható optimalizálási algoritmusok alkalmazására, és megjelenhettek egészen új fázisok is, amelyeket eddig egyaltalán nem támogatott a fordító. Ilyen új fázis például az *interprocedurális elemzés* megjelenése a GCC-ben, ami lehetővé teszi a függvények közötti transzformációk elvégzését.

A magasabb szinten dolgozó optimalizálási algoritmusok, mint a korábban említett redundanciákat kiküszöbölő algoritmusok is, általában a *control flow* módosításával dolgoznak, amiről önmagában a *Tree* struktúra nem tartalmaz elegendő információt. Ezért a *Tree* struktúrával együtt egy olyan köztes nyelvet is bevezettek, amit a *Static Single Assignment* (SSA) [5] szabványhoz igazítottak. Ezt a szabványt pedig a fordító programok széles körben használják hatékony CFG optimalizálásokhoz. Az új struktúrát *Tree SSA*-nak nevezték el és először 2003-ban, majd az implementációt 2004-ben a *GCC Developers' Summit* konferencián mutatták be [34, 35].

Az SSA struktúra alapja, hogy minden változó csak egy helyen kaphat értéket a programban. Ha a változóhoz többször is új értéket akarunk hozzárendelni, akkor a változónak egy új verzióját hozzuk létre. A fordító az SSA-ra való transzformálásnál az értékadó műveleteket megvizsgálja és minden változóhoz nyomon követi annak a verziószámát (az új verziót alsó indexként a_x alakban jelölik). (1.11. ábra)

A *control flow* elágazásainál előfordulhat, hogy egy változó használatánál nem egyértelmű fordítási időben (majd csak futási időben válik egyértelművé), hogy melyik korábbi

```
a = foo ();
b = a + 10;
c = 5;
T1 = b + c;
if (a > T1)
  {
    T2 = b / a;
    T3 = b * a;
    c = T2 + T3;
    b = b + 1;
  }
bar (a, b, c);
```

(a) Eredeti GIMPLE forma.

```
a_1 = foo ();
b_1 = a_1 + 10;
c_1 = 5;
T1_1 = b_1 + c_1;
if (a_1 > T1_1)
  {
    T2_1 = b_1 / a_1;
    T3_1 = b_1 * a_1;
    c_2 = T2_1 + T3_1;
    b_2 = b_1 + 1;
  }
b_3 = phi(b_1, b_2);
c_3 = phi(c_1, c_2);
bar (a_1, b_3, c_3);
```

(b) Ugyanaz a program SSA formában.

1.11. ábra. Static Single Assignment forma.

verziót kell éppen behelyettesíteni. Ilyen esetekben a fordító úgynevezett ϕ (phi) *node*-okat hoz létre. A 1.11. ábrán ilyen ϕ *node* jön létre a `bar()` függvényhívás előtt a b és a c változókhoz. A $b_3 = \phi(b_1, b_2)$ jelentése, hogy b_3 változó a b_1 és b_2 értékét is felveheti, úgymond „összefűzi" a korábbi verzióit a változónak.

Az SSA egy hiányossága, hogy a verziókövetés módszere alkalmazhatatlan az összetett adattípusokra, a tömbökre és a pointerekre. Gondoljunk csak bele, hogy hogyan követhetné egy `M[100][100]`-as tömb 10000 verzióját egy fordító, vagy hogyan döntené el egy `M[i][j]` és `M[k][l]` referenciáról hogy megegyeznek-e. Ezeknek a hibáknak a kiküszöbölésére úgynevezett *valós* és *virtuális operandusokat* vezettek be, amelyek az olyan utasításoknál, ahol összetett adattípust használ a fordító, segítenek feloldani az említett problémákat.

„n alatt a k" SSA példája

A korábbi `nalattk.c` példát SSA alakba is átalakítottam, hogy szemléltessem a GIMPLE *Tree* és a *Tree SSA* közötti különbségeket. A példa elkészítéséhez a forrást a következő utasítással fordítottam:

```
gcc -c -Os -fdump-tree-ssa nalattk.c
```

Itt a `-Os` kapcsolóra azért van külön szükség, mert az optimalizálási fázisokat kikapcsolva a fordító a teljes SSA fázist kihagyja és így SSA alakba sem transzformálja a forrást. (5.3. ábra a függelékben)

1.5.4. RTL

A *Register Transfer Language* (*Regiszterátviteli Nyelv*, RTL) a GCC legalacsonyabb szintű köztes nyelve [14, 15]. Sok optimalizáló algoritmus még a *middle end* alatt fut rajta, és ebből a formából generálja a *back end* a végső kódot is. Korábban ez volt az egyetlen köztes nyelve a fordítónak, nagyon alacsony szintje miatt azonban, az újabb, absztraktabb algoritmusok más nyelvek bevezetését is megkövetelték (*Tree*, *Tree SSA*).

RTL formában az utasítások az *assembly* programozási nyelvhez hasonlóan, egyszerű utasítások formájában követik egymást. Itt viszont még mindig nyelvfüggetlen (konkrétabban architektúra független) utasítások vannak, azaz ez a nyelv lényegében annyiban tér el az assemblytől, hogy olyan utasításokat illetve regiszter műveleteket használ, amelyekben nincsenek adott architektúrákra jellemző utasítások. Tulajdonképpen úgy is felfoghatjuk ezt a nyelvet, mintha egy absztrakt gépet programozhatnánk gépi kódon, aminek végtelen sok regisztere van. Az absztrakt gépünk regisztereiből ki-be tölthetünk értékeket, módosíthatjuk őket, programozhatjuk őket, majd a fordító a konkrét architektúrára legenerálja belőle a megfelelő kódot.

Az RTL kifejezéseket az „*RTL expressions*” angol kifejezésből RTX-nek rövidítik és a fordító az `rtx` struktúrában tárolja ezeket. Ez a struktúra a `tree` struktúrához hasonlón több adattagot tárol, amelyek közül az egyik legfontosabb az „RTX kód”. A fordító ennek a kódnak a segítségével különbözteti meg a különféle utasításokat és ez alapján a kód alapján sorolja azokat külön osztályokba is. (5.2. ábra a függelékben)

Az RTX-eknek az azonosítójuk mellett a `tree` *node*-okhoz hasonlóan vannak operandusaik és van egy *machine mode* adattagjuk is, ami az adott utasításban tárolt adat méretére és reprezentációs formájára utal. (5.3. táblázat a függelékben)

„n alatt a k” RTL példája

Az `nalattk.c` program RTL fázisáról a következő utasítással készítettem „dump” állományt:

```
gcc -c -fdump-rtl-all nalattk.c.
```

Mivel ez a fájl azonban mintegy 185 soros állomány és 46 külön RTX-et tartalmaz, ezért a teljes tartalmából csak az eredeti forrás fő ciklusát, azaz a

```
for (i = 1; i <= k; i++)
      nak = nak * (n - i + 1) / i;
```

utasításokat emeltem ki (még ez a ciklus is „alig” 10 utasításból tevődik össze). Az RTL sajátosságait és alacsony szintjét ez a kis szemelvény is nagyon jól szemlélteti a függelékben található 5.4. ábrán.

2. fejezet

A Columbus keretrendszer

Ebben a fejezetben a diplomamunkám fő tárgyát képező forráskód elemző szoftvert, a Columbus keretrendszert mutatom be. A keretrendszert forráselemzésre fejlesztették, ezért a Columbus bemutatásával először egy rövid bevezetést is szeretnék nyújtani a forráselemző szoftverek világába. Ezt követően egyesével mutatom be a keretrendszerben található alkalmazásokat és használatukat, majd részletesebben tárgyalom a Columbus „köztes kódjának" leíró nyelvtanát, az úgynevezett *Columbus Schema*-t. Utóbbit azért fontos külön kiemelnem, mert ezt a köztes nyelvet felhasználva fogom tudni összekapcsolni a Columbus *front end* alkalmazását – a CAN-t – a GCC-vel.

2.1. Bevezetés

Napjainkban az informatika dinamikus fejlődése átformálta mind a felhasználói, mind a fejlesztői igényeket. Egyre nagyobb és robusztusabb alkalmazások jelennek meg, hogy minél jobban kielégítsék a felhasználók telhetetlen elvárásait. A nagy projektek forrásaiban az egyes részek közötti kapcsolatokat azonban igen nehéz átlátni, és ez a szoftver forrásának bonyolultságával egyre inkább csak nehezedik. Ez eredményezte, hogy olyan szoftverek jelentek meg, amelyek a forráskódot elemzik, különböző metrikákat számolnak és segítenek átlátni a fejlesztőknek a forráskód szerkezetét. Az ilyen alkalmazások segíthetnek az esetleges hibák feltárásában, rámutathatnak arra, hol és hogyan érdemes módosítani, „újratervezni" a forrást.

Egy ilyen úgynevezett „*reverse engineering*" alkalmazás a *Columbus* is, amit a A Szegedi Tudományegyetem Informatika Tanszékcsoportjának dolgozói a helsinki Nokia Research Center-rel és a FrontEndART Kft.-vel együttműködve kezdtek fejleszteni [11, 10]. A Columbus egy keretrendszert ad nagy C/C++ programok elemzésére. A segítségével UML Osztály Modellt [38], *call graph*-ot (hívási gráf) [39] és rengeteg metrikát számolhatunk adott alkalmazásokon.

A Columbus összetettségét és hatékonyságát jellemzi, hogy több, nagyobb és közismert projekten is végeztek illetve végeznek a segítségével elemzéseket. A Columbus segítségével analizálták például a Mozilla [9, 12, 29] illetve az OpenOffice forráskódját is [21].

Ezeknek az elemzéseknek az alapját általában úgynevezett *metrikák* adják, amelyeket a forrásprogram egy kiemelt jellemzőjének a leírására használnak. Néhány a Columbus keretrendszer segítségével számolható ismertebb metrika:

WMC. (Weighted Methods per Class) az összes metódus és operátor együttes száma (az örökölteket nem beleszámolva)

DIT. (Depth of Inheritance Tree) az ősök száma

NOC. (Number Of Children) a közvetlen leszármazottak száma

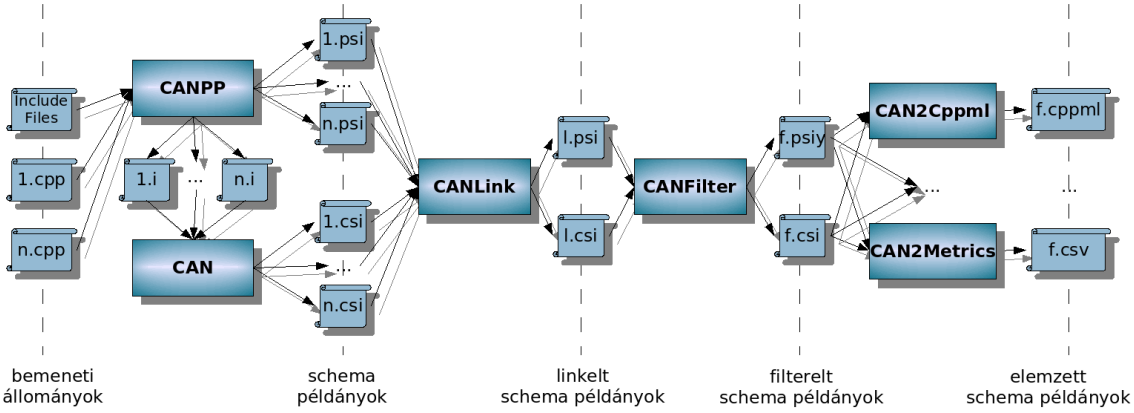
LCOM. (Lack of Cohesion on Methods) azoknak a függvény-pároknak a száma, amelyek nem használnak közös attribútumot, mínusz azon párok száma, amelyek használnak (az eredmény nulla, ha a különbség negatív)

LOC. (Lines Of Code) az osztály hasznos kódot tartalmazó sorainak száma (beleértve a tagfüggvények törzsét is)

NCL. (Number of Classes) az osztályok száma

2.2. A Columbus felépítése

A Columbus keretrendszer több kisebb alkalmazást foglal magába, amelyek mindegyike külön feladatokat lát el. Az elemzési folyamatban ezeket a programokat kell egymásután sorban meghívni a végső számítások elvégzésére [19] (2.1. ábra).



2.1. ábra. A Columbus elemzési folyamata.

2.2.1. Columbus REE

A *Columbus REE* (Reverse Engineering Environment) a keretrendszer legfőbb alkalmazása. Ez a program egy grafikus felhasználói felületettel rendelkezik, aminek a segítségével lényegében az egész keretrendszert irányítható. Az REE-ben minden feladatot plug-in modulokra osztottak ki a fejlesztők, amiknek köszönhetően könnyen továbbfejleszthetővé válik az egész rendszer.

A Columbus plug-inek három fő csoportba sorolhatók, amelyek a következők:

extractor plug-inek. Egy extractor plug-in feladata, hogy az inputként megadott forrásfájlt elemezve minál több adatot kinyerjen az adott programforrásból. Feladata továbbá, hogy az így nyert információt egy adott fájlba elmentse. Jelenleg a Columbusban ilyen plug-in csak C/C++ nyelvekre van implementálva.

Linker plug-inek. A linker plug-inek feladata, hogy egy projekt adott köztes reprezentációit összefűzze a memóriában. Az összefűzés közben pedig esetleges szűrési feladatokat is elláthat, amivel szabályozható az adott forrásból milyen mérési adatokat akarunk megkapni.

Exporter plug-inek. Az exporter plug-inek feladata, hogy a linker plug-in által felépített és megszűrt köztes reprezentációt különbböző output formátumokba átalakítsa. Ilyen output formátum lehet például a HTML, XML, ASCII formátum, vagy akár más elemzőprogramok formátumai is, mint a TDE Memaid 2.2, TED 1.0, Rational Rose vagy Microsoft Jet Database.

2.2.2. CANPP

A *CANPP* (C/C++ ANalayer-PreProcessor) egy parancssori program, aminek a feladata adott C/C++ forrásfájlok preprocesszálása azaz előkészítése a későbbi elemzéshez. Ebben a fázisban a CANPP elsősorban *include fájlok* és makrók behelyettesítését végzi.

Működését parancssori kapcsolókkal befolyásolhatjuk és megadhatunk neki kezdeti makró definíciókat, típusdefiníciókat vagy különböző elérési útvonalakat az include fájlokhoz.

A CANPP bemenete egy `.c` vagy `.cpp` esetleg `.h` fájl, kimenete pedig egy preprocesszált forrást tartalmazó `.i` fájl, illetve egy `.psi` fájl. Utóbbi a *Preprocesszált C++ Columbus Schema* [41] egy példányát tartalmazza bináris formában.

2.2.3. CAN

A CANPP (vagy esetleg más preprocesszor program[1]), által generált `.i` preprocesszált fájlt a *CAN* (C++ ANalyzer) segítségével elemezhetjük tovább. Ez az elemző a megadott forrásfájlt beolvassa és felépít egy saját ASG[2]-t a *C++ Columbus Schema* [8] alapján. Ennek az ASG-nek a GCC-hez hasonlóan az a szerepe, hogy az elemző számára egy absztrakt belső reprezentációs formát nyújtson a későbbi transzformációk elvégzéséhez.

A CAN bemenete tehát egy `.i` preprocesszált forrásfájl, a kimenet pedig egy `.csi` (*Columbus Schema Instance*) állomány, ami a *Columbus Schema* példányát tartalmazza bináris formában.

A CAN parsere az ANSI C++ nyelvnek több dialektusát is jól kezeli és ezekből a forrásfájlokból is helyesen tudja felépíteni az ASG-t. Ilyenek a Microsoft Visual C++ fordítója által használt, a Borland C++ Builder által használt és a GCC g++ fordítójának a dialektusai.

[1] Például a *gcc*-t a *-E* kapcsolóval futtatva is *.i* fájlt kapunk outputnak.

[2] Az ASG (*Abstract Syntax Graph*) az AST (*Abstract Syntax Tree*) egy általánosabb alakja, ami nem feltétlenül fa szerkezetű.

Fontos tulajdonsága továbbá, hogy a *parser* úgynevezett „*hibatűrő*” (*fault tolerant*) elemzést végez. Ez azt jelenti, hogy a CAN ASG építés közben nem áll meg a szintaktikai hibáknál, hanem képes azokat felismerni, majd figyelmeztet róluk és a felépített struktúrából kihagyja a hibás részeket. Ezáltal az ismeretlen dialektusú forrásokról vagy az esetleges preprocesszálási hibákat tartalmazó (kimaradt header fájlok, stb.) inputokról is készíthetünk alapvető méréseket.

2.2.4. CANLink

A *CANLink* (CAN Linker) egy linker eszköz a „*schema instance*” fájlokhoz, hasonlóan egy fordító programnak az objectfájlokat összefűző linkeréhez. Ennek a linkelésnek a segítségével az összetartozó forrásrészek a különböző schema példányokból egy közös példányba kerülhetnek. Az új közös példánynak a formátuma a korábbi formátummal megegyezik, tehát ugyanúgy lehet később kezelni, mint az eredeti példányokat, sőt, akár más fájlokkal is össze lehet később linkelni.

A CANLink bemenete több `.psi` vagy `.csi` fájl. A kimenete pedig az input formátumokkal megegyező formátumú állomány.

2.2.5. CANFilter

A „*schema instance*” fájlok tartalmát a *CANFilter* (CAN Filter) program segítségével szűrhetjük is. Megszabhatjuk, hogy milyen tulajdonságú (pl. milyen nevű) node-okat szeretnénk benne hagyni az adott ASG gráfban.

A bemeneti állománya a CANLink-nek egyetlen `.psi` vagy `.csi` fájl. A kimenet formátuma pedig az inputtal megegyezik.

2.2.6. Exporterek

Ezeknek a kisebb parancssori alkalmazásoknak a segítségével a „schema instance” fájlokon különböző méréseket vagy transzformációkat tudunk végezni. A segítségükkel tudjuk a forrást különböző formátumokba is exportálni, amelyek közül néhányat mutat be a 2.1. táblázat.

Formátum	Leírás
CPPML	C++ Markup Language (XML)
FAMIX	Famix XMI
GXL	Graph eXchange Language (XML)
HTML	HyperText Markup Language
RIGI	Rigi Standard Formati
UML	Unified Modelling Language (XMI)

2.1. táblázat. Néhány a Columbus által támogatott export formátum.

2.3. A C++ Columbus Schema

Az angol *schema* szó magyar jelentése *minta*, *vázlat*. A *C++ Columbus Schema* pedig a C++ nyelvnek egy olyan leírását (vázlatát) jelenti, amelyben az egyes nyelvtani elemeket külön osztályok példányai és azok attribútumai jellemzik. A nyelvtanban meghatározott szabályokat, azaz a nyelvtani elemek közötti kapcsolatokat, pedig a schema osztályainak példányai közötti kapcsolatok írják le. Ennek az absztrakt reprezentációnak a célja, hogy egy olyan általános leírást adjon a C++ nyelvről, amivel minden forrásprogram egyértelműen leírható egy olyan logikai struktúrában, ami alkalmas forráselemzési számítások elvégzésére.

A schema egy példányát *schema instance*-nak hívják és egy olyan megtestesülését jelenti a *C++ Columbus Schema*-nak, ami egy konkrét programot modellez.

A *C++ Columbus Schema* az objektum orientált megközelítésnek köszönhetően ábrázolható UML Osztály Diagrammok [38] segítségével. Az osztály diagrammok pedig egyértelművé teszik az implementálás módját is, és ami még fontosabb, hogy lehetőséget adnak egy API[3] elkészítésére, amivel a schema nagyon könnyen kezelhetővé válik. A Columbus fejlesztői környezete (*Columbus API*) minden fontos schema műveletet biztosít (schema példány bejárása, schema példány módosítása, stb.) ezért könnyen lehet a segítségével újabb alkalmazásokat fejleszteni, mint különböző *front end*-ek, *reverse engineering* alkalmazások, dokumentációs szoftverek, vagy akár fordító programok. Az egyik legnagyobb Columbus API-t használó alkalmazás a GXL (*Graph Exchange Language*) [23, 24], ami egy elfogadott, XML alapú általános formátumot biztosít *re-* és *reverse engineering* alkalmazások közötti adatmegosztásra.

Ennek a reprezentációnak fő szerepe a Columbus szerkezetében, hogy a schema adja a keretrendszernek a belső reprezentációs formáját, azaz a forrásprogramoknak az ASG szerkezetét. Ezzel a szereppel gyakorlatilag az egész keretrendszerben a legfontosabb, központi szerepet tölti be, hiszen a belső reprezentáción dolgozik az összes elemzési fázis. Igen fontos ezért, hogy egy jól strukturált és könnyen átlátható formában minden fontos információt le tudjunk írni a segítségével.

A *C++ Columbus Schema* az ISO/EIC C++ szabványt [28] követi, ami azt jelenti, hogy a schema-t a preprocesszált, úgynevezett „tiszta” C++ szintaktika leírására fejlesztették. Nem kezeli tehát a makrókat és egyéb preprocesszáláshoz szükséges nyelvtani elemeket, arra a fejlesztők egy másik struktúrát, a *Preprocesszált C++ Columbus Schema*-t vezették be [41].

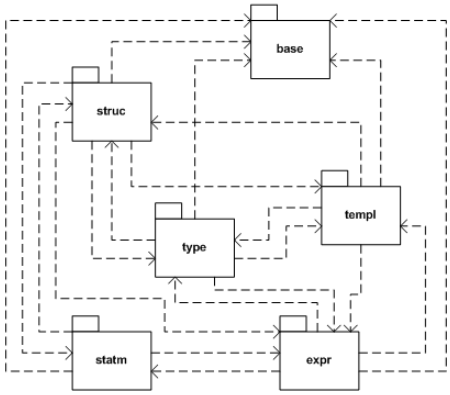
2.3.1. A schema szerkezete

A C++ nyelv összetettsége miatt a schema is több nagyobb modulból épül fel, amelyek a nyelvtan különböző, fontosabb egységeit írják le. A fő modulok (2.2. ábra) a következők:

base. A base csomag a schema a többi moduljához szükséges alapvető osztályokat és típusokat tartalmazza.

struc. Ebbe a csomagba a főbb programrészek kerülnek hatáskörük szerint, mint az objektumok, függvények, illetve osztályok.

[3] API - Application Programming Interface, fejlesztői felület



2.2. ábra. A Columbus Schema csomagjai.

type. Ebben a csomagban a *struc*, *templ* és *expr* csomagokban definiált nyelvtani egységek típusainak leírásához szükséges osztályok szerepelnek.

templ. A template-ek (sablonok) jellemzéséhez szükséges osztályok csomagja.

statm. Az utasítások jellemzéséhez fontos osztályok.

expr. A különböző kifejezések jellemzéséhez fontos osztályok.

A következő fejezetekben az említett modulokat mutatom be röviden a schema jellemzéséhez használt UML Osztály Diagrammok segítségével. Ezek a diagrammok méretük miatt a függelékben találhatók.

2.3.2. A *base* csomag

A *base* csomag tartalmazza azoknak az osztályoknak és típusoknak a definícióit, amelyek a többi csomaghoz szükségesek (5.5. ábra a függelékben).

Ezek közül az egyik legfontosabb a `Base` osztály, ami minden a schema-ban használt osztály őse. Egyetlen attribútuma egy `id`, aminek a segítségével az ASG-nek minden csúcsa saját azonosítót kap. A `Base` osztályból származik a `Positioned` osztály, ami egyben a `Named` osztály őse. Utóbbival olyan nyelvtani elemeket írhatunk le, amelyeknek saját azonosítójuk van a forrásban (pl. változók, függvények, stb.), a `Positioned` osztály pedig gyakorlatilag minden forrásbeli elem jellemzésére használandó, hiszen olyan fontos attribútumai vannak, mint az adott elem sorának és oszlopának a száma.

A csomag többi része általában olyan felsorolás (*enumeration*) típus, amelyek segítségével egyes nyelvtani kategóriák külön csoportjait azonosíthatjuk be. Ilyen felsorolás típus például a `BinaryArithmeticKind`, ami a bináris aritmetikai operátorokat sorolja fel, mint a szorzás (`bakStar`), osztás (`bakDivide`), stb.

2.3.3. A *struc* csomag

A *struc* csomagba olyan nyelvtani elemek kerülnek amelyek általában saját hatókörrel (*scope*) rendelkeznek a programon belül (5.6. ábra a függelékben). Ezek az egységek mind a `Member` osztály köré szerveződnek, ami egy absztrakt osztály olyan elemek

jellemezésére, amiket egy scope-ba be tudunk sorolni. Ilyenek lehetnek a függvények (`Function`), object-ek, azaz változódeklarációk (`Object`), a scope-ok (`Scope`) vagy maguk az osztályok (`Class`) is.

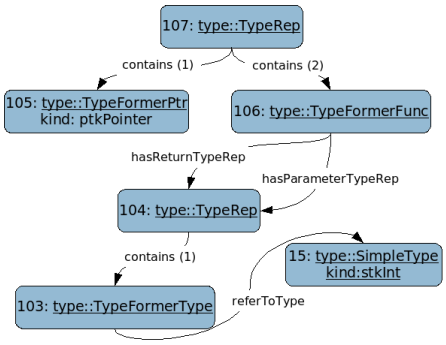
2.3.4. A *type* csomag

A C++ nyelv által leírható típusok jellemzéséhez szükséges osztályokat találhatjuk a *type* csomagban (5.7. ábra a függelékben).

Egy C++ típus reprezentációját a `TypeRep` osztály egy példánya adja, ami több kisebb egységből, `TypeFormer` osztályokból épül fel. A `TypeFormer` osztályok egymás utáni szekvenciában határozzák meg, hogy az adott típus milyen elemekből tevődik össze. Ilyen elemek lehetnek az egyszerű típusok (`SimpleType`), tömb típusok (`TypeFormerArr`), pointer típusok (`TypeFormerPtr`), vagy akár a függvény típusok is (`TypeFormerFunc`).

Ennek az ábrázolásmódnak egy igen nagy előnye, hogy minden – a programban használt – típust egyszer kell létrehozni és eltárolni a memóriában. Nem kell így később minden alkalommal újra és újra felépíteni a bonyolultabb típusokhoz tartozó fákat, hanem elég csak a már korábban felépített példányra hivatkozni.

A 2.3. ábra egy példát mutat a `int (*f)(int)` típus schema szerinti reprezentációjára. Jól látható, hogy az `int` típusnak csak egy `TypeRep` osztályt példányosít a Columbus illetve, hogy a létrehozott függvénytípus visszatérési értéke és paramétere is ugyanarra az objektumra fog mutatni.



2.3. ábra. A *int (*f)(int)* típus ábrázolása a schema alapján.

2.3.5. A *templ* csomag

A schema *templ* csomagja a C++ sablonjait tartalmazza (5.8. ábra a függelékben).

Az osztály és függvény sablonoknak a C++ nyelvben két fő jellemzőjük van:

– a template paraméterlistája (`ParameterList`),

– és az argumentumlistája (`ArgumentList`).

A template paraméterek szimbólumok, amik a template példányosításakor kapnak értéket az argumentum alapján. Egy paraméter jelölhet egy típusnevet (`ParameterType`),

egy másik sablont (`ParameterTempl`) vagy valamilyen „nem-típus” (pl. konstans) értéket (`ParameterNonType`).

Az argumentumok már nem csak szimbólumok, hanem tényleges típusok (vagy konstans értékek), amelyek egy template példány argumentumait jelölik. Ezek értelemszerűen ugyanolyan kategóriákba sorolhatóak, mint a paraméterek.

2.3.6. A *statm* csomag

A *statm* csomag tartalmazza egy C++ forráskódnak a lehetséges utasításait (amik egy függvény törzsében vagy egy block-on belül találhatók) leíró osztályokat (5.9. ábra a függelékben).

A csomagban található utasítások a `Statement` absztrakt osztály köré szerveződnek, ami a `base::Positioned` osztályból van származtatva.

Ebbe a csomagba olyan alapvető utasítások tartoznak, mint a `Block`, `Try-Catch` blokkok, a szelekciós vezérlés utasításai (`If`, `Switch`), a különböző ismétléses vezérlések utasításai (`While`, `For`, `Do`) vagy a forráskódban az ugrásokhoz tartozó utasítások (`Break`, `Continue`, `Return`, `Goto`). Ide tartoznak még a különböző címkék is, mint a `CaseLabel`, `DefaultLabel` is `IdLabel`, de ezeket leíró osztályok már nem a `Statement` osztályból, hanem az ugyancsak absztrakt `Label` osztályból vannak származtatva.

2.3.7. Az *expr* csomag

A *struc* csomagnál talán egy kicsit egyszerűbb, de méretében a *statm* csomaghoz hasonlóan nagy és igen fontos csomag a schema-ban az *expr* csomag is. Ebbe a csomagba a C++ forrás lehetséges kifejezései kerülnek (5.10. ábra a függelékben).

A csomagban gyakorlatilag az `ExpressionList` osztály kivételével minden egyéb osztály az `Expression` absztrakt osztályból van származtatva. Az `ExpressionList` pedig egy olyan konténer osztály, ami `Expression` osztályok rendezett listáját valósítja meg.

Ide tartoznak az operátorok külön csoportosítva egy operandusú (`Unary`) illetve két operandusú (`Binary`) operátorokként. Valamint itt találhatóak még az operátorok mellett az olyan fontos kifejezések is, mint a literálok (a `Literal` absztrakt osztályból származtatva) és a `New`, `This`, `Throw`, `Conditional` kifejezések illetve az azonosítókat leíró `Id` osztály.

3. fejezet

A kiterjesztés megvalósítása

Ebben a fejezetben a diplomamunkám feladatkiírásának megoldását, Columbus CAN *front end* alkalmazásának és a GCC fordító program összekapcsolásának módját mutatom be. Mindkét alkalmazás, a Columbus és a GCC is, nagy és robusztus alkalmazás hatalmas és összetett forráskóddal. Ezért ahhoz, hogy megértsük, hogyan lehet egy fordítási folyamatként összilleszteni a két rendszer *front end*, *middle end* és *back end* részeit, fontos ismernünk a GCC forrásának és a Columbus fejlesztői környezetének a felépítését is. A korábbi fejezetekben mind a GCC-t, mind a Columbus keretrendszert csak általánosan, használatuk és felépítésük logikai struktúrájának alapján mutattam be és nem tértem rá az implementálási részletekre. Rövid bevezető után ezért ebben a fejezetben először ismertetem a GCC forrásszerkezetét (3.3.1. fejezet), majd a Columbus API nyújtotta lehetőségeket (3.3.2. fejezet). Magának a kiterjesztés leprogramozásának részleteire pedig csak ezt követően (3.3.1. fejezet) térek rá.

3.1. Bevezetés

Diplomamunkám fő feladata a GCC fordítót hozzáilleszteni egy új *front end* alkalmazáshoz, a Columbus keretrendszer CAN forráselemzőjéhez. Ennek a kiterjesztésnek a segítségével a Columbus-t gyakorlatilag felruházzuk a GCC képességeivel, és olyan fordítási folyamattal bővítjük ki, ami kezdeti fázisaiban alkalmas újabb, magasabb szinten végrehajtható műveletek elvégzésére. A Columbus *absztrakt szintaxis gráfja* (2.3. fejezet) ugyanis a modern objektum orientált megközelítésnek és a bonyolultabb *reverse engineering* célokra tervezett, logikus szerkezetének köszönhetően olyan transzformációkra (pl. optimalizálásokra) is alkalmas, amelyekre a GCC fordítási folyamatokra optimalizált felépítése miatt alkalmatlan.

Ennek a kiterjesztésnek egy lehetséges megvalósítása – amit diplomamunkámban is bemutatok –, hogy a Columbus API segítségével egy újabb modult ágyazunk be a GCC forrásába. Ez a modul, amit később *Columbus-GCC Converternek* nevezek el (továbbiakban csak Converter), kapcsolja össze a Columbus keretrendszerrel a GCC-t. Mivel a GCC forrásában ilyen általános modulok támogatásához nincsen külön keretrendszer, ezért a forrásba való beágyazást saját eszközökkel kell megoldani.

3.2. A kiterjesztés elméleti háttere

A kiterjesztés megértéséhez el kell különítenünk az elméleti és a gyakorlati hátteret. Különösen, mert a két igen bonyolultan felépített alkalmazás – a GCC és a Columbus – összekapcsolásához használt implementálási (azaz gyakorlati) technikák megértéséhez mindenképpen bővebb elméleti magyarázatra van szükség.

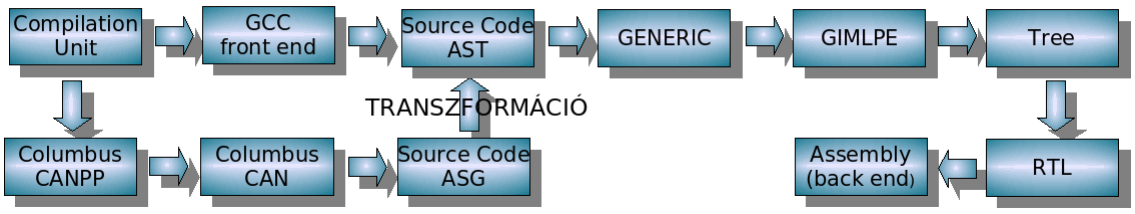
A következő alfejezetekben ezért elméleti megközelítésből mutatom be, hogyan kapcsolhatjuk össze a forráselemzőt és a fordító programot.

3.2.1. A GCC és Columbus összekapcsolása

Ahhoz, hogy ezt a két hatalmas projektet, a Columbust és a GCC fordítót összekacsolhassuk, meg kell találni a GCC fordítási folyamatában azt a pontot, ahol be tudunk csatlakozni ebbe az igen összetett a folyamatba. Magát a fordítási folyamatot először általánosan, majd a GCC köztes nyelveire fókuszálva már bemutattam az 1.4. és 1.5. fejezetekben. Azt már ismerjük tehát, hogy a GCC *front end* része először AST szinten építi fel a forrásprogram köztes kódját, majd ezt a magas szintű még nyelvi elemeket is tartalmazó reprezentációs nyelvet konvertálja alacsonyabb, nyelvfüggetlen köztes reprezentációra, a GENERIC formára.

Mint azt a 2.3. fejezetből megismerhettük, a Columbus ASG nyelvezete egy igen magas szintű köztes nyelv. A Columbus ezzel az egyetlen köztes nyelvvel dolgozik, és ezen a nyelven sem végez igazán transzformációkat, hanem felépítés után csak arra használja, hogy adatokat nyerjen ki belőle. Csupán két olyan fázis van ebben a folyamatban, amikor módosítja az ASG-t, a *CANLink* és a *CANFilter* fázisa, de igazából szerkezeti átalakítást ezekben sem végez rajta, pusztán forrásfájlok összefűzését illetve néhány adat szűrését végzi el. A Columbus oldaláról tehát egyértelműnek tűnik, hogy az ASG, egészen pontosan az ezt bináris formában tároló *schema instance* fájl (azaz a `.csi` fájl) lesz az, amivel majd dolgoznunk kell. Ezt az állományt kell majd beolvasni a fordítóval, adatokat nyerni belőle, és rávenni a fordítót, hogy az így nyert információ alapján fordítson tovább.

Magas reprezentációs szintje miatt szerkezetében és felépítésében a Columbus ASG nyelvét leginkább a GCC AST nyelvéhez hasonlíthatjuk, amelyet korábban a 1.5.1. fejezetben mutattam be. Ahhoz, hogy a Columbust és a GCC-t összekapcsolhassuk tehát, a fordító *front end* részének az AST építése környékén kell nekünk is bekapcsolódnunk.



3.1. ábra. A Columbus és a GCC összekapcsolási pontja a fordítási folyamatban.

A fordítási folyamatba tehát úgy tudunk belépni, hogy a GCC forráselemzési és AST építési folyamata helyett, mi magunk olvassuk be az AST építéséhez szükséges adatokat egy `.csi` fájlból. Az így nyert forrásadatok alapján pedig mi magunk építjük fel az AST fát. Ez a folyamat valójában nem más, mint egy átalakítás, transzformáció, amely során

egy adott Columbus ASG-t egy megfelelő GCC AST formába transzformálunk át. Ennek a transzformációnak a fordítási folyamatba történő bekopcsolását szemlélteti a 3.1. ábra.

3.2.2. A Columbus ASG - GCC AST transzformáció

A Columbus kiterjesztéséhez az igazi feladat valójában a Columbus ASG gráfjának a GCC AST gráfjába történő átalakítása.

Programozói megközelítésből ehhez a faladathoz, két lényegében teljesen eltérő adatszerkezetről tudjuk biztosan, hogy eredetileg ugyanazt az adathalmazt, a forrásprogramot reprezentálják. Illetve abban is biztosak lehetünk, hogy ezen a két reprezentáción az eltárolt adathalmaz is ugyanaz. Nem áll fent egyik oldalon sem az tehát, hogy a forrásprogramból más jellegű információt nyernének ki. (Ha az történne például, hogy a GCC csak az `if` utasítások szerkezetét tárolja, a Columbus, pedig csak a `for` utasításokét, akkor igen nagy bajban lehetnénk a transzformációval.)

A Columbus a forrásprogramot különálló osztályok közötti aggregációkkal és asszociációkkal írja le, amíg a GCC minden nyelvi elem reprezentálására ugyanazt a `tree` struktúrát használja. A Columbus szemlélete egy magasabb szintű objektum orientált megközelítés, a GCC szemlélete pedig egy alacsonyabb szintű funkcionális megközelítés.

A két külön szemlélettel leírt adatszerkezetet úgy lehet összehozni, hogy a Columbus gráfját bejárva, minden csúcsból minden tárolt adatot kinyerünk, majd felépítjük az adott csúcsnak megfejelő GCC struktúrát. A felépített GCC struktúrának az adatmezőit pedig az ismert adatok alapján kitöltjük. Ezzel a kitöltéssel együtt fogjuk behúzni az új gráfban az éleket is. Ha pedig az adatfeltöltés közben bármilyen hibát észlelünk, úgy az átalakítást nem tudjuk elvégezni.

Elméletben ez egyszerűnek tűnik, de a gyakorlatban sok apróbb részletre is oda kell figyelni, amelyeket a következő alfejezetben ismertetek.

3.3. A kiterjesztés implementálása

3.3.1. A GCC forrásszerkezete

A GCC forrása a fordító honlapjáról [13] és a fejlesztői SVN[1] szerverről szabadon, ingyenesen letölthető. Diplomamunkám elkészítéséhez először a mainline GCC (4.2-es, fejlesztői verzió) 2006. szeptember 9-én letöltött verzióját használtam. Ez a verzió azonban még a fejlesztésnek olyan fázisában volt, amikor a fordító fejlesztői sok hibajavítást és módosítást vittek fel a forráskódba. Ezért még a fejlesztés közben, 2007. február 7-én (amikor már olyan fázisba ért a GCC fejlesztése, hogy a súlyosabb hibákat javították), újrafrissítettem a diplomamunkámhoz használt GCC forrását.

A fordító forrásának mérete még tömörítve is közel 40 MByte (*bz2* formátumban), kitömörítve pedig több, mint 350 MByte. Ez a hatalmas forrás alig 35000 fájlból valamint 2000 könyvtárból tevődik össze. Ebben a fejezetben ezért nem is törekszem arra, hogy teljes precizitással minden összetevőjét ismertessem ennek a hatalmas forrásnak, csupán azokat a fontosabb komponenseket emelem ki, amelyek az én munkámhoz is fontosak

[1] SVN - Subversion, egy verziókövető alkalmazás

voltak. Előbbire egyébként a fejlesztők is csak felületesen vállalkoznak a GCC fejlesztői leírásának [14] „*Source Tree Structure and Build System*" (Forrásfa Szerkezete és a Fordítási Rendszer) fejezetében.

A forrás főkönyvtárában egy `gcc` könyvtáron belül találhatók a fordítónak a fő összetevői. Itt vannak például az egyes optimalizáló algoritmusok implementációi, a különböző architektúrák támogatásai és a *front end*-ek is. Emellett a könyvtár mellett a főkönyvtárban elsősorban különböző library-k, scriptek és a fordításhoz szükséges egyéb komponensek találhatók. Itt található a `libcpp` könyvtárban többek között a C preprocesszor is.

A `gcc` könyvtáron belül alkönyvtárakban találhatóak az egyes *front end*-ek, mint a C++ a `cp` könyvtárban, az Ada az `ada` könyvtárban, stb. Egy `config` könyvtáron belül pedig a *back end* külön architektúrákhoz tartozó részei. Magában a `gcc` könyvtárban a *middle end* optimalizáló algoritmusai és a C *front end* forrása mellett az egyéb fontos forrásfájlok vannak, mint például a *Tree* és *RTL* implementációi.

Diplomamunkám során leginkább a C++ *front end* forrásállományait módosítottam, illetve használtam. A használt forrásfájlok közül pedig a munkámhoz fontosabbakat a 3.1. táblázat mutatja be.

Fájlnév	Leírás
c-common.c	globális változók, kapcsolók változói, általában fontos forráselemek
c-common.def	GENERIC és egyes nyelvekhez tartozó azonosítók
c-common.h	korábbiakhoz tartozó header fájl
c-opts.c	indulási beállítások, pl. a fordítási kapcsolók
c-parser.c	C parser
c.opt	fordítási kapcsolók beállításai
cp/class.c	C++ osztályok kezeléséhez fontos forráselemek
cp/cp-tree.def	a *Tree* C++ struktúrában használt TREE_CODE azonosítók
cp/cp-tree.h	a *Tree* struktúrához tartozó C++ specifikus deklarációk
cp/decl.c	C++ eljárás- és változódeklarációk kezeléséhez használt forráselemek
cp/decl2.c	C++ eljárás- és változódeklarációk kezeléséhez használt forráselemek
cp/parser.c	C++ parser
cp/semantics.c	C++ a parser szemantikai elemzője

3.1. táblázat. A Converter szempontjából fontosabb GCC forrásállományok.

3.3.2. A Columbus API

A *Columbus API* a *Columbus Schema* (2.3. fejezet) köré épülő fejlesztői környezet. A segítségével a keretrendszert olyan új tulajdonságokkal egészíthetjük ki, mint például az ASG-n végzett transzformációk vagy módosítások, és új exportálási formátumok.

A fejlesztői környezet egy logikus struktúrában biztosít eljárásokat ahhoz, hogy az ASG-n gyakorlatilag minden műveletet elvégezhessünk. Tetszés szerint bejárhatjuk, minden információt kinyerhetünk belőle és mindezek alapján kényelmesen módosíthatjuk is.

Az schema-ban ismertetett egyes csomagokat külön névterekben különíti el az API, így könnyen felismerhetőek az egy csomagba tartozó utasítások és osztályok is. Az API összes utasítása a `columbus` névtéren belül található, amin belül a C/C++ *front end* egy külön `cpp` névtérrel van elkülönítve. A `columbus::cpp::base` névtér jelöli tehát a C++ schema *base* csomagját.

Az ASG bejárását az API a közismert tervezési mintákból (az úgynevezett *design pattern*-ekből) átvett algoritmusokkal segíti [20]. A további alfejezetekben ezeket az algoritmusokat ismertetem.

A „*Visitor*” tervezési minta

Az ASG bejárásához az egyik leghatékonyabb módszer a tervezési mintákból ismert „*Visitor Pattern*”. Ennek a mintának a segítségével egy adott kollekció osztályai fölött (az ASG esetében az ASG gráf csúcsain), definiálhatunk műveleteket egy külső osztály segítségével anélkül, hogy magukat elemeket a módosítanánk. Egy kicsit egyszerűbb formában ismertetve a lényeget, arról van szó, hogy egy bejárási algoritmus alapján végigjárjuk az összes elemét a gráfnak és minden aktuálisan érintett elemre meghívunk egy *visit* eljárást, ami egy adott műveletet hajt végre az adott csúcson.

A Columbus API leggyakrabban használt visitor algoritmusa a *pre-order* azaz mélységi bejáráson [3] alapszik, ami (rekurzív definíciójában) az ASG fa szerkezetében minden csúcs további testvéreinek a bejárása előtt az adott csúcs gyermekeit járja be. A Columbus API-ban ezt az algoritmust a `columbus::cpp::AlgorithmPreorder` eljárás segítségével hívhatjuk meg, ami egy absztrakt `columbus::cpp::Visitor` osztályból származtatott osztályt vár paraméteréül.

Egy visitor osztály felépítését a 3.2. ábra szemlélteti. Minden olyan osztálytípushoz, amin szeretnénk, hogy egy általunk végrehajtandó művelet lefusson, egy `visit` metódust kell létrehoznunk egy olyan paraméterrel, aminek típusa az érinteni kívánt osztály típusával egyezik meg. Ilyen a példában a `ConverterVisitor` osztály egyetlen `visit` metódusa, ami osztályokon fut majd le. Hasonlóan a `visitEnd` metódus pedig a következő csúcsra lépés előtt, a `visit` lefutása után fog majd végrehajtódni. A `Factory` típusú `_fact` adattag pedig az ASG szerkezetéről tárol adatokat ugyancsak egy *design pattern*, a *Factory Pattern* alapján.

Az „*Iterator*” tervezési minta

Egy másik fontos és sokat használt tervezési minta, amivel a Columbus API dolgozik, az „*Iterator Pattern*” [20]. Ezt az egyik legegyszerűbb, és leggyakrabban használt mintának ismernek, amit arra lehet használni, hogy lista, vagy valamilyen kollekció elemeit sorban végigjárva adatokat nyerjünk ki az elemekből, vagy utasításokat hajtsunk végre rajtuk.

A Columbus a *schema*-ban sok adatot tárol olyan formában, amely elemeit iterátorok segítségével érhetünk el közvetlenül. Csak a legalapvetőbb példát tekintve, egy függvény törzsében, vagy egy block-ban lévő utasításokat is iterátorok segítségével járhatunk végig.

Az iterátorok jellemzően olyan *interface* felülettel vannak definiálva, amelyek lehetővé teszik az első elem vagy az utolsó elem elérését, lehetővé teszik az iterátor következő elemre való léptetését. Nyújtanak továbbá egy olyan ellenőrzést is, amivel lekérdezhetjük, hogy van-e még következő elem, amire átléphetünk. Ezeket az opciókat nyújtja a Columbus fejlesztői felülete is (3.3. ábra).

```
class ConverterVisitor : public columbus::cpp::Visitor {
public:
    ConverterVisitor(columbus::cpp::Factory* fact,
        columbus::cpp::AlgorithmClassDiagram& ac) : _fact(fact) {};

private:
    columbus::cpp::Factory* _fact;

    // Visit CLASS
    virtual void visit(const columbus::cpp::struc::Class& _node) {
        std::cout << "A " << _node.getName()
                << " osztalyon hajtok vegre muveletet." << std::endl;

        // muvelet ...
    };

    virtual void visitEnd(const columbus::cpp::struc::Class&) {
        std::cout << "Az osztalyon a muveletet befejeztem." << std::endl;
    };

    // egyeb visitor methodusok ...
};
```

3.2. ábra. A Columbus egy példa *Visitor* osztálya.

```
//...

// mutasson az iterator az elso elemre
statm::Block::ConstIterator it = _node->constIterator();

// amig van kovetkezo elem
while (it.hasNext()) {
    // legyen az stmt az aktualis elem, majd lepjen egyet az iterator
    const base::Positioned& stmt = it.next();

    std::cout << "Most a " << cpp::AlgorithmCommon::NodeId2str(col_stmt.getId())
        << " utasitason vegzek muveletet." << std::endl;

    // valamilyen muvelet vegrehajtasa
}

// ...
```

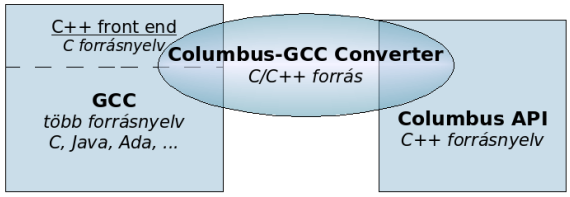
3.3. ábra. Példa egy *block* utasításainak bejárására *iterátor* segítségével.

3.3.3. A Converter implementálása

A kiterjesztés implementálásához, mint korábban írtam, a mainline GCC (4.2-es, fejlesztői verzió) 2006. szeptember 9-én letöltött verzióját használtam. A Columbus fejlesztő felületéhez pedig a 3.6-os, még ugyancsak fejlesztői verziót kaptam meg a Szegedi Tudományegyetem Szoftverfejlesztés Tanszékétől. Mindkét alkalmazás fordítható POSIX illetve Windows rendszerekre is, én a GCC GNU volta miatt azonban Debian/GNU Linux rendszeren dolgoztam.

A két rendszer összetettsége miatt a fejlesztést hosszas kutatási munka előzte meg, hogy megtalálhassam a 3.2.1. fejezetben írt kapcsolási pontot, illetve, hogy utánajárjak a két ASG felépítésének is.

Magát az alkalmazást *Columbus-GCC Converter*-nek neveztem el, utalva a két külön alkalmazásra és a két köztes nyelv közötti átalakításra, a konvertálásra. Az implementálási munka igen hosszas volt és sok utánajárást is igényelt, hiszen két nagyon összetett rendszert kellett összekapcsolni (3.4. ábra). Ezt a keletkezett forrás mérete is szemlélteti, hiszen közel 100 forrásállomány és hozzávetőleg 7700 sor forráskód keletkezett a munkám során.



3.4. ábra. A *Columbus-GCC Converter* és a két alkalmazás kapcsolata.

A tényleges implementálásnak az összekapcsolási pont megtalálása után kezdhettem csak neki. A csatlakozási pont megtalálása után, először fel kellett építenem egy keretrendszert a *Converter* számára, hogy a két robusztus, ráadásul külön forrásnyelven írt alkalmazást összekötve dolgozhassak. A transzformáció lépéseit (a visitor metódusokat, segédeljárásokat, stb.) csak ezután kezdhettem el. Ebben az alfejezetben is ezek szerint a lépések szerint, további alfejezetekre bontva tárgyalom az implementálás egyes részleteit.

Az összekapcsolási pont

A GCC magát a parsert a `c-opts.c` forrásfájlban található *c_common_parse_file ()* függvényhívás alatt inicializálja és hívja meg egy *c_parse_file ()* eljárással. Ez az eljárás végzi az elemzési fázisokat és építi fel az AST-t is. Ezt követően a *finish_file ()* függvény hívódik meg, ami további nyelvfüggő transzformációkat végez az AST-n (default konstruktorok, destruktorok, template példányosítások, stb.), majd meghívja a *genericizer*-t, hogy átadja a felépített fát a *middle end* számára.

Az összekapcsoláshoz tehát a *c_parse_file ()* függvényhívást kell kiküszöbölni és lecserélni egy olyan saját eljárásra, ami a `.csi` fájlt betölti, az abban tárolt ASG-t bejárja, és eközben a bejárás közben felépít a memóriában egy ezzel ekvivalens GCC AST-t. Ha ezt az AST-t sikerült hibátlanul felépíteni, akkor az irányítást visszaadhatjuk a GCC-nek, és a *finish_file ()* elvégezhetné az általunk felépített fán a *middle end* számára fontos transzformációkat, majd folytatódhat tovább a fordítási folyamat.

A megtalált összekapcsolási pont helyességét azzal ellenőriztem, hogy létrehoztam a GCC-ben egy új kapcsolót, aminek az volt a szerepe, hogy a parsert kihagyja a fordítási folyamatból. Ez a kapcsoló `-fnoparse` paraméter, ami csak annyit módosított a fordítási folyamaton, hogy a *c_parse_file ()* függvényhívás előtti feltétel szerint a parser *ne* hívódjon meg, ha ez a kapcsoló ki van adva. Egyébként, normális esetben, ugyanúgy meghívódik a parser, mint korábban. Ettől a kapcsolótól első lépésben azt vártam, hogy bármilyen forrásbemenetre sikeresen létrehoz így a fordító egy üres objectfilet, ami a célrendszer szabványainak megfelel. Tehát nem egy teljesen üres 0 byte méretű fájlt vártam kimenetnek, hanem egy olyan objectfilet, amiben a betöltéshez fontos fejlécek helyesen vannak kitöltve. Ez azért különösen fontos, mert így ellenőrizhető, hogy az említett függvény kihagyásával tényleg csak az AST építést hagytuk ki, és ettől még a fordításhoz szükséges inicializálásokat a fordító maradéktalanul elvégzi. A kapcsoló a vártaknak megfelelően jól működött, és gyakorlatilag bármilyen inputállományból (szintaktikailag rossz inputokból is) létrejött a fordítás után egy üres objectfile.

A `-fnoparse` kapcsoló mellé egy másikat, a `-fcsi-file=<filenév>` kapcsolót is bevezettem, ami arra szolgál, hogy *c_parse_file ()* hívás után meghívja magát a *Converter*-t, az ASG felépítésére. Ez a Converter hívás egy *columbus_gcc_converter ()* függvény hívásával történik, ami már a converternek egy önálló forrásállományában található.

A forrás szerkezete

A Converter forrását a GCC forrásfájába ágyaztam bele a `gcc/cp/columbus` alkönyvtárba. Ezen a könyvtáron belül pedig külön elkülönítettem a *Columbus API* library és fejléc állományait, valamint a Converter forrásállományait is a *schema* felépítése alapján strukturáltam. Az így elkészült forrásfa szerkezetét a 3.2. táblázat mutatja be.

Könyvtárnév	Leírás
converter	A Columbus-GCC Converter forrásai
converter/templ	*templ* csomag konvertálásához tartozó forrásrészek
converter/statm	*statm* csomag konvertálásához tartozó forrásrészek
converter/base	*base* csomag konvertálásához tartozó forrásrészek
converter/struc	*struc* csomag konvertálásához tartozó forrásrészek
converter/type	*type* csomag konvertálásához tartozó forrásrészek
converter/expr	*expr* csomag konvertálásához tartozó forrásrészek
converter/inc	A Converter include állományai
converter/inc/templ	A *templ* csomaghoz tartozó include állományok
converter/inc/statm	A *statm* csomaghoz tartozó include állományok
converter/inc/base	A *base* csomaghoz tartozó include állományok
converter/inc/struc	A *struc* csomaghoz tartozó include állományok
converter/inc/type	A *type* csomaghoz tartozó include állományok
converter/inc/expr	A *expr* csomaghoz tartozó include állományok
include	A *Columbus API* include állományai
lib	A *Columbus API* lib állományai

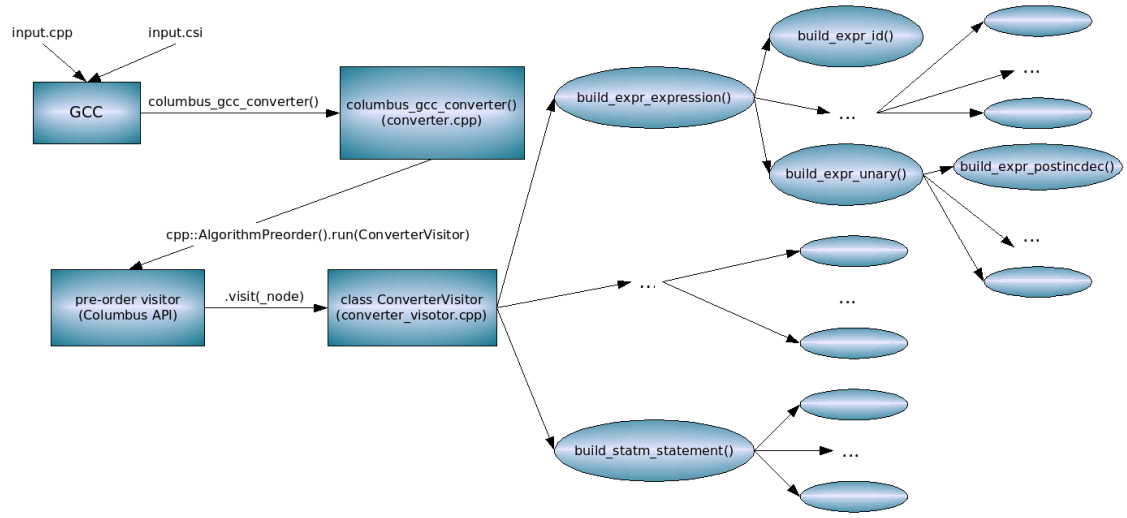
3.2. táblázat. A *Columbus-GCC Converter* forrásfájának felépítése.

A Converter maga is C++ nyelven íródott, de nagyon sok C nyelvi megoldást is alkalmaz. Ennek a keveredésnek az oka, hogy még a Columbus API egy tisztán C++ forráson alapszik, a GCC fordítónak a C++ *front end* része tiszta C nyelven íródott. Magát a Convertert tehát úgy kellett beágyazni a GCC forrásába, hogy eközben egyfajta hidat is képezzen a két nyelvi megvalósítás között.

Ehhez a köztes C/C++ szerephez módosítani kellett a GCC *build* mechanizmusát, hogy a `g++` összelinkelését ne a `gcc` fordító végezze, hanem a `g++` (a GCC fordítását ugyanis ugyancsak egy már lefordított GCC fordítóval végezzük). A Converter forrásában továbbá, több helyen is – elsősorban a GCC header fileok includeolásánál – használni kellett az `extern "C"` kulcsszót, hogy a fordító helyesen tudja kezelni az eredetileg C nyelven deklarált függvényeket.

A forrás logikai struktúrája két fő állomány köré épül. Az egyik a korábban említett *columbus_gcc_converter ()* hívást definiáló `converter/converter.cpp`, a másik pedig a *ConverterVisitor* Visitor osztályt definiáló `converter_visitor.cpp` (illetve az osztályhoz tartozó header file, a `convert_visitor.h`).

A `converter.cpp`-ben a *columbus_gcc_converter ()* függvény kifejtésén kívül egyéb nem található. Ennek a függvénynek a szerepe tehát, hogy megnyissa a paraméterként átadott `.csi` fájlt és elindítsa rajta *pre-order visitor bejárást*, valamint lekezelje az esetlegesen visszadobott hibákat. Ha pedig ilyen hiba nem volt, akkor visszaadja a GCC-nek az irányítást. Az új GCC AST az itt indított visitornak a futási eredményeként fog felépülni.



3.5. ábra. A *Columbus-GCC Converter* logikai felépítése.

A 3.5. ábra szemlélteti a Converter meghívásakor történteket. A GCC tehát egy függvényhívással indítja el a *Columbus-GCC Convertert*, ami egy visitor segítségével járja be a megadott `.csi` file által reprezentált ASG-nek az összes csúcspontját. Minden csúcspont típushoz létezik egy `build_csomag_tipus(node)` eljárás. Ezek a segédeljárások végzik el a megfelelő típusú csúcsok szükség szerinti bejárását és átalakítását a GCC AST nyelvére. A segédfüggvények visszatérési értéke a paraméterként megadott csúcsnak megfelelő `tree` struktúra lesz.

A *ConverterVisitor* osztály

A *ConverterVisitor* osztály központi szerepet tölt be a forrás szerkezetében. Ennek az osztálynak a megfelelő `visit` metódusait hívja ugyanis meg a mélységi bejárás közben a visitor, aminek a segítségével az egész forrásprogramot be tudjuk járni. Fontos tehát, hogy a megfelelő `visit` metódusok logikusan és helyesen legyenek felépítve, nehogy előforduljanak olyan esetek, hogy egy forrásrészt többször is bejárunk, vagy esetleg kihagyjuk.

A vizitálás során ezért a konverter asszociatív tömbökben folyamatosan jegyzi a már felépített node-okat a hozzájuk tartozó egyedi *nodeid*[2] alapján. A csúcsok vizitálása során pedig mindig ellenőrzi, hogy az adott csúcsot már bejárta-e.

Erre azért is van külön szükség, mert a csúcsok konvertálása közben igen gyakran előfordul, hogy „előre dolgozik" a converter a visitorhoz képest. Ha például a visitor egy függvényre „lép rá", akkor a converter az adott függvény alatt lévő teljes részgráfot fel fogja építeni. Ahhoz hiszen, hogy magát a függvény node-ot fel tudjuk építeni, fel kell tudni építeni annak a paramétereit és a törzsét is. Ez pedig csak úgy kivitelezhető, ha a teljes függvénytörzset, azaz a függvény alatt lévő összes utasítást is felépítjük.

Ennek a logikának köszönhetően egyébként magát a *ConverterVisitor* osztályt is egyszerűbben képzelhetjük el, hiszen így bizonyos node típusokat kihagyhatunk a vizitálásból. Nem kell ugyanis a visitor metódust létrehoznunk azokhoz a node-okhoz, amelyekről tudjuk, hogy önmagukban nem fordulhatnak elő a forrásban, csak úgy ha van fölöttük egy szülő node. Ilyen csúcsok tipikusan a *statm* vagy *expr* csomagnak a csúcsai, hiszen egy utasítás szigorúan egy blokkhoz vagy egy függvénytörzshöz kapcsolódhat. Nem „lóghat" a globális scopeban továbbá egy kifejezés sem, a fölött is biztosan van egy másik csúcs az ASG gráfban, aminek a részét képezi majd.

Ezzel viszont egyszerűsít a *ConverterVisitor* osztálynak gyakorlatilag csak a *struc* csomag osztályaihoz kell *visit* metódusokat szolgáltatnia. Az implementációmban jelenleg ezek a csúcsok rendre a következők: *Class*, *ClassTempl*, *Namespace*, *Enumeration*, *Function*, *Object*, *Typedef*.

A *visit* metódusokhoz általában a 3.6. ábrán látható konstrukciót használtam. Amiben még a metódus legelején történik egy ellenőrzés, hogy biztosan nem jártuk-e már be ezt a típust. Ezután a converter dump állományába[3] kerül bejegyzés, hogy hiba esetén könnyebben visszafejthessem, hol történt az elakadás. Ezt követően egy *try-catch* blokkba ágyazva hívom meg a megfelelő segédeljárást (a példa esetében a `build_struc_class ()` eljárást), ami elvégzi az osztály tényleges felépítését, illetve el is helyezi azt a GCC AST-ben. Ha ez az eljárás hibát dob vissza, vagy NULL_TREE esetleg *error_mark_node* értékekkel tér vissza, akkor biztosan tudhatjuk, hogy a konvertálás közben valami hiba merült fel.

A csúcsok átalakítása a segédeljárásokkal

Minden a schema-ban definiált osztályhoz létrehoztam egy *build_csomag_név ()* nevű „segédeljárást". Egy `struc::Object` átalakításához például a *build_struc_object ()* eljárást kell meghívnunk.

[2] A *schema* a minden node őseként értelmezett *base* osztályban definiál egy *id* attribútumot, ami minden ASG csúcsnak egy egyedi azonosítójaként funkcionál.

[3] A dump állomány egy naplófájl szerűség, amibe a program futása közben kerülnek elsősorban a fejlesztők számára hasznos információk.

```
// Visit CLASS
virtual void visit(const columbus::cpp::struc::Class& _node) {
   if (already_visited_type(_node.getId()))
      return;

   convdump.dump("Visit: Class node: " + _node.getName());

   try {
      tree gcc_node = build_struc_class(_fact, &_node)
      if (gcc_node==NULL_TREE || gcc_node == error_mark_node)
         throw ConverterError("Error building class");
   } catch (ConverterError e) {
      e.show();
      exit(-1);
   }
};
```

3.6. ábra. A *struc::Class* visitor metódusa.

Ezek az eljárások felépítik a paraméterként megkapott ASG csúcshoz tartozó AST részgráfot, illetve szükség esetén beszúrják azt a megfelelő helyen az eddig felépített AST-be. Ehhez a beszúráshoz folyamatosan nyomon kell követni az építés közben, hogy hol járunk a forrásfa bejárásánál. Ennek a nyomonkövetéséhez `stack`[4] változókat definiáltam, amelyeknek a tetejére minden csúcs érintésekor beillesztem az új, érintett csúcsot, majd a következő csúcsra való lépés előtt leveszem. Ezzel a módszerrel folyamatosan nyomon lehet követni, hogy melyik függvény melyik blokkjában járunk éppen azaz, hova kell beszúrni az aktuális utasítást.

Ugyanilyen konvenciót követve segédeljárásokat készítettem az absztrakt osztályokhoz is, amik azt a célt szolgálják, hogy mégáltalánosabban hívhassuk meg ezeket az eljárásokat. Az absztrakt osztályokhoz készített segédeljárások a kapott paraméter node típusát megvizsgálják, és attól függően, hogy melyik gyerek típusával egyezik, meghívják a megfelelő átalakító eljárást. Az '=' operátor esetében például a *build_expr_expression ()* meghívja a *build_expr_binary ()* eljárást, ami felismerve, hogy `expr:Assignment` node a paraméter, meghívja a megfelelő *build_expr_assignment ()* eljárást, ami ténylegesen elvégzi az átalakítást.

Ennek a módszernek az igazi előnye az, hogy schema egyes osztályainak kapcsolatait könnyen lehessen az átalakításnál is nyomon követni. A schema szerint ha például egy `if` node feltételéről csak annyit tudunk, hogy az biztosan egy *expression*, akkor azt egyszerűen a *build_exp_expression ()* eljáráshívással alakíthatjuk át. Az pedig már futási időben fog eldőlni, hogy ott melyik *Expression* típus átalakítása fog ténylegesen megtörténni.

A 3.7. ábra a '++' és '−−' operátorokhoz tartozó, `expr::PostIncDec` node típusokat átalakító *build_expr_postincdec ()* eljárást mutatja be. Az ilyen típusú csúcsoknak egyetlen attribútuma a `kind` attribútum, ami azt mondja meg, hogy növeljük, vagy csökkentjük a bal oldalon álló kifejezés értékét. Mivel a schema szerint az egy operandusú `expr::Unary` típusok egy `expr:Expression` node-ot tartalmaznak, ezért a *node->getContains()* eljárással lekérdezett csúcsot a *build_expr_expression ()* függvényhívással konvertálhatjuk GCC formára. Végül pedig a *build_x_unary_op ()* eljárás fogja felépíteni a megfelelő AST részgráfot, ami már a GCC C++ *front end* részének egy eljárása.

[4] stack - verem

```
// i++, i-- operators
tree build_expr_postincdec(Factory* _fact, const expr::PostIncDec* _node) {
    tree gcc_expr=build_expr_expression(_fact,
        (expr::Expression*)_fact->getPtr(_node->getContains()));

    if (_node->getKind() == cpp::base::idk::Inc) {
        return build_x_unary_op(POSTINCREMENT_EXPR, gcc_expr); //i++
    } else {
        return build_x_unary_op(POSTDECREMENT_EXPR, gcc_expr); //i--
    }
}
```

3.7. ábra. Az *expr::PostIncDec* node típushoz tartozó *build_expr_postincdec ()* eljárás

Egyéb fontos részletek

A Converter szerkezete – a Columbus API által rendelkezésre bocsátott Visitor algoritmusnak, és az egyes csúcsok felépítéséhez használt segédeljárásoknál alkalmazott konvencióknak köszönhetően – egyszerű és könnyen átlátható. Akadnak azonban mégis igen összetett node típusok is, amelyek bonyolult nyelvtani szerkezetük miatt a schemaban is bonyolult felépítéssel rendelkeznek. Ezek a csúcsok általában a GCC AST szintjén még-összetettebb struktúrával rendelkeznek, ezért átkonvertálásuk sem olyan triviális, mint a vázolt példákban.

Ilyen összetett adattípus például a *struc* csomagban az osztályhoz tartozó *Class* típus, a függvényhez tartozó *Function* típus, illetve a különféle deklarációkhoz tartozó *Object* típus. Ezeknek a típusoknak a konvertálásához (a korábbiakban írtak mellett) egyéb részletekre is oda kell figyelni, mint például a *forward deklarációk*[5] lekezelése. Az ehhez hasonló részletek a diplomamunkámhoz leadott program forrásában angol nyelvű megjegyzések segítségével vannak dokumentálva.

Egy másik fontos részlete az implementálásnak a Columbus *type* csomagjához kapcsolódik. Ennél a csomagnál ugyanis érdemes külön megjegyezni, hogy mennyire fontos szerepe van a transzformációban, annak ellenére, hogy egy aránylag kevés komponensből álló modul a Columbus Schema-ban. A típusok felépítésének ugyanis rengeteg helyen van kiemelt szerepe a transzformáció során, hiszen gyakorlatilag minden utasításhoz, kifejezéshez és struktúrához is tartozik egy-egy típus. Ezeknek a helyes összeállítása igen fontos. Továbbá a konvertálás futási idejét jelentősen optimalizálja a Columbus Schema által nyújtott lehetőség, hogy ugyanazt a típust mindig ugyanaz a *nodeid* azonosítja, azaz egy objektum készül el hozzá. Ezeket a típus *id*-ket ezért külön asszociatív tömbben gyűjti a Converter a hozzájuk tartozó `tree` struktúrákkal együtt, hogy a transzformálás során is egyszer konvertálódjanak át és később gyorsan, hatékonyan el lehessen érni őket.

3.4. A kiterjesztett forráselemző használata

A fordítóként kiterjesztett forráselemző használatához szükség van a Columbus CAN-PP és CAN alkalmazására, amelyek oktatási és kutatási célokra szabadon letölthetők a Columbus honlapjáról [18] a keretrendszerrel együtt.

[5] A forward deklaráció olyan (eljárás/típus/változó)deklaráció, amihez tartozó definíció a forrásban később kerül kifejtésre.

3.4.1. Telepítés

A GCC forrásába beágyazott *Columbus-GCC Converter* saját *Makefile*-lal rendelkezik, amit a GCC fordítás közben automatikusan meghív ha C++ nyelvi támogatással is fordítjuk. A Converter így automatikusan lefordul, összelinkeli magát, majd a `g++` fordításakor hozzálinkelődik a fordítóhoz.

A Converter telepítéséhez tehát a GCC fordítót kell telepítenünk, aminek forrása a diplomamunkámhoz csatolt CD mellékleten megtalálható, a telepítéshez szükséges instrukciók pedig a manuálban olvashatók [17]. Röviden összegezve a GCC telepítéséhez az alábbi utasításokat kell rendre kiadnunk, hogy forrásból telepíthessük:

```
$ ./configure --enable-languages=c,c++
  --prefix=/ahova/telepitunk --srcdir=/ahol/a/gccforras/van
$ make all-ccc
$ make install-gcc
```

Ezt követően a fordító futtatható binárisát a `--prefix` paraméter segítségével megadott könyvtár `bin` alkönyvtárában találhatjuk.

3.4.2. Futtatás

Egy forrásprogram lefordítása a kiterjesztett GCC-vel több lépcsőben zajlik. Először le kell futtatnunk a fordítani kívánt forrásállományon a CANPP preprocesszort. Majd ennek a kimeneti állományán kell futtatnunk a CAN forráselemzőt. Ha az elemző hiba nélkül lefutott, akkor a generált `.csi` fájlt le tudjuk fordítani a kiterjesztett `g++` fordítónak a megfelelő kapcsolókat átadva:

```
$ CANPP nalattk.c
$ CAN nalattk.i
$ g++ --fnoparse --fcsi-file=nalattk.i.csi nalattk.cpp
$ ./a.out
```

4. fejezet

Eredmények, lehetőségek

A projekt implementálásában sikerült gyakorlatilag a teljes C szintaktikának megfelelő csúcsokat átalakítani a Columbus ASG-ből a GCC AST-re. Emellett az alapvető C++ szintaktikai elemek (osztályok, „try-catch”, „this”, stb.) átalakítására is képes a program. A támogatott nyelvtani elemeket a 4.1. táblázat mutatja be a Columbus Schema csomagok szerinti bontásában.

Csomagnév	Támogatott nyelvi elemek
struc	BaseSpecifier, Class, Namespace, Enumeration, Enumerator, Function, Object, Typedef, Parameter
type	SimpleType, TypeFormer, TypeRep, TypeFormerType, TypeFormerArr, TypeFormerPtr, TypeFormerFunc
statm	Block, TryBlock, Handler, CatchParameter, If, Switch, While, For, Do, Break, Continue, Return, Goto, Empty, CaseLabel, DefaultLabel, IdLabel
expr	ExpressionList, New, FunctionalConversion, Id, Conditional, SizeofType, This, Throw, IntegerLiteral, CharacterLiteral, FloatingLiteral, StringLiteral, BooleanLiteral, FunctionCall, PostIncDec, PreIncDec, TypeidExpr, Indirection, AddressOf, UnaryArithmetic, UnaryLogical, SizeofExpr, Delete, Assignment, ArraySubscription, MemberSelection, Comma, PointerToMember, BinaryArithmetic, BinaryLogical

4.1. táblázat. A Converter által támogatott elemek a Columbus Schema csomagjaiból.

A támogatott nyelvtani elemek elegendőek ahhoz, hogy akár nagyobb C forrású alkalmazásokat is le tudjunk fordítani az ezzel a képességgel kiterjesztett elemzővel. Sikeresen fordítottam le például Debian/GNU Linux rendszeren a *bzip2* tömörítőprogram 1.0.4-es verzióját, amelynek 15 forrásállománya közel 8000 sorával gyakorlatilag minden C szintaktikai elemet mindenféle helyzetben tesztel. Diplomamunkám készítése során emellett közel 120 kisebb (általában 1-1 *node* típushoz készített), saját forrásfájlon verifikáltam az implementációt. Mindemellett teszteltem az alkalmazást a GCC hivatalos Code-Size Benchmark Enviroment (CSiBE) [6, 2] környezetének egyes forrásállományain is. Ez a környezet pedig a GCC-nek egy olyan keretrendszere, amit a GCC (elsősorban kódméret)

optimalizáló algoritmusainak tesztelésére, valamint hatékonyságuk elemzésére készítettek. Több összetett és gyakran használt alkalmazás forrása található benne, mint a zlib, bzip, Linux kernel részei, fordító programok részei, grafikus segédkönyvtárak, stb..

A tesztelés közben lassította a munkát az esetleges hibák nehéz feltárása, hiszen a kiterjesztett fordítási folyamat összetettsége miatt, igen sok helyen csak nehézkes visszafejtési módszerekkel volt esély a hibák lokalizálására. Gyakran pedig az is előfordult, hogy ebben az összetett folyamatban a hibát nem egy rosszul implementált transzformációs lépés okozta, hanem esetleg maga a Columbus *front end* kezelte le rosszul elemzés közben a bemenetet. Szerencsére a fejlesztés közben gyakorlatilag napi szintű kapcsolatot tarthattam a Columbus/CAN fejlesztőivel, aminek köszönhetően közel féltucatnyi *bug* beküldésével egy kicsit én is hozzájárulhattam ennek az összetett forráselemzőnek a biztosabb működéséhez. A kapcsolatnak köszönhetően az ilyen jellegű hibák feltárása is könnyebben és gyorsabban ment végbe.

Az implementálás teljességéhez hiányzik még, hogy a teljes C++ szintaktikai támogatottsága működjön. Ehhez azonban fontos megjegyezni, hogy a Columbus/CAN jelenlegi legfrissebb verziója (amivel én is dolgoztam) hiányosan támogatja a template példányokat. Template példányok kezelése nélkül pedig szinte lehetetlen komolyabb méretű C++ forrást lefordítani, mivel ezek a nyelvi elemek már a *standard könyvtár* olyan fontos header állományaiban is megtalálhatóak, mint az `iostream`.

Amennyiben mind a Columbus/CAN, mind a *Columbus-GCC Converter* alkalmas lesz a teljes C++ szintaktika kezelésére, úgy az így kiterjesztett forráselemzőnek a segítségével újabb, egészen magas szintű elemzéseket, transzformációkat és optimalizálási fázisokat lehet bevinni a GCC-vel összekapcsolt fordítási folyamatba.

A diplomamunkámban alkalmazott módszer első olyan próbálkozásnak minősül, amiben a GCC-t új előrésszel egészítik ki. Ezek a módszerek és technikák pedig tovább általánosíthatóak olyan esetekre is, amelyekben más hatékony *front end* alkalmazásokat (mint az EDG[1] [7]) köthetünk össze a fordítóval. Sőt, a módszer alapján elképzelhető egy olyan általános modul fejlesztése is, ami lehetővé teszi bármilyen *front end* hozzákapcsolását a fordítóhoz.

Diplomamunkámban tehát amellett, hogy egy megoldási módszert és egyben egy megoldást is mutattam a feladatkiírásban kitűzött feladatra, egy széles körben használt forráskód elemző eszközt egészíthettem ki új és hasznos értékekkel. Ez a kiegészítés pedig, további újabb lehetőségek előtti kapukat tár fel.

[1] Az Edison Design Group Inc által fejlesztett EDG, több neves kereskedelmi fordító által is használt *front end*.

5. fejezet

Függelék

5.1. táblázat: Néhány fontosabb TREE_CODE.

Azonosító	Op.	Leírás
ERROR_MARK	0	Hibákat jelző típus.
IDENTIFIER_NODE	0	Azonosítók reprezentálásra szolgáló node típus (DECL_NAME, változók deklarációi, stb.).
TREE_LIST	0	Utasítások listáját tartalmazó típus.
TREE_VEC	0	Utasítások egy tömbjét tartalmazó típus.
BLOCK	0	Block-ot jelölő szimbólum.
INTEGER_CST	0	Egészeket jelölő típus 64 bitnyi adatot tárolására.
REAL_CST	0	Valós típus, az érték a TREE_REAL_CST mezőben van eltárolva.
STRING_CST	0	String típus.
FUNCTION_DECL	0	Függvény deklaráció.
LABEL_DECL	0	Label deklaráció.
FIELD_DECL	0	Field deklaráció.
VAR_DECL	0	Változó deklaráció.
CONST_DECL	0	Konstans deklaráció.
PARM_DECL	0	Paraméter deklaráció.
TYPE_DECL	0	Típus deklaráció.
RESULT_DECL	0	Result deklaráció.
COMPONENT_REF	3	Komponens referencia.
INDIRECT_REF	1	* operátor
ARRAY_REF	4	Tömb indexelés.
PLUS_EXPR	2	Összeadás.
MINUS_EXPR	2	Kivonás.
MULT_EXPR	2	Szorzás.
TRUNC_DIV_EXPR	2	Egész osztás, ami a hányadost lefele kerekíti.
LSHIFT_EXPR	2	Balra léptetés.
RSHIFT_EXPR	2	Jobbra léptetés.
LT_EXPR	2	< operátor.

Folytatás a következő oldalon

táblázat 5.1 – folytatás az előző oldalról

Azonosító	Op.	Leírás
LE_EXPR	2	<= operátor.
GT_EXPR	2	> operátor.
GE_EXPR	2	>= operátor.
EQ_EXPR	2	== operátor.
NE_EXPR	2	!= operátor.
BIT_IOR_EXPR	2	„vagy” bitművelet.
BIT_XOR_EXPR	2	„kizáró vagy” bitművelet.
BIT_AND_EXPR	2	„és” bitművelet.
BIT_NOT_EXPR	1	„nem” bitművelet..
TRUTH_ANDIF_EXPR	2	logikai „és ha” művelet.
TRUTH_ORIF_EXPR	2	logikai „vagy ha” művelet.
TRUTH_AND_EXPR	2	logikai „és” művelet.
TRUTH_OR_EXPR	2	logikai „vagy” művelet.
TRUTH_XOR_EXPR	2	logikai „kizáró vagy” művelet.
TRUTH_NOT_EXPR	1	logikai „nem” művelet.
IF_STMT	3	`if` utasítás.
FOR_STMT	3	`for` utasítás.
WHILE_STMT	4	`while` utasítás.
DO_STMT	2	`do` utasítás.
BREAK_STMT	2	`break` utasítás.
CONTINUE_STMT	0	`continue` utasítás.
SWITCH_STMT	3	`switch` utasítás.

Osztály	Leírás
RTX_CONST_OBJ	konstans reprezentálása (pl. CONST_INT)
RTX_OBJ	objektum reprezentálása (pl. REG, MEM)
RTX_COMPARE	összehasonlítás (pl, LT, GT)
RTX_COMM_COMPARE	kommutatív összehasonlítás (pl. EQ, NE, ORDERED)
RTX_UNARY	1 operandusú aritmetikai kifejezés (pl. NEG, NOT)
RTX_COMM_ARITH	2 operandusú kommutatív művelet (pl PLUS, MULT)
RTX_TERNARY	3 operandusú művelet (IF_THEN_ELSE)
RTX_BIN_ARITH	2 operandusú nem kommutatív művelet (pl. MINUS)
RTX_BITFIELD_OPS	bit-műveletek (pl. ZERO_EXTRACT)
RTX_INSN	INSN, JUMP_INSN, CALL_INSN
RTX_MATCH	MATCH_DUP
RTX_AUTOINC	címzési mód (pl. POST_DEC)
RTX_EXTRA	minden egyéb

5.2. táblázat. Az RTX osztályok.

```
stmt <decl_expr
    arg 0 <var_decl n>>
stmt <decl_expr
    arg 0 <var_decl k>>
stmt <decl_expr
    arg 0 <var_decl nak>>
stmt <decl_expr
    arg 0 <var_decl i>>
stmt <cleanup_point_expr
    arg 0 <expr_stmt
        arg 0 <convert_expr
            arg 0 <modify_expr
                arg 0 <var_decl n> arg 1 <integer_cst 90>>>>>>
stmt <cleanup_point_expr
    arg 0 <expr_stmt
        arg 0 <convert_expr
            arg 0 <modify_expr
                arg 0 <var_decl k> arg 1 <integer_cst 5>>>>>
stmt <if_stmt
    arg 0 <truth_orif_expr
        arg 0 <lt_expr
            arg 0 <var_decl n> arg 1 <var_decl k>>
        arg 1 <lt_expr
            arg 0 <var_decl k>
            arg 1 <integer_cst constant invariant 0>>>
    arg 1 <cleanup_point_expr
        arg 0 <expr_stmt
            arg 0 <convert_expr
                arg 0 <call_expr
                    arg 0 <addr_expr> arg 1 <tree_list >>>>>
    arg 2 <statement_list
        stmt <cleanup_point_expr
            arg 0 <expr_stmt
                arg 0 <convert_expr
                    arg 0 <modify_expr>>>>
        stmt <cleanup_point_expr
            arg 0 <expr_stmt
                arg 0 <convert_expr
                    arg 0 <modify_expr>>>>
        stmt <for_stmt
            arg 1 <le_expr
                arg 0 <var_decl i> arg 1 <var_decl k>>
            arg 2 <cleanup_point_expr
                arg 0 <convert_expr
                    arg 0 <postincrement_expr>>>
            arg 3 <cleanup_point_expr
                arg 0 <expr_stmt
                    arg 0 <convert_expr>>>>
        stmt <cleanup_point_expr
            arg 0 <expr_stmt
                arg 0 <convert_expr
                    arg 0 <call_expr>>>>
stmt <return_expr
    arg 0 <init_expr
        arg 0 <result_decl D.2604> arg 1 <integer_cst 0>>>>
```

5.1. ábra. „n alatt a k” programban a „main” függvény törzsének AST alakja.

```
main ()
{
  int D.1777;
  int D.1778;
  int D.1779;
  int D.1780;
  int n;
  int k;
  int nak;
  int i;

  n = 90;
  k = 5;
  if (n < k)
    {
      goto <D1774>;
    }
  else
    {

    }
  if (k < 0)
    {
      goto <D1774>;
    }
  else
    {
      goto <D1775>;
    }
  <D1774>:;
  printf (&"%d alatt %d nem értelmezett!\n"[0], n, k);
  goto <D1776>;
  <D1775>:;
  nak = 1;
  i = 1;
  goto <D1772>;
  <D1771>:;
  D.1777 = n - i;
  D.1778 = D.1777 + 1;
  D.1779 = D.1778 * nak;
  nak = D.1779 / i;
  i = i + 1;
  <D1772>:;
  if (i <= k)
    {
      goto <D1771>;
    }
  else
    {
      goto <D1773>;
    }
  <D1773>:;
  printf (&"%d alatt %d = %d\n"[0], n, k, nak);
  <D1776>:;
  D.1780 = 0;
  return D.1780;
}
```

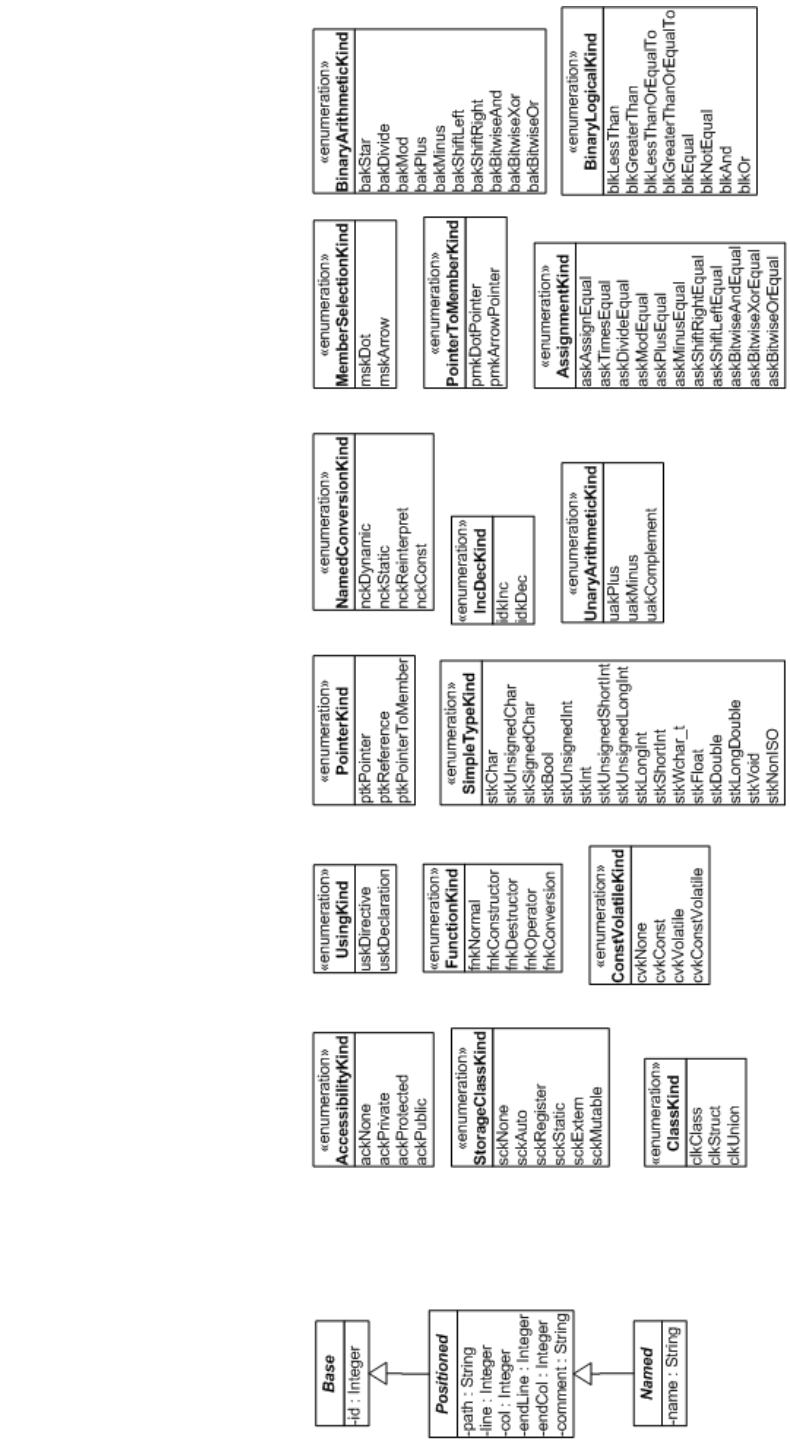
5.2. ábra. „n alatt a k” program GIMPLE alakja.

```
;; Function main (main)

main ()
{
  int i;
  int nak;
  int k;
  int n;
  int D.1780;
  int D.1779;
  int D.1778;
  int D.1777;
  _Bool D.1776;
  _Bool D.1775;
  _Bool D.1774;

<bb 0>:
  n_3 = 90;
  k_4 = 5;
  D.1774_5 = n_3 < k_4;
  D.1775_6 = k_4 < 0;
  D.1776_7 = D.1774_5 || D.1775_6;
  if (D.1776_7) goto <L0>; else goto <L1>;

<L0>:;
  printf (&"%d alatt %d nem értelmezett!\n"[0], n_3, k_4);
  goto <bb 6> (<L5>);

<L1>:;
  nak_10 = 1;
  i_11 = 1;
  goto <bb 4> (<L3>);

<L2>:;
  D.1777_12 = n_3 - i_2;
  D.1778_13 = D.1777_12 + 1;
  D.1779_14 = nak_1 * D.1778_13;
  nak_15 = D.1779_14 / i_2;
  i_16 = i_2 + 1;

  # i_2 = PHI <i_11(2), i_16(3)>;
  # nak_1 = PHI <nak_10(2), nak_15(3)>;
<L3>:;
  if (i_2 <= k_4) goto <L2>; else goto <L4>;

<L4>:;
  printf (&"%d alatt %d = %d\n"[0], n_3, k_4, nak_1);

<L5>:;
  D.1780_8 = 0;
  return D.1780_8;

}
```

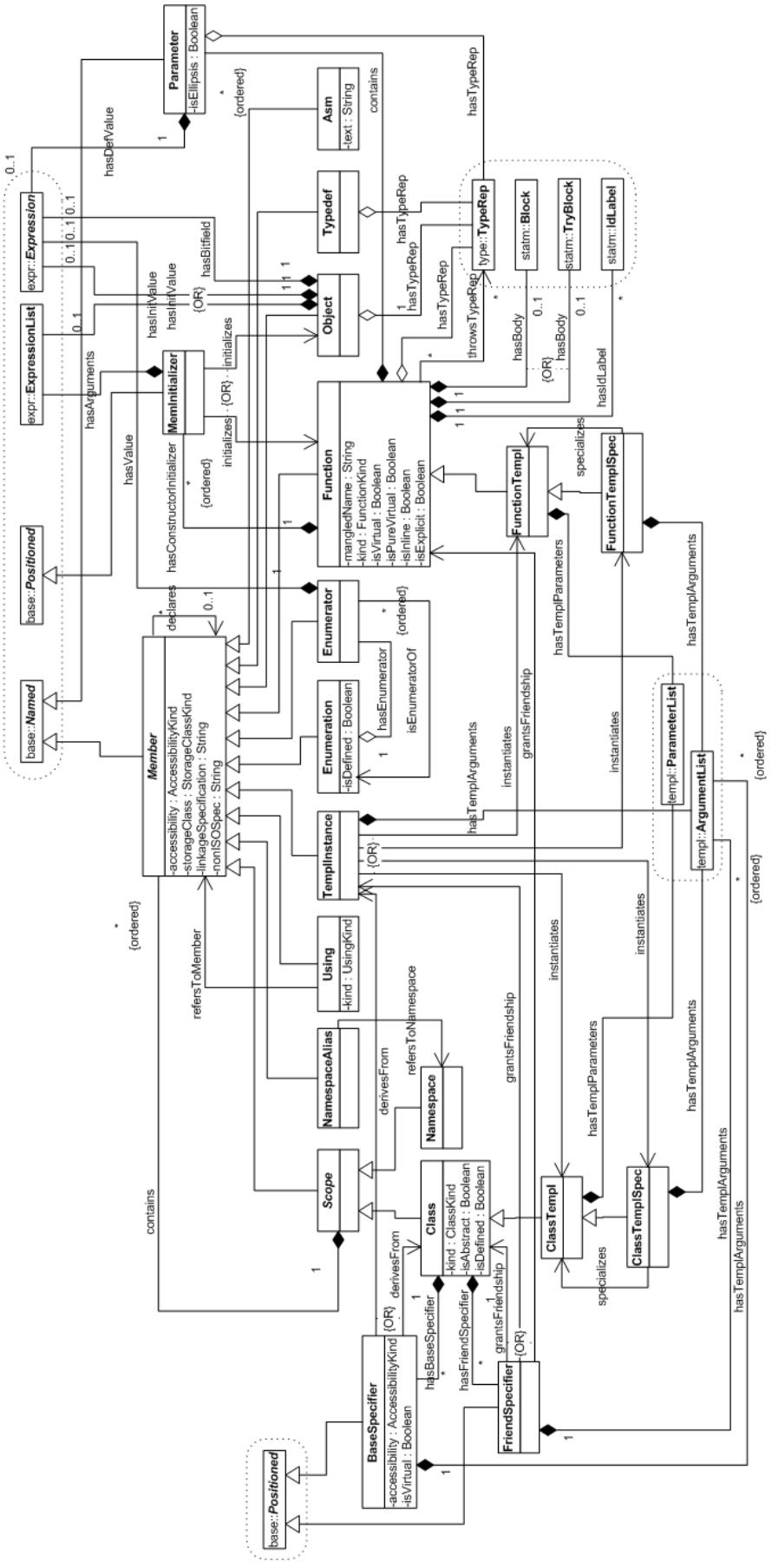
5.3. ábra. „n alatt a k” program *Tree SSA* alakja.

```
;; Start of basic block 2, registers live: (nil)
(code_label 14 13 15 2 2 "" [1 uses])

(note 15 14 17 2 [bb 2] NOTE_INSN_BASIC_BLOCK)

(insn 17 15 18 2 (set (reg:SI 64)
        (const_int 91 [0x5b])) -1 (nil)
    (nil))

(insn 18 17 19 2 (parallel [
            (set (reg:SI 63)
                (minus:SI (reg:SI 64)
                    (reg/v:SI 59 [ i ])))
            (clobber (reg:CC 17 flags))
        ]) -1 (nil)
    (nil))

(insn 19 18 20 2 (parallel [
            (set (reg:SI 65)
                (mult:SI (reg/v:SI 60 [ nak ])
                    (reg:SI 63)))
            (clobber (reg:CC 17 flags))
        ]) -1 (nil)
    (nil))

(insn 20 19 21 2 (parallel [
            (set (reg:SI 66)
                (div:SI (reg:SI 65)
                    (reg/v:SI 59 [ i ])))
            (set (reg:SI 67)
                (mod:SI (reg:SI 65)
                    (reg/v:SI 59 [ i ])))
            (clobber (reg:CC 17 flags))
        ]) -1 (nil)
    (nil))

(insn 21 20 23 2 (set (reg/v:SI 60 [ nak ])
        (reg:SI 66)) -1 (nil)
    (nil))

(insn 23 21 24 2 (parallel [
            (set (reg/v:SI 59 [ i ])
                (plus:SI (reg/v:SI 59 [ i ])
                    (const_int 1 [0x1])))
            (clobber (reg:CC 17 flags))
        ]) -1 (nil)
    (nil))

(insn 24 23 25 2 (set (reg:CCZ 17 flags)
        (compare:CCZ (reg/v:SI 59 [ i ])
            (const_int 6 [0x6]))) -1 (nil)
    (nil))

(jump_insn 25 24 26 2 (set (pc)
        (if_then_else (ne (reg:CCZ 17 flags)
                (const_int 0 [0x0]))
            (label_ref 14)
            (pc))) -1 (nil)
    (expr_list:REG_BR_PROB (const_int 8000 [0x1f40])
        (nil)))
```

5.4. ábra. Részlet az „n alatt a k” program RTL alakjából.



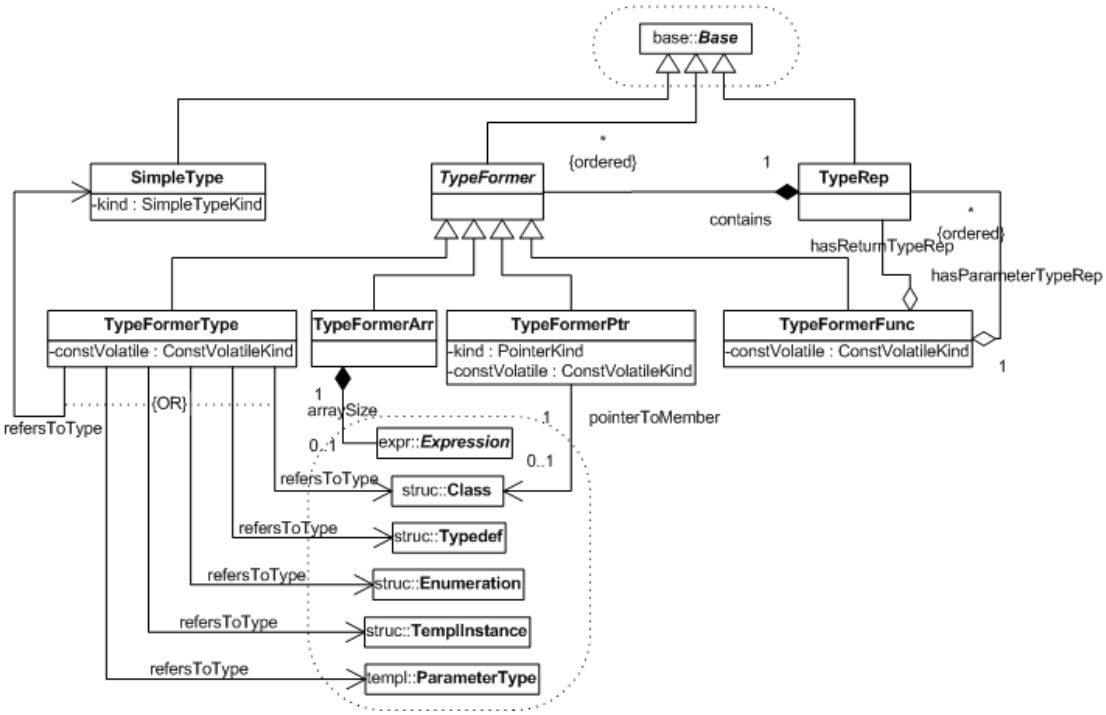
5.5. ábra. A Columbus Schema *base* csomagja.



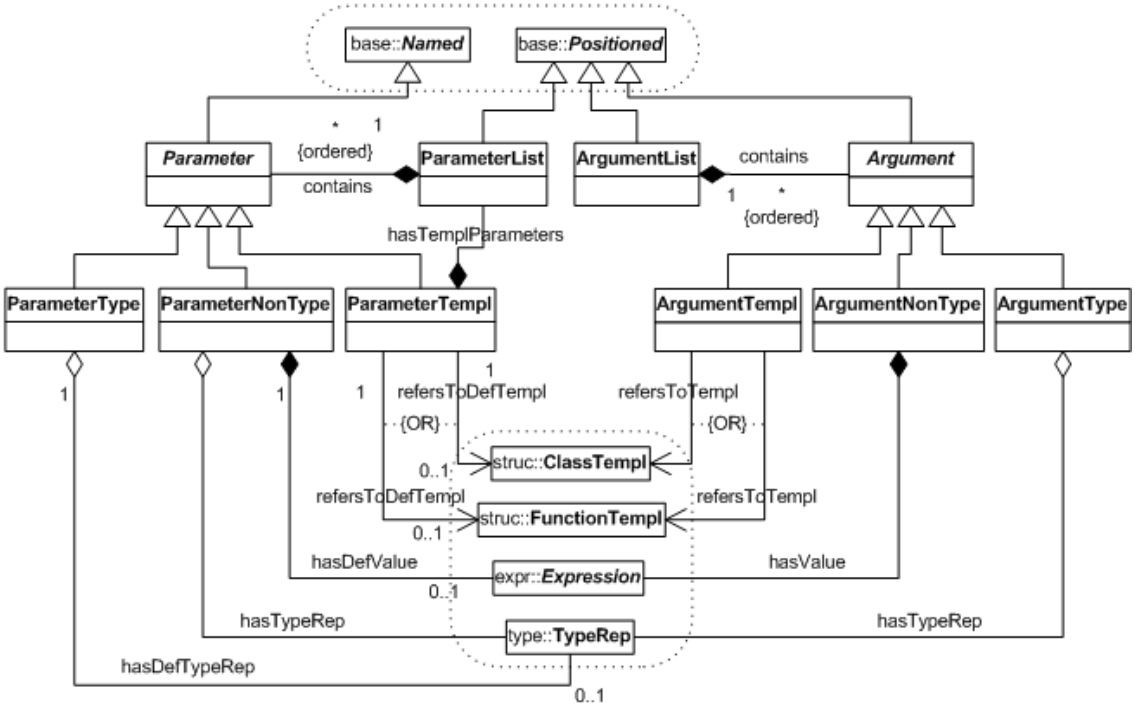
5.6. ábra. A Columbus Schema *struc* csomagja.

machine mode	Leírás
BImode	„Bit” mód, egy bit reprezentálására.
QImode	„Quarter-Integer” mód, egy byte egészként való reprezentálása.
HImode	„Half-Integer” mód, két byte egészként való reprezentálása.
PSImode	„Partial Single Integer„ mód, 4 byteos egész reprezentálása, ami valójában nem használja ki mind a 4 byteot.
SImode	”Single Integer„ mód, 4 byteos egész reprezentálása.
BLKmode	”Block„ mód, a többi módhoz nem illő értékek reprezentálására.
VOIDmode	Void mód, egy meghatározatlan módot jelöl.

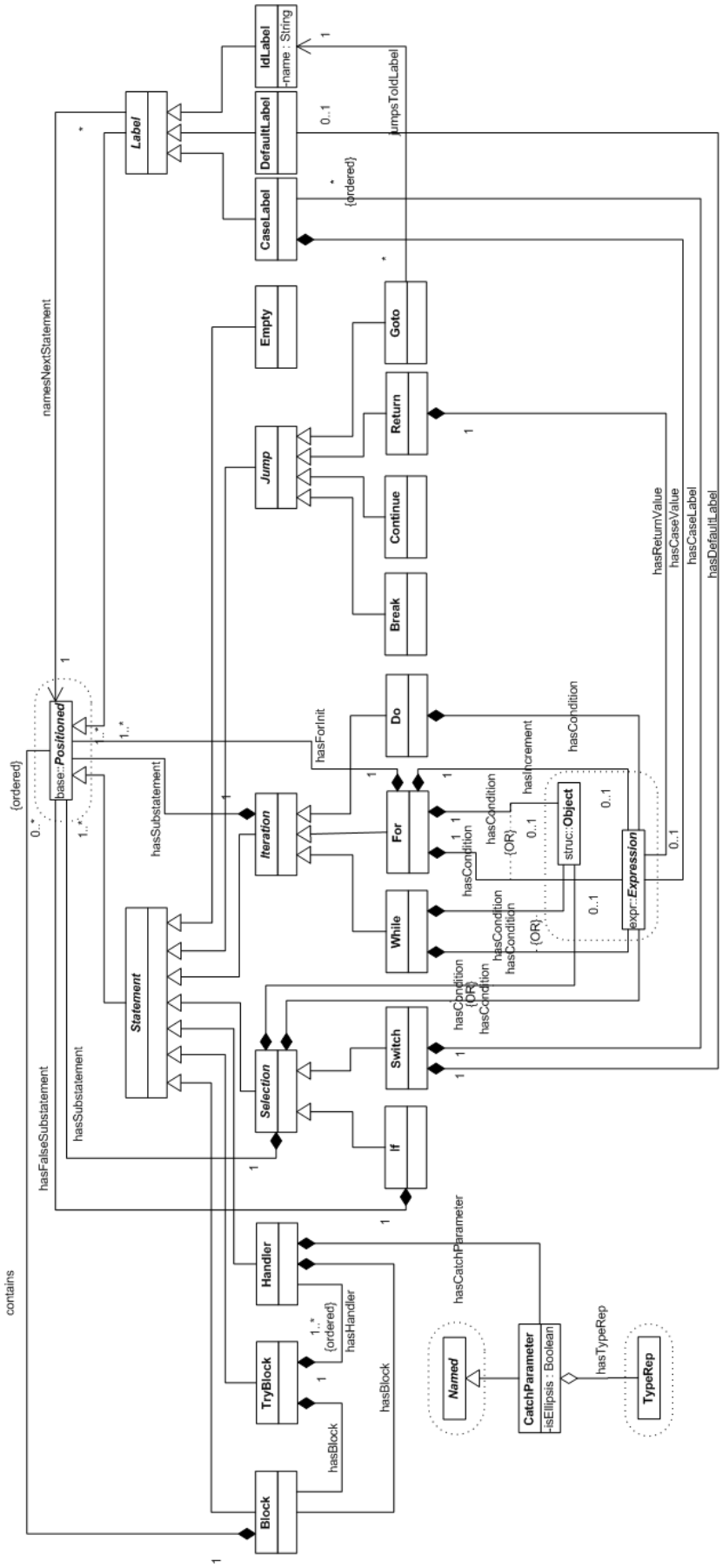
5.3. táblázat. Néhány fontosabb *machine mode*.



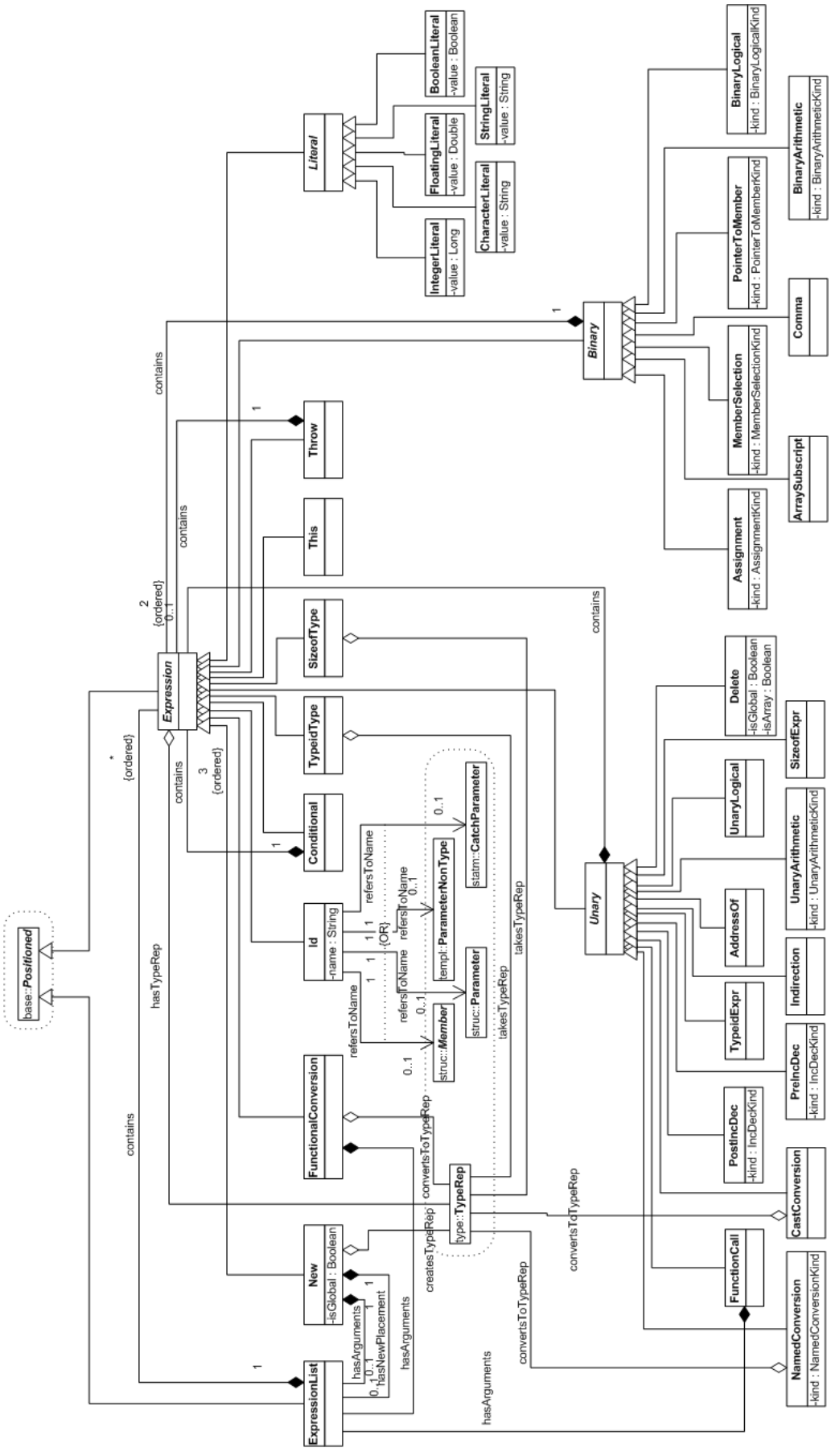
5.7. ábra. A Columbus Schema *type* csomagja.



5.8. ábra. A Columbus Schema *templ* csomagja.



5.9. ábra. A Columbus Schema *statm* csomagja.



5.10. ábra. A Columbus Schema *expr* csomagja.

Nyilatkozat

Alulírott Nagy Csaba, közgazdasági programozó matematikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem Informatikai Tanszékcsoport Szofverfejlesztés Tanszékén készítettem, közgazdasági programozó matematikus diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy diplomamunkámat a Szegedi Tudományegyetem könyvtárában, a kölcsönözhető könyvek között helyezik el.

Szeged, 2007. május 17. .....

aláírás

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek, Dr. Beszédes Árpádnak, diplomamunkám elkészítéséhez nyújtott segítségéért. Külön köszönöm továbbá Lóki Gábor tanácsait és a sok személyes konzultáció lehetőségét, ami dolgozatom elkészítését segítette.

Irodalomjegyzék

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Pearson Education, Inc., 2001. „The Dragon Book".

[2] A. Beszédes, R. Ferenc, T. Gergely, T. Gyimóthy, G. Lóki, and L. Vidács. CSiBE benchmark: One year perspective and plans. In *Proceedings of the 2004 GCC Developers' Summit*, pages 7–15, June 2004.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Algoritmusok*. Műszaki Könyvkiadó, 1997.

[4] B. Cough. *An Introduction to GCC*. Network Theory Limited, 2004.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[6] Department of Software Engineering, University of Szeged. GCC Code-Size Benchmark Environment (CSiBE). `http://www.csibe.org`.

[7] Edison Design Group, Inc. EDG C++ front end. `http://www.edg.com`.

[8] R. Ferenc, A. Beszédes, and T. Gyimóthy. Data exchange with the columbus schema for c++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59 – 66, Mar. 2002.

[9] R. Ferenc, Á. Beszédes, and T. Gyimóthy. Extracting Facts with Columbus from C++ Code. In *Tool Demonstrations of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 4–8. IEEE Computer Society, Mar. 2004.

[10] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, Oct. 2002.

[11] R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss, and M. Tarkiainen. Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST 2001)*, pages 16–27. University of Szeged, June 2001.

[12] R. Ferenc, I. Siket, and T. Gyimóthy. Extracting Facts from Open Source Software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69. IEEE Computer Society, Sept. 2004.

[13] Free Software Foundation. GCC, GNU Compiler Collection. `http://gcc.gnu.org`.

[14] Free Software Foundation. GNU Compiler Collection (GCC) internals. `http://gcc.gnu.org/onlinedocs/gccint`.

[15] Free Software Foundation. GNU Compiler Collection (GCC) internals (Wikibook). `http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals`; accessed May, 2007.

[16] Free Software Foundation. GCC 2.95 release notes, July 1999.

[17] Free Software Foundation. GCC 4.1.2 manual, February 2006.

[18] Front End Art Ltd. Front End Art homepage. `http://www.frontendart.com/`.

[19] Front End Art Ltd. User's guide to Columbus/CAN. Version 3.6.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1998.

[21] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897 – 910, Oct. 2005.

[22] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the mccat compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420. Springer-Verlag, 1992.

[23] R. Holt, A. Schürr, S. E. Sim, and A. Winter. GXL Graph eXchange Language. `http://www.gupro.de/GXL`.

[24] R. Holt, A. Winter, and A. Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of the 27th Working Conference on Reverse Engineering (WCRE2000)*, pages 162–171, November 2000.

[25] J. Hubička. The GCC call graph module, a framework for interprocedural optimization. In *Proceedings of the 2004 GCC Developers' Summit*, pages 65–75, June 2004.

[26] IBM Haifa Research Lab. 1st HiPEAC GCC tutorial: Middle-end and back-end program manipulation. `http://www.hipeac.net/node/745`, May 2006.

[27] IBM Haifa Research Lab. 2nd HiPEAC GCC tutorial: How to and return on experience. `http://www.hipeac.net/node/746`, January 2007.

[28] ISO. *Programming languages*. ISO/IEC 1488c:1998(E) edition, 1998.

[29] S. István and R. Ferenc. Calculating Metrics from Large C++ Programs. In *Proceedings of the 6th International Conference on Applied Informatics (ICAI2004)*, pages 319–328, Jan. 2004.

[30] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[31] J. Merill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179, May 2003.

[32] MinGW. Minimalist GNU for Windows, Homepage. `http://www.mingw.org`.

[33] R. Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, 1998.

[34] D. Novillo. Tree SSA a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, May 2003.

[35] D. Novillo. Design and implementation of Tree SSA. In *Proceedings of the 2004 GCC Developers' Summit*, pages 119–130, June 2004.

[36] D. Novillo. From source to binary: The inner workings of gcc. *Red Hat magazine*, 2, December 2004.

[37] Red Hat, Inc. Cygwin Homepage. `http://www.cygwin.org`.

[38] J. Rumbaugh, I. Jacobson, , and G. Booch. *The Unified Modeling Language - Reference Guide*. Addison-Wesley Publishing Company, 1998.

[39] B. Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, SE-5(3):216– 226, May 1979.

[40] R. M. Stallman. Gnu status. *GNU's Bulletin*, 1(1), February 1986.

[41] L. Vidács, Á. Beszédes, and R. Ferenc. Columbus schema for c/c++ preprocessing. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 75 – 84, Mar 2004.