



Eötvös Loránd Tudományegyetem  
Informatikai Kar  
Programozási Nyelvek és  
Fordítóprogramok Tanszék

# Szemantikus keresés C++ kódbázisban

**Pataki Norbert**  
Adjunktus

**Horváth Gábor**  
Programtervező Informatikus BSc

Budapest, 2014

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>2</b>
<b>2. Felhasználói dokumentáció</b>	<b>3</b>
2.1. Alkalmazás telepítése és beállítása . . . . .	3
2.2. Projekt létrehozása és megnyitása . . . . .	4
2.3. Grafikus felület . . . . .	6
2.4. Lekérdező nyelv . . . . .	11
<b>3. Fejlesztői dokumentáció</b>	<b>14</b>
3.1. Megoldási terv . . . . .	16
3.2. Megvalósítás . . . . .	21
3.2.1. Fogalmak . . . . .	21
3.2.2. Eszközwálasztás . . . . .	21
3.2.3. Szemantikus kiegészítés . . . . .	23
3.2.4. Munkamenet kezelés . . . . .	31
3.2.5. Felhasználói felület . . . . .	33
3.2.6. Fizikai és logikai elrendezés . . . . .	34
3.2.7. Fejlesztői környezet . . . . .	35
3.2.8. Továbbfejlesztési lehetőségek . . . . .	36
3.3. Tesztelés . . . . .	36
<b>4. Összefoglalás</b>	<b>39</b>

# 1. fejezet

## Bevezetés

A szoftverek költségének egyre nagyobb részét teszi ki a meglévő kódok javítása, karbantartása, bővítése. Az ilyen típusú feladatoknak egyik kulcskérdése annak a megtalálása, hogy hol kell átírni a forráskódot. A mai szoftverek esetében lehetetlen feladat szemmel átnézni az egész kódbázist és megtalálni a kapcsolódási pontokat. A programozók gyakran használnak reguláris kifejezéseken alapuló kereséseket a forráskódban való tájékozódáshoz. Ez azonban egy nagyon limitált eszköz csupán, amivel olyan egyszerű kérdésekre sem tudjuk megkapni a választ, mint például egy adott típus összes altípusának a megtalálása. Ezért ezekhez a feladatokhoz mára különböző eszközök fejlődtek ki, amelyek elősegítik a programozók munkáját. Az eszközök hatékonysága nagymértékben függ attól, hogy milyen nyelven megírt kódban próbálunk keresni. Jelenleg a C++ [5] programozási nyelvhez készült hasonló eszközök vagy nagyon limitált tudással rendelkeznek [13] vagy zárt forráskódúak és költséges a beszerzésük [12].

A dolgozat témája egy olyan eszköz, amelyben a C++ forráskódon lehet különböző lekérdezéseket végrehajtani, valamint a találatokat megjeleníteni. Az eszköz a *Clang* [4] fordítóprogramra épül, ami pontos reprezentációval rendelkezik a forráskódról, emellett eszközöket biztosít ennek a lekérdezések elvégzésére. Ezek az eszközök elsősorban a refaktoráló programok és a fordító fejlesztői számára készültek, azonban a dolgozatomban bemutatott alkalmazásban egy felhasználóbarát grafikus felületen keresztül teszem elérhetővé ezt a funkcionalitást. A grafikus felület segít a felhasználónak összeállítani a lekérdezéseket, kezelni a projekteket, szűrni és megjeleníteni a találatokat, valamint tartalmaz egy katalógust a gyakran használt lekérdezésekről. Az alkalmazást *CppQuery* névre kereszteltem el.

## 2. fejezet

# Felhasználói dokumentáció

A *CppQuery* egy olyan alkalmazás, mely segítségével speciális kereséseket, lekérdezéseket végezhetünk C++ kódokon. Olyan kérdéseket is meg lehet válaszolni ezekkel a lekérdezésekkel, amelyekre a szövegszerű keresésekkel nem lehetséges. Ez az alkalmazás C++ programozók számára készült, azzal a céllal, hogy könnyebben tudjanak navigálni és tájékozódni nagy kódbázisokban, segítse az alkalmazások működésének a megértését.

A program a *Clang* fordítót használja a forráskódok elemzésére, valamint a lekérdezések végrehajtására. Mivel a lekérdezéseket a fordító által generált hű reprezentáción végzi a program, ezért a program bemenete nem csupán a forrásfájlokból áll, a fordítási paraméterekre is szüksége van.

### 2.1. Alkalmazás telepítése és beállítása

Az alkalmazás működéséhez szükséges, hogy az alkalmazást futtató gépen telepítve legyen a *Clang* fordító 3.4-es változata. Emellett követelmény még a Qt keretrendszer 5.0-nál frissebb változata. A minimális követelmények a hardverrel szemben nagyon alacsonyak, azonban a memória felhasználás nagyban függ attól, hogy mekkora projekten történik a lekérdezés futtatása. Ezért nagy projektek esetén minél több memóriájú számítógép javasolt. A telepítéshez a `cppquery.tar.gz` tömörített állományt csomagoljuk ki a `tar` alkalmazás segítségével egy tetszőleges helyre. Az alkalmazás telepítése után, az alkalmazás mellett megtalálható egy `settings.ini` nevű állomány. Ezt tetszőleges szövegszerkesztővel megnyitva a `resource-dir` paramétert állítsuk be a Clang erőforrás könyvtárára. Ez a könyvtár Unix rendszerek esetében gyakran a `/usr/lib/clang/3.4/` könyvtár. A beállítások elvégzése után a CppQuery bináris állomány futtatásával elindíthatjuk a programot.

## 2.2. Projekt létrehozása és megnyitása

Az első lépés a szoftver használatában egy úgynevezett fordítási adatbázis előállítás. Ez egy olyan JSON fájl, ami tartalmazza az összes állomány elérési útját, amit elemezni akarunk, a hozzá tartozó fordítási paraméterekkel együtt. Ennek az állománynak az előállítására számos módszer van, amennyiben egyik módszer sem megfelelő ahhoz a projekthez, amin a lekérdezéseket futtatni akarjuk, akkor még mindig van lehetőségünk szkript segítségével vagy akár kézzel előállítani ezt az állományt. A fordítási adatbázis valójában egy JSON tömb, amiben olyan objektumok vannak, amik három attribútummal rendelkeznek. Az első attribútum a `directory`, ami a fordítás során felhasznált munkakönyvtár, a második attribútum a `command`, ami a fordítónak átadott paramétereket tartalmazza, az utolsó attribútum pedig a `file`, ami a fordítandó fájl abszolút elérési útvonalát tartalmazza. Az alábbi példában a fordítási paramétereket és a fájl útvonalának egy részét lecseréltem három darab pontra helytakarékosági okok végett.

```
[
{
  "directory": "/home/xazax/C++Modelling/cppquery-build",
  "command": "clang++...",
  "file": ".../gui/query_widget.cpp"
},
{
  "directory": "/home/xazax/C++Modelling/cppquery-build",
  "command": "clang++...",
  "file": ".../gui/mainwindow.cpp"
},
{
  "directory": "/home/xazax/C++Modelling/cppquery-build",
  "command": "clang++...",
  "file": ".../gui/querycatalog_window.cpp"
}
]
```

Vegyük észre, hogy a fordítási adatbázis csak forrásfájlokat tartalmaz, hiszen csak azok kerülnek fordításra. Amennyiben egy olyan könyvtárral szeretnénk használni az alkalmazást, ami csak headerekből áll, akkor több lehetőségünk is van. A könyvtár nagy valószínűséggel rendelkezik tesztesetekkel. A tesztesetek fordításából kell fordítási adatbázist építeni. Ha egy csak headerekből álló könyvtárnak nincsenek

tesztesetei, egy a könyvtárat használó minél apróbb alkalmazást felhasználhatunk hasonló célra.

Amennyiben a projekt, amin a lekérdezéseket végezni akarjuk, rendelkezik *CMake* build rendszerrel is, akkor ezt a fordítási adatbázist könnyedén előállíthatjuk. Amikor kiadjuk a `cmake` parancsot, akkor a parancssori opciókhoz fűzzük hozzá a következő paramétert.

```
-DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

Ezek után, amikor kiadjuk a `make` parancsot, akkor abban a könyvtárban, ahova a fordítás eredménye is kerül, meg fog jelenni egy `compile_commands.json` nevű fájl. Ez felhasználható az alkalmazás bemeneteként.

Amennyiben a projekt nem rendelkezik *CMake*-kel, akkor még mindig lehetőség van a *Build Ear* eszköz [14] használatára, amit röviden *Bear*nek is szokás hívni. Ennek az alkalmazásnak függőségei a *CMake*, egy ANSI C fordító, a *pkg-config*, a *make* és a *libconfig*. Első lépésként le kell tölteni a git repository-ból az alkalmazást, majd lépünk be a könyvtárba, ahova klónoztuk a repository-t.

```
git clone https://github.com/rizotto/Bear.git Ear
cd Ear
```

Fontos, hogy a repository-t egy *Ear* nevű könyvtárba klónozzuk, ugyanis ellenkező esetben nem fog lefordulni. Ha minden függőség telepítve van, akkor adjuk ki a következő parancsokat.

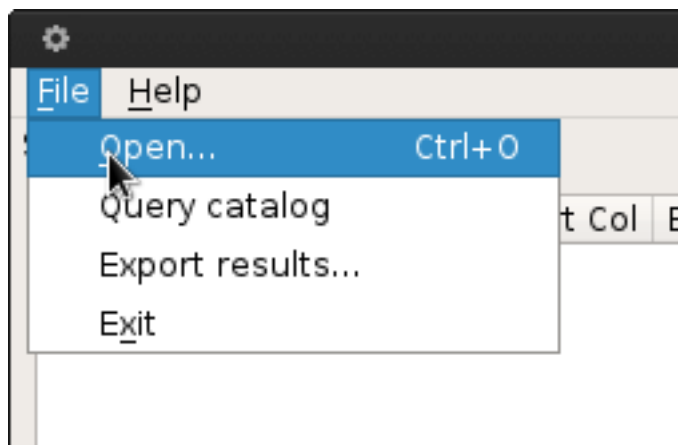
```
mkdir build && cd build
cmake ..
make all
sudo make install
```

Ezek után navigáljunk el annak a projektnek a könyvtárába, amin használni akarjuk az eszközt. Itt tegyük meg a szükséges előkészületeket a `make` parancs kiadásához, például futtassuk le a `configure` scriptet. A `make` kiadása helyett a következő parancsot futtassuk, ahol a `params` helyére helyettesítsük be azokat a paramétereket, amivel a `make` parancsot egyébként is meghívnánk.

```
bear -- make params
```

A `make` parancs lefutása után az aktuális könyvtárban megjelenik egy `compile_commands.json` nevű fájl, amit felhasználhatunk az alkalmazás bemeneteként. Megjegyzem, hogy a fordítási adatbázis előállításának számos előnye van C és C++ projektek esetén, mivel a *CppQuery*-n kívül számos egyéb *Clang* alapokra építő fejlesztői eszköz használja bemeneteként ezeket az állományokat.

## 2.3. Grafikus felület

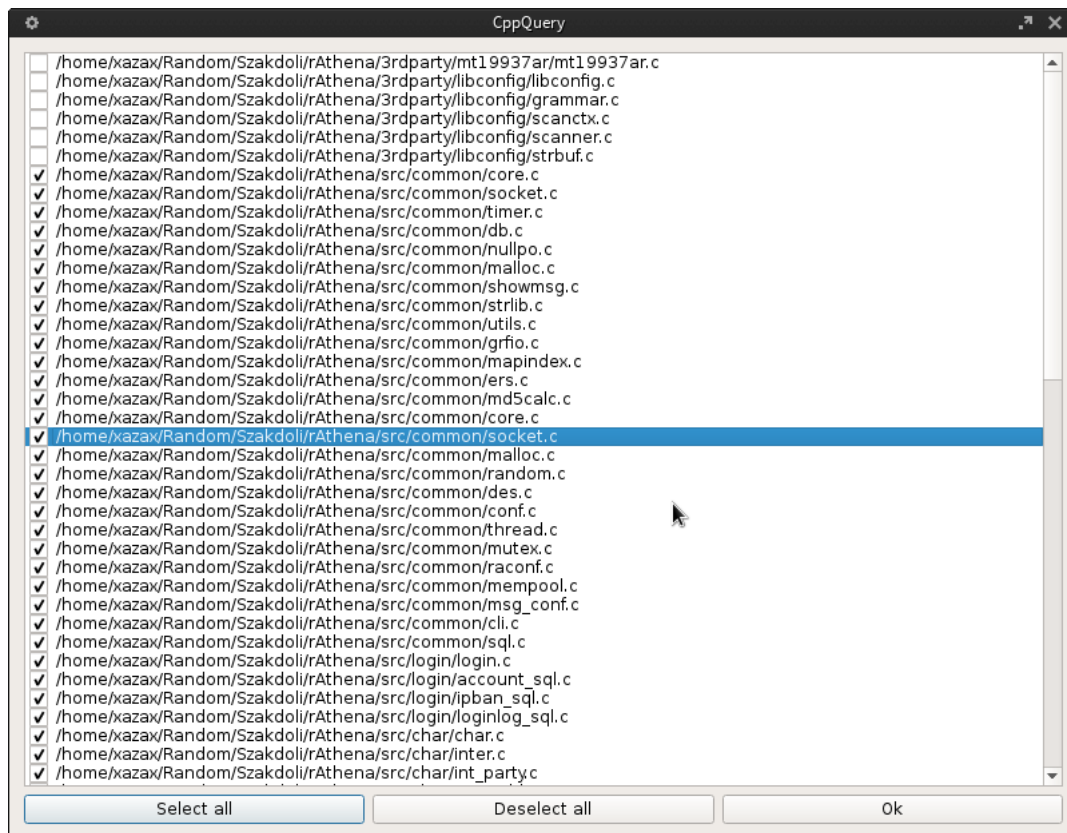


2.1. ábra. Projekt megnyitása

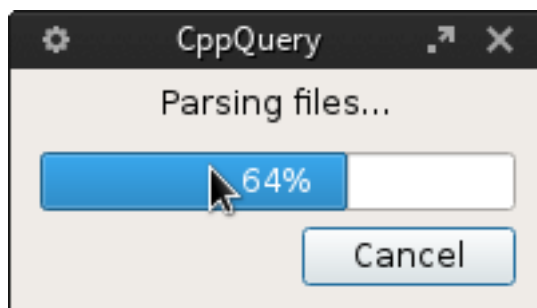
Amennyiben rendelkezünk annak az alkalmazásnak a fordítási adatbázisával, amin a lekérdezéseket futtatni akarjuk, akkor nyissuk meg a **CppQuery** programot. A *File* menüben belül válasszuk ki az *Open...* opciót. Ezt szemlélteti a 2.1 ábra. A standard fájl dialógusban amit kapunk tallózzunk el ahhoz a fordítási adatbázishoz, ami a megnyitni kívánt projekthez tartozik, majd kattintsunk rá az *Open* gombra.

Ennek hatására fel fog ugrani egy ablak azoknak a fájloknak a listájával, amit a fordítási adatbázis tartalmaz. Ezt a listát szemlélteti a 2.2 ábra. Azok a fájlok fognak elemzésre kerülni, amik ebben a listában be vannak pipálva. A lekérdezések csak az elemzett fájlokon futnak le. Abban az esetben, ha nem rendelkezik a gépünk elég memóriával egy nagy projekt elemzéséhez vagy tudjuk, hogy melyik részében fogjuk a projektnek az általunk keresett kódrészletet megtalálni, akkor nagyon hasznos ez a funkció. Alapértelmezetten minden fájl előtt van pipa, tehát az *Ok* gombra való kattintáskor elindul a fájlok elemzése. Amennyiben csak a fájlok töredékét akarjuk elemezni érdemes a *Deselect all* gombbal eltüntetni az összes kijelölést, majd egyesével hozzáadni azokat, amiket elemezni szeretnénk. Mivel minden egyes fájl a listában egy fordítási egységet jelent, ezért minden kiválasztott fájl esetében rekurzív módon az összes fájl által *include*-olt fájl is elemzésre fog kerülni. A kiválasztott fordítási egységeknek Clang által fordíthatónak kell lenniük. Mivel kódgenerálás és szerkesztés nem történik, ezért nem okoz linkelési hibát ha hiányzik néhány definíció, viszont ezesetben ezek a definíciók nem találhatóak meg lekérdezések segítségével sem.

Az *Ok* gomb megnyomása után egy töltő csík fogja mutatni, hogy hogyan áll éppen a fájlok elemzése. Ez látható a 2.3 ábrán.

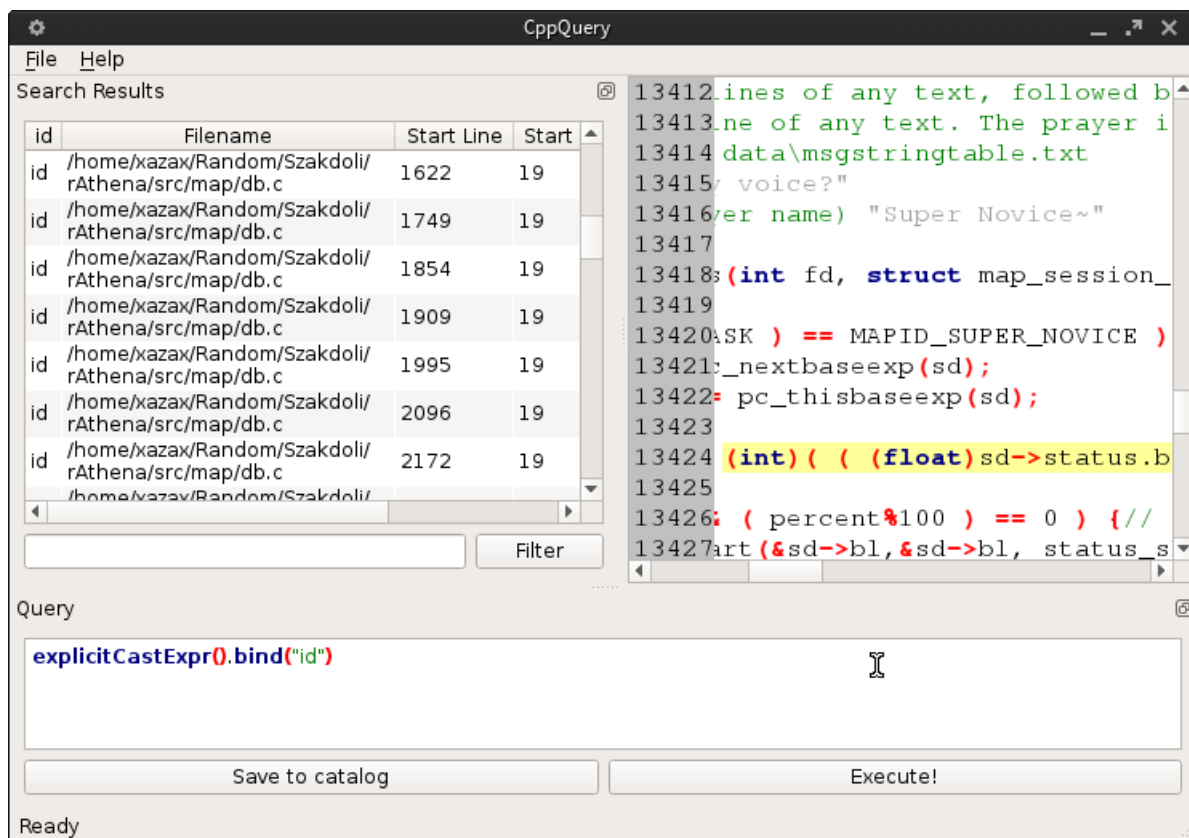


2.2. ábra. Fájl lista



2.3. ábra. Progress bar

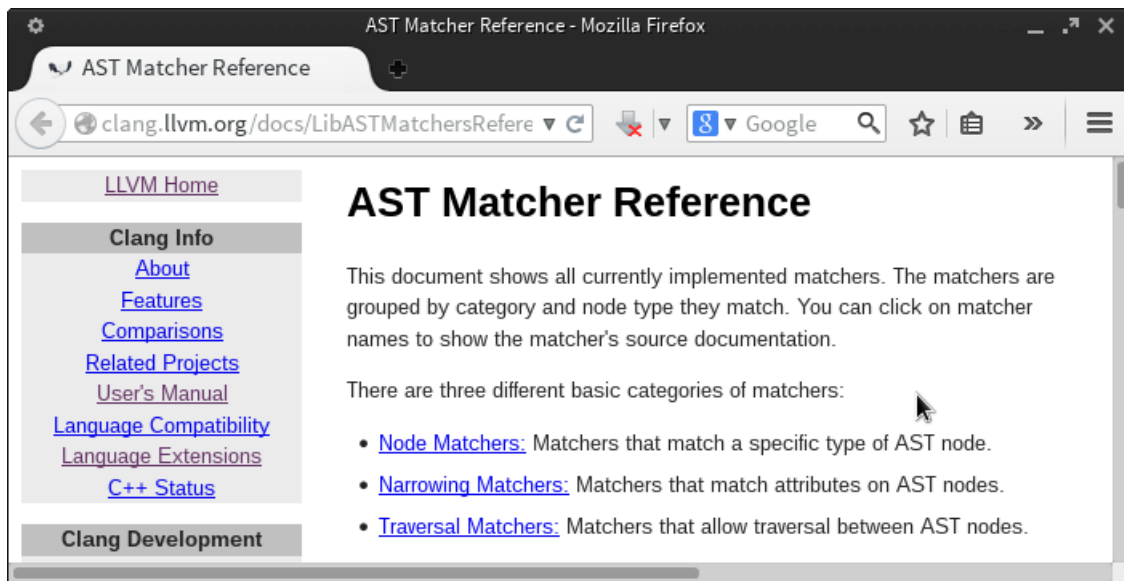




2.4. ábra. Főablak

A főablak a 2.4 ábrán látható. Az ablak alsó részén található a *Query* dokk, ami tetszőlegesen áthelyezhető az ablak más részeire, vagy akár külön ablakba is. Hasonlóan mobilis az ablak bal oldalán található *Search Result* dokk. A *Query* dokkban lévő beviteli mezőbe kell a lekérdezéseket írni. A lekérdezések szerkesztésével kapcsolatban nagy segítséget nyújt a szintaxis kiemelés, valamint a szemantikus kiegészítés. A lekérdezésekkel kapcsolatban további segítség érhető el, ha a *Help* menüben a *Matcher Reference* opcióra kattintunk. Ez böngészőben fogja megnyitni azt a honlapot, ahol megtalálható a lekérdezések alapjául szolgáló elemek dokumentációja. Emiatt a menüpont működéséhez internet hozzáférés szükséges. A lekérdezőnyelv referencia oldaláról a 2.5 ábrán látható egy kép. A *Help* menüben továbbá megtalálható egy *About* menüpont, ami a program rövid leírásul szolgál, valamint egy *About Qt* menüpont, aminek a segítségével megállapítható, hogy a Qt keretrendszer melyik verzióját használták az alkalmazás lefordításához. A leggyakrabban előforduló keresésekre a felhasználó számára rendelkezésre áll egy katalógus, amit a későbbiekben fogok tárgyalni. A lekérdezések megírásával kapcsolatban további információk olvashatóak a 2.4 fejezetben. Miután kész a lekérdezés, az *Execute!* gombra nyomva futtathatjuk le. Ekkor a státusz barban megjelenik a *Querying...* felirat. Amikor

ez a felirat átvált *Ready*-re, a lekérdezés eredményei megjelennek a *Search Results* dokkban.



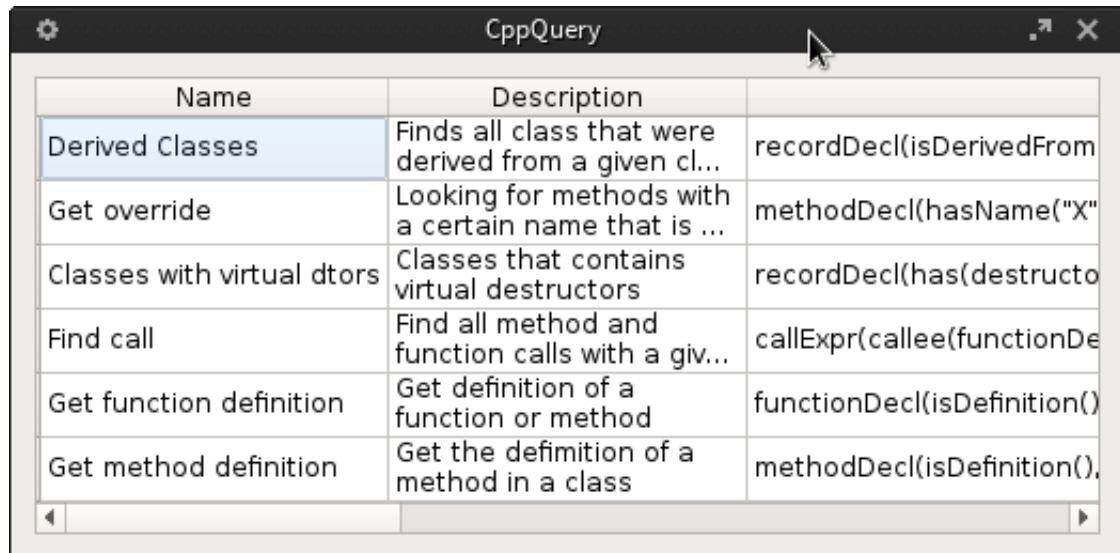
2.5. ábra. Lekérdezőnyelv referencia oldala

Ez a dokk tartalmaz egy táblázatot, benne a találatokkal. Minden találat esetén tartalmazza a találat kezdetének és végének a pozícióját sorszám és oszlopszám formájában. Ezen felül tartalmazza a fájlnevet, amiben a találat van, valamint azt az azonosítót, amivel az adott találat el lett nevezve. Dupla kattintás hatására a kódnézegető részben, ami az ábrán jobb felül látható, meg fog jelenni annak a fájlnek a tartalma, amelyikben a találat volt és arra a pozícióra ugrik a kódnézegető, ahol ez a találat van. Ezen felül az az intervallum, ami a találatot jelzi, halvány sárga kiemelés fog kapni. Amennyiben egy fájlt valamilyen oknál fogva nem sikerült megnyitni, azt az állapot sor jelezni fogja.

A kódnézegető részben nem szerkeszthetőek a fájlok, azonban képes szintaxiskiemelést végezni. Ismeri a C++11 összes kulcsszavát, valamint a Qt moc fordítója által felismert kulcsszavakat is.

Néhány esetben a találatokat nem akarjuk egyből átnézni, vagy esetleg egy másik eszköz számára akarjuk bemenetként használni. Jó példa lehet, hogyha arra vagyunk kíváncsiak, hogy egy adott metódust mennyire bonyolult függvényeken belül hívunk meg. Ez esetben le tudjuk kérdezni azokat a függvényeket, amik meghívják a vizsgált metódust. Ezeknek a függvényeknek a pozícióiból áll majd a találati lista, amit egy szöveges fájlba ki lehet exportálni. Erre a *File* menünek az *Export results...* menüpontját válasszuk. Egy hagyományos fájl mentés dialog segítségével ki tudjuk választani, hogy hova és milyen néven mentjük a találati listát. Ezután ezt a szövegfájlt fel tudjuk dolgozni azon program segítségével, ami a metrikákat számolja

a megtalált függvényekre. Ilyen technikák segítségével alá lehet támasztani vagy éppen megcáfolni, hogy a program mely moduljai szorulnak refaktorálásra, valamint mely metódusok vagy függvények igényelnek alaposabb figyelmet a tesztesetek megírásakor.



The screenshot shows a window titled "CppQuery" with a table containing query information. The table has three columns: "Name", "Description", and a third column containing query code snippets. The "Derived Classes" row is highlighted in blue.

Name	Description	
Derived Classes	Finds all class that were derived from a given cl...	recordDecl(isDerivedFrom
Get override	Looking for methods with a certain name that is ...	methodDecl(hasName("X"
Classes with virtual dtors	Classes that contains virtual destructors	recordDecl(has(destructo
Find call	Find all method and function calls with a giv...	callExpr(callee(functionDe
Get function definition	Get definition of a function or method	functionDecl(isDefinition()
Get method definition	Get the definition of a method in a class	methodDecl(isDefinition(),

2.6. ábra. Lekérdezés katalógus

Vannak olyan lekérdezések, amiket gyakran futtatnak a fejlesztők, csak épp a paraméterek változnak. Ilyen paraméterek lehetnek a lekérdezésben szereplő osztályok és függvények nevei. Ezeket a lekérdezéseket fölösleges lenne újra és újra megírni, ezért egy katalógus szolgál arra, hogy gyorsan, csak a paraméterek kitöltésével lehetőség legyen produktív munkára ezekben az esetekben. Az előbb említett funkcionalitás a *File* menü *Query catalog* menüpontjából érhető el. Ez a katalógus tartalmaz általános lekérdezéseket, amik tovább szerkeszthetők és további feltételek írhatóak beléjük mielőtt a lekérdezés futtatásra kerül. Emellett lehetőség van arra is, hogy további lekérdezéseket adjunk hozzá a katalógushoz, mivel minden projekt esetén felmerülhetnek olyan kérdések, amiket gyakran fel szoktak tenni a fejlesztői, de specifikusak az adott projektre. A katalógus tartalma jó támpontot nyújthat akkor is, amikor még valaki nem ismeri túlzottan a lekérdezésekre használt nyelvet. A katalógus bejegyzései három komponensből állnak. A bejegyzés nevéből, ami röviden utal a funkcionalításra. Egy leírásból, ami tömören, de részletesebben tárgyalja, hogy mit is csinál az adott lekérdezés, valamint a lekérdezés szövegéből. A katalógus tartalma bármikor szerkeszthető, amennyiben egy bejegyzés egyik komponensét kijelöli a felhasználó és gépelésbe kezd, akkor megváltoztathatja annak a tartalmát. Egy minta katalógus megtekinthető a 2.6 ábrán. Amennyiben az egyik bejegyzésre duplán kattintunk, a lekérdezés szövege beillesztésre kerül a 2.4 ábrán látható Query

dokk beviteli mezőjébe. Itt további a paraméterek behelyettesítésén túl további módosításokat végezhetünk a lekérdezésen mielőtt végrehajtjuk azt.

A katalógusba új lekérdezéseket felvenni a *Save to catalog* gomb segítségével lehetséges, amit a 2.4 ábrán figyelhetünk meg. Ekkor megjelenik a katalógus, amihez egy új sor kerül hozzáadásra. A bejegyzés lekérdezés komponense ki lesz töltve pontosan ugyanazzal a szöveggel, ami a lekérdező dokk beviteli mezőjében található. A nevet és a leírást ki kell tölteni. A katalógus perzisztens, tehát minden módosítás amit végzünk rajta, mentésre kerül. Amikor egy másik projekten vagy ugyanazon a projekten később megnyitjuk az alkalmazást, akkor a katalógus minden esetben ugyanabban az állapotban lesz, mint mikor az alkalmazásunkat legutoljára bezártuk.

## 2.4. Lekérdező nyelv

A lekérdezőnyelv a *Clang* fordítónak az *ASTMatcher* [7] könyvtárán alapszik. A könyvtár tartalmaz egy beágyazott nyelvet, amely segítségével mintaillesztést lehet végezni a forráskódok absztrakt szintaxisfáján. Ez elsőre ijesztően hangozhat, de látni fogjuk, hogy mivel a szintaxisfa alapvetően a program szerkezetét tükröző reprezentációja a kódnak, ezért a lekérdezéseket intuitív módon meg lehet alkotni, anélkül, hogy ismernénk a fordítók működését.

A könyvtárban lévő *matcherek* fogják alkotni a lekérdezőnyelv primitíveit, amiket kombinálva összetett lekérdezéseket tudunk alkotni. A könyvtár alapvetően funkcionális megközelítést alkalmaz, ezért a szokásos SQL szerű nyelvektől eltér ugyan a szintaxis, de ennek ellenére ugyanolyan hatékonyan használható. A *matcherek* alapvetően három csoportra lehet bontani. Az első csoportot az úgynevezett *node matcherek* alkotják. Ezeknek a segítségével választhatjuk ki azt, hogy milyen nyelvi konstrukció az, amit keresünk, vagy milyen nyelvi konstrukcióhoz kapcsolható az általunk keresett kódrészlet. Ha változódeklarációkra vagyunk kíváncsiak, akkor például a `varDecl()` matcher segítségével szűkíthetjük ezekre a keresést. Ez felfogható egy fajta szelekciónak. Ezeket a lekérdezéseket később tovább finomíthatjuk a *szűkítő* és *bejáró matcherekkel*, amiket paraméterként adunk át a *node matcherek*-nek. Kényelmi okokból ezek a *node matcherek* tetszőleges számú paramétert fogadnak el. A paraméterek további megszorításoknak tekinthetők, amiknek a konjunkcióját fogja tartalmazni a lekérdezés.

Szűkítő *matcherek* segítségével lehet további megszorításokat adni egy adott *node matcherre*. Lehetséges, hogy csak azok a változódeklarációk érdekelnek, amik egyben definíciók is. Ezt a `varDecl(isDefinition())` matcher kifejezéssel tudjuk leírni. Ezeket a szűkítő feltételeket különböző logikai kombinátorok segítségével

tudjuk bonyolultabb feltételekké formálni. Ilyen kombinátorok az `allOf`, `anyOf`, `anything` és az `unless`.

Az utolsó csoportot a bejáró matcherek alkotják amik segítségével kapcsolatot lehet definiálni különböző nyelvi konstrukciók között. Például ha azokra a változódeklarációkra vagyunk kíváncsiak, amik definíciók és egy függvényhívással vannak inicializálva, az a következőképp adható meg.

```
varDecl(isDefinition(), hasInitializer(callExpr()))
```

Figyeljük meg, hogy az előbbi lekérdezésben több nyelvi konstrukció is szerepet játszik. Egy változódeklaráció és egy függvényhívás. Nem egyértelmű ezért, hogy találatként mit akarunk megjeleníteni. A függvényhívást vagy a változódeklarációt? Jelen esetben a kettő közel van egymáshoz a forráskódban, azonban konstruálható olyan lekérdezés is, ahol a lekérdezésben szereplő nyelvi elemek akár külön forrásfájlokban vannak. Ennek a többértelműségnek a feloldása miatt kötelesek vagyunk a megjelenítendő nyelvi elemet elnevezni egy azonosítóval. Ezt a `bind` metódussal tehetjük meg. Az azonosító teljesen önkényesen választott tetszőleges szöveg, ez meg fog jelenni a találatok között az *Id* oszlopban. Ha a függvényhívást akarom a pozíció helyéül, akkor a következőképp fog kinézni a lekérdezés.

```
varDecl(isDefinition(),
        hasInitializer(callExpr().bind("Call")))
```

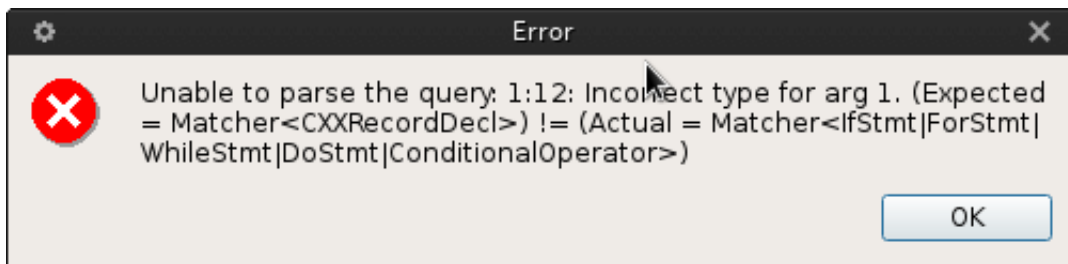
Arra is lehetőség van, hogy egy lekérdezésen belül akár több nyelvi elem pozícióját is belevegyük a találatok listájába. Ebben az esetben több `bind` hívást kell alkalmaznunk.

```
varDecl(isDefinition(),
        hasInitializer(callExpr().bind("Call"))
    ).bind("Decl")
```

A `bind` hívás csak azokon a matchereken értelmes, amik rendelkezhetnek forráskódon belüli pozícióval. Így például az `isDefinition` matcher nem köthető azonosítóhoz, hiszen ez csupán egy predikátum egy nyelvi konstrukcióra, éppen ezért nem rendelkezik egyértelmű fizikai hellyel a forráskódban.

Ha még részletesebb szűrést akarunk végezni a deklarációk között, akár reguláris kifejezés segítségével a deklaráció nevére is tehetünk megszorításokat.

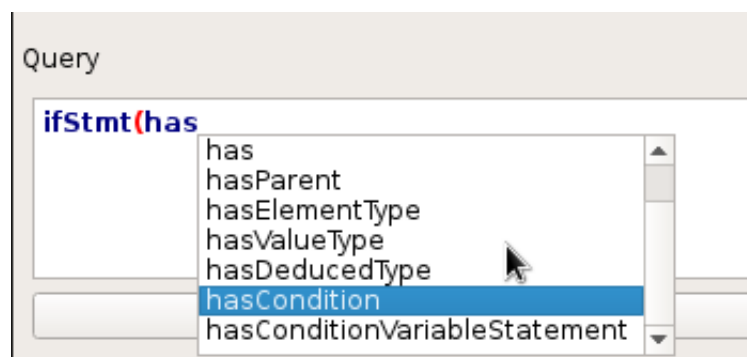
```
varDecl(isDefinition(),
        matchesName("is.*"),
        hasInitializer(callExpr().bind("Call"))
    ).bind("Decl")
```



2.7. ábra. Hibás lekérdezés esetén látható figyelmeztetés

Szerencsére azonban nagy problémát nem lehet okozni egy hibásan megírt lekérdezéssel. Mind a szintaktikus és a szemantikai hibákat észreveszi a program még a lekérdezés futtatása előtt. Amennyiben egy ilyen hiba felmerül a 2.7 ábrán lévőhöz hasonló hibaüzenet jelzi informatív módon, hogy mi a probléma a lekérdezéssel. A helyes lekérdezések írásában nagy segítségünkre lehet a szemantikus kiegészítés, ami a 2.8 ábrán látható.

Ha egy lekérdezés nem tartalmaz egyetlen `bind` hívást sem, akkor attól függetlenül, hogy szemantikailag és szintaktikailag is helyes, találatokat nem produkál. Éppen ezért ebben az esetben is hibát fog jelezni a lekérdezés futtatásának a megkísérlése után a program. Amennyiben még nem nyitottunk meg projektet, a lekérdezéseket nem lehet futtatni, hiszen nem is lenne min. Mivel a projektekben lévő forrásfájlok elemzése *Clang* segítségével történik, a projektnek a fordítója pedig lehet más fordító is, ezért a projekt lehetséges, hogy tartalmaz nem szabványos kódokat, amit valamilyen bővítmény miatt a másik fordító elfogad. Ebben az esetben a forráskód elemzését követően kapunk hasonló hibaüzenetet arról, hogy nem sikerült elemezni a kódot. Ez azért szükséges, mivel a nem szabványos kódot nem tudja jól modellezni a fordító és egy hibás modellen fölösleges lenne lekérdezéseket futtatni, hiszen nem lenne megbízható a kapott eredmény.



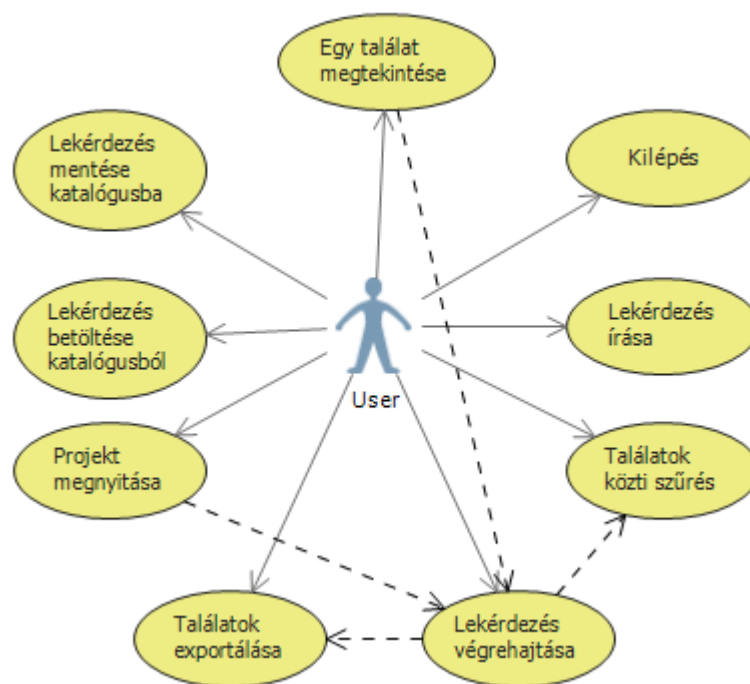
2.8. ábra. Szemantikus kiegészítés

## 3. fejezet

# Fejlesztői dokumentáció

A szövegszerű keresések, akár reguláris kifejezések segítségével nem alkalmasak arra, hogy egy C++ forráskódban hatékonyan tájékozódjon egy fejlesztő. Ennek több oka is van. Többek között a makrók ellehetetlenítik, hogy tisztán C++ forráskódra találjuk ki a mintát, mivel lehetséges, hogy a keresésünk eredménye egy makró kifejtésének a hatására jelenik meg. Emellett ha a keresésünkben szerepel egy típus is, akkor nem elég az adott típust számításba venni, hanem az összes típuszinonimát is figyelemmel kell kísérni, amiknek a létrehozására ráadásul a C++11 szabvány szerint több lehetséges mód is létezik. Több ugyanolyan nevű szimbólum is létezhet különböző névterekben, azonban az a kód ami meghatározza, hogy egy adott szimbólum milyen névtérben található meg, az az adott szimbólum bevezetésének helyétől nagyon távol is lehet. Végül ami a legfontosabb, hogy egy szövegszerű keresés segítségével környezetfüggő lekérdezéseket nem tudunk megfogalmazni, így például már olyan egyszerű kérdésekre sem tudunk hatékonyan választ találni, mint egy adott osztály összes leszármazott osztályának a megtalálása.

A probléma megoldására sok fejlesztői környezet végez kódelemzést, miközben a programozó szerkeszti a forráskódot. Az elemzés során felépített belső reprezentáció segítségével számos kérdést meg tud válaszolni az adott fejlesztői környezet, mint például egy adott szimbólum bevezetésének a helyét. Azonban a C++ nagyon bonyolult nyelvtannal rendelkezik, aminek az elemzése hosszú ideig tart más nyelvekéhez képest. Ebből kifolyólag ezek a fejlesztői környezetek, hogy reszponzívak maradjanak, heurisztikus elemzést végeznek. Az elemzőjük által felépített reprezentáció nem elég pontos összetett kérdések megválaszolásához, sőt, sok esetben még a beépített lekérdezések sem adnak kielégítő eredményt. Sokszor az összetettebb kódokat (például amik metaprogramokat tartalmaznak) még elemezni sem tudják ezek a heurisztikák.



3.1. ábra. Felhasználói esetek

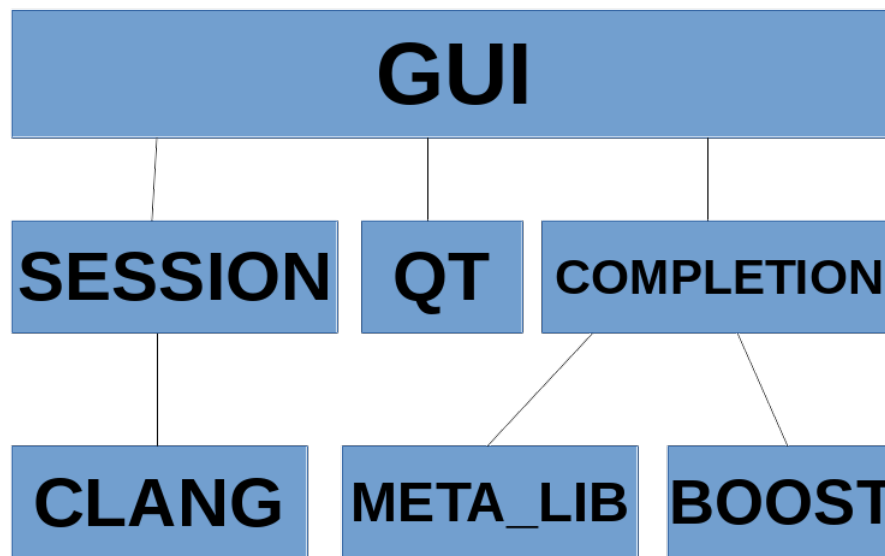
Ebből adódik, hogy ha egy olyan lekérdező szoftvert akarunk készíteni, ami tetszőlegesen bonyolult, de szabványos forráskódra precíz eredményeket képes adni, akkor érdemes egy fordító belső reprezentációját használni. A feladat egy olyan szoftver készítése, ami összetettebb kereséseket tud végezni, mint amire a fejlesztői környezetek és a szövegszerű keresések alkalmasak. Emellett precíz eredményekkel szolgál és minden szabványos forráskódra működik. A lekérdezések megfogalmazása nem egyszerű feladat, ezért az alkalmazásnak segítséget kell nyújtania a felhasználó számára, hogy az minél egyszerűbben elkészíthesse azt a lekérdezést, ami pont az általa feltett kérdést válaszolja meg. A találatok megjelenítését kényelmessé kell tenni, a forráskódon belül a pontos pozíciót meg kell jeleníteni. Mivel egy lekérdezésre nagyon sok találat érkezik, ezért a találatok közötti szűrésre lehetőséget kell adni. Emellett a találatok későbbi feldolgozásának az érdekében a találatok exportálását is lehetővé kell tenni. A felhasználói esetek a 3.1 ábrán láthatóak.

Ezen dokumentációban leírtak átfogó képet hivatottak adni az alkalmazás működéséről és a különböző tervezési döntésekről. Amennyiben az egyes osztályok illetve metódusok részletes dokumentációjára kíváncsi valaki, az angol nyelven elérhető HTML formátumban a CD mellékleten, valamint kigenerálható a `make docs` paranccsal fordítás után.



### 3.1. Megoldási terv

Egy felhasználóbarát grafikus alkalmazás fejlesztése a cél. A komponensmodell sematikusán a 3.2 ábrán látható. A nézetnek megfelelő GUI komponenst a *Qt* keretrendszer segítségével készítem el. Az egyik előnye ennek a döntésnek, hogy több platformon is elérhető alkalmazás készíthető. A másik nagy előny, hogy a C++ nyelven írt felhasználói felület könnyen együttműködik a többi komponens C++-os API-jával.



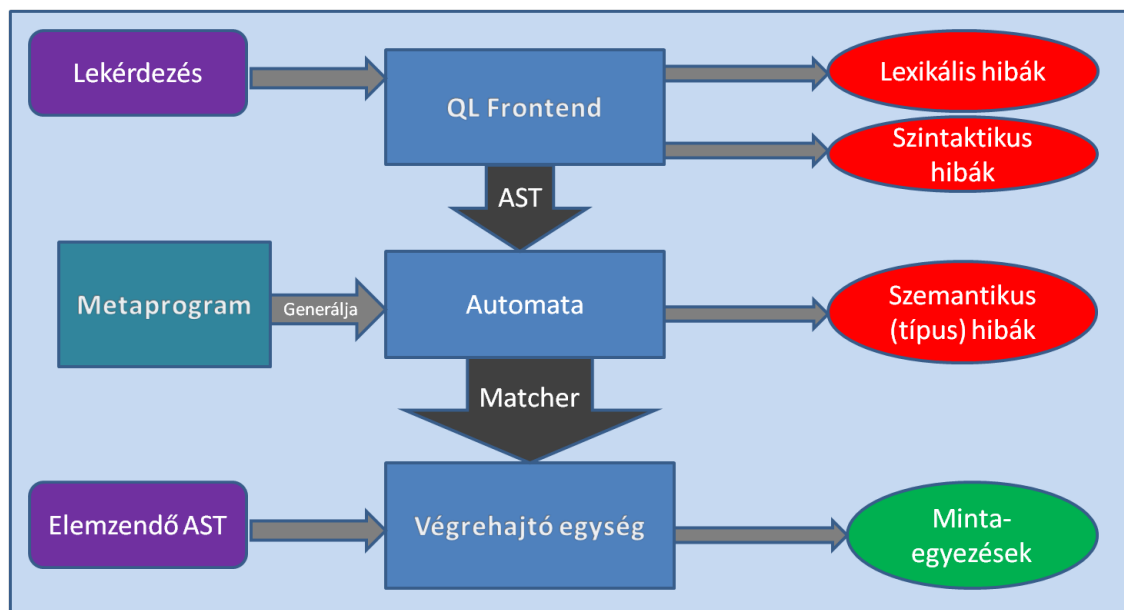
3.2. ábra. Komponensek

A felhasználói felület két nagy komponensre támaszkodik. A *session* komponens fogja számon tartani az összes munkamenetre jellemző információkat, mint például az elemzendő forrásfájlok listája. Mivel ezek az információk szükségesek a lekérdezések végrehajtásához, ezért ez a modul fogja a lekérdezéseket végrehajtani is, valamint a lekérdezések eredményét szolgáltatni a GUI számára. Továbbá ez a modul felelős számos lehetséges hiba kezeléséért, mint például hibás lekérdezés vagy hibás elemzendő kód. A hibajelentést ugyanakkor a GUI komponens fogja megjeleníteni. A *session* a *Clang* fordító által nyújtott lehetőségek segítségével hajtja végre a feladatát. Az eszközválasztást a 3.2 fejezetben részletesebben tárgyalom.

A *completion* modul felelős azokért a kényelmi szolgáltatásokért, amik megkönnyítik a felhasználó számára a lekérdezések létrehozását. Ez elsősorban szemantikus kiegészítést jelent. Ez a modul a szemantikus kiegészítéshez szükséges nyelvtant a lekérdezőnyelv típusinformációiból fogja leszármaztatni metaprogramok segítségével. Ezekhez a metaprogramokhoz egy saját készítésű segédkönyvtárat és a *Boost* könyvtárat fogom felhasználni.

Ahogy az ábrán is látszik egy fontos tervezési szempont volt a függőségek tudatos kezelése. A *Qt* keretrendszer egyedül a felhasználói felület függősége, a *Clang* pedig egyedül a *session* komponensé. A szigorú elkülönítése a függőségeknek a komponensek között nagyobb rugalmasságot tesz lehetővé a jövőben az egyes komponensek módosítására valamint lecserélésére.

A lekérdezések működése absztrakt módon a 3.3 ábrán látható. A *session* komponens felelőssége, hogy a lekérdezés végrehajtó egységet ellássa a szükséges bemenetekkel. Így a *session* menedzseli a forrásfájlok elemzését, a lekérdezések elemzését, valamint a lekérdezés végrehajtását is. Akár az elemzendő forrásfájlok, akár az elemzett lekérdezések tartalmazhatnak hibákat. A hibák felmerülése esetén a *session* fogja a GUI számára eljuttatni a hiba jelentéséhez szükséges összes információt. Sikeres lekérdezések esetén pedig a találati eredményeket fogja a GUI számára elérhetővé tenni.

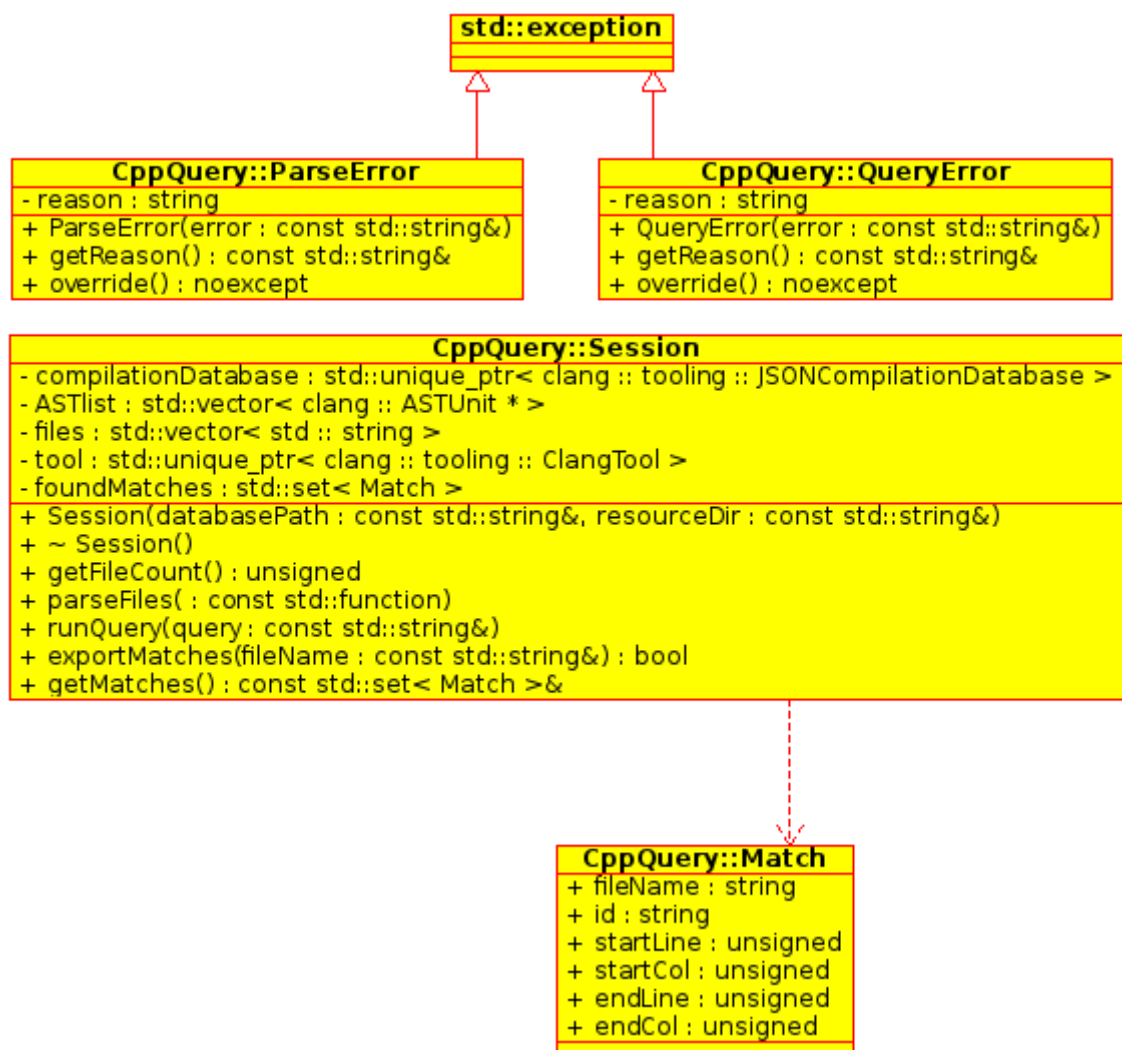


3.3. ábra. Absztrakt működés

A *session* létrehozásáért viszont a GUI lesz felelős minden egyes projekt megnyitásakor. Amikor a felhasználó új projektet nyit meg, olyankor a régi *session* megsemmisítésre kerül. Mivel a felhasználó bármikor nyithat meg új projektet, ezért a *session* minden időpillanatban megsemmisíthető kell, hogy legyen anélkül, hogy ez hibákat okozna a program futásában vagy memóriaszivárgással járna.

A *session* komponens szíve a **Session** osztály lesz, aminek az osztálydiagramja a 3.4 ábrán látható.

A találatokat egy **Match** struktúra fogja tárolni, ami tartalmazza a fájlt, ahol a találat fellelhető, a fájlban belüli pontos pozíciót sor és oszlopszám ábrázolással.



3.4. ábra. Session osztálydiagram

A pozíció az egy intervallum, tehát az intervallum kezdetének és végének külön letárolásra kerül a sor és az oszlop száma. Zárt intervallumok kerülnek letárolásra. Fájlok vagy lekérdezések elemzése közben fellépő hibák esetén kivételeket dobnak a megfelelő metódusai a **Session** osztálynak. Ezek a kivételek rendre a **ParseError** és a **QueryError** példányai.

A grafikus felület tervezésekor a két fő szempont az volt, hogy minél több információt tudjon a felhasználó megjeleníteni a képernyőjén, valamint intuitív működést produkáljon a felület. A felület vázlatos terve a 3.5 ábrán látható.

### Results

Match1	Matchinfo
Match2	Matchinfo
Match3	Matchinfo
Match4	Matchinfo
Match5	Matchinfo
Match6	Matchinfo
Match7	Matchinfo
Match8	Matchinfo
Match9	Matchinfo
Match10	Matchinfo
Match11	Matchinfo
Match12	Matchinfo

### Code

```

template <typename... Ts> struct list {};

template <typename vec, typename... elements> struct push_back;

template <typename... Ts, typename... elements>
struct push_back<list<Ts...>, elements...> {
    using type = list<Ts..., elements...>;
};

template <typename From, template <typename... Ts> class To> struct copy_pack;

template <typename... Ts, template <typename...> class To>
struct copy_pack<list<Ts...>, To> {
    using type = To<Ts...>;
};

template <typename list, template <typename Val> class func> struct map;

template <typename... elems, template <typename Val> class func>
struct map<list<elems...>, func> {
    using type = list<typename func<elems>::type...>;
};

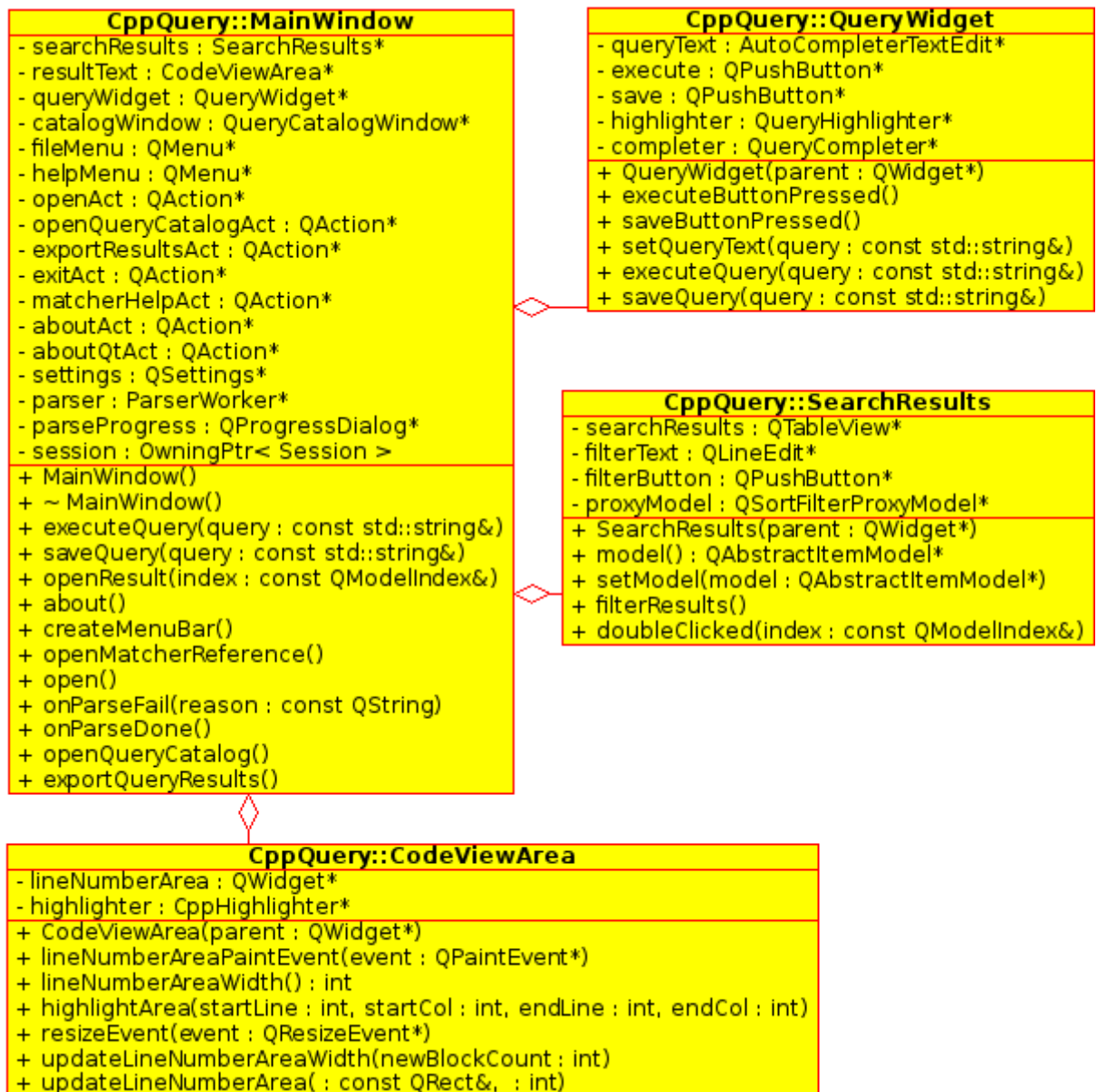
```

### Query

3.5. ábra. A felhasználói felület vázlata

A felhasználói felület három fő részre osztható. A kód nézet, ahol azt a forráskódot láthatja a felhasználó, ahova a lekérdezés találatot adott. Mivel egy lekérdezés eredménye számos találat lehet, ezért az eredmény nézet listaszerűen felsorolva tartalmazza az összes találatot, valamint alapvető információkat a találatokról, ami segít a programozónak a számára megfelelő találat kiválasztásában. Amennyiben a felhasználó duplán kattint az egyik találatra az eredmény nézetben, akkor a kód nézet meg kell, hogy jelenítse azt a forrásfájlt, ahol a találat volt, és a forrásfájlon belül pontosan arra a pozícióra ugrania, ahol a kiválasztott találat megtalálható. Nagy projekteknél várhatóan sok találat lehetséges, amit nem feltétlen akar a felhasználó kézzel feldolgozni. Ennek a megkönnyítésére a találatokra szűrés lehet végezni az fájlok elérési útvonalai alapján. A lekérdezés nézetnél írhatja meg a felhasználó a lekérdezést. Mivel a lekérdezések hossza általában nem túl nagy, ezért ez foglalja el a legkevesebb helyet a felhasználó képernyőjén. Mivel az eddig említett funkciók a leggyakrabban használtak és a legfontosabbak, ezért min-

den további funkciót menükön keresztül lehet csak elérni. Emellett a kis monitorral rendelkezők támogatása végett az eredmény nézet és a lekérdező nézet is dokkolható, így akár külön ablakban is megjeleníthető kell, hogy legyen.



3.6. ábra. GUI osztálydiagram

A felhasználói felület osztálydiagramja a 3.6 ábrán látható. A három fő nézet külön osztályokban van megvalósítva, amik további Qt widgetekre és saját osztályokra vannak visszavezetve. Helytakarékoság végett a további felhasznált osztályokat nem tartalmazza a diagram. A felhasználói felület működésének részletesebb tárgya a fejezetben olvasható.

## 3.2. Megvalósítás

Ebben a fejezetben fogom részletezni az implementáció során alkalmazott megoldásokat. Elsőként bevezetem a szükséges fogalmakat, majd bemutatom a választott eszközöket. Ezt követően a szoftver főbb komponenseit fogom részletesen leírni. Kitérek majd a fájlok fizikai és a modulok logikai szervezésére is. Végül összefoglalom a fejlesztéshez szükséges környezet minimális követelményeit és a program fordításához szükséges lépéseket.

### 3.2.1. Fogalmak

*Elemzésnek* hívjuk azt a folyamatot, amikor egy adott forrásszöveg és egy nyelvtan esetén a szöveget feldolgozva kikövetkeztetjük, hogy milyen levezetési szabályok alapján állítható elő a nyelvtan kezdőszimbólumából a forrásszöveg. Az elemzés eredményeképpen kaphatunk egy *absztrakt szintaxisfát*, ami a forrásszöveg egy számítógépes programok segítségével egyszerűbben feldolgozható reprezentációja. Az absztrakt jelző a nevében arra utal, hogy nem feltétlen jelenik meg minden forráskódon belüli konstrukció a fában csúcsként, lehetséges hogy néhány nyelvi elem implicit módon a fa szerkezetében jelenik meg.

Szakterület-specifikus nyelvnek (Domain Specific Language vagy DSL) [2] nevezzük azokat a nyelveket, amelyek célja egy-egy szakterülethez kapcsolódó problémakör tömör és hatékony megoldására eszközöket biztosítani. Ilyen nyelv például az SQL is, ami lekérdezések megfogalmazására specializálódott. Ezek a nyelvek megjelenhetnek önálló nyelvként is, mint például a HTML, vagy beágyazott nyelvként is. A beágyazott nyelvek egy host nyelvbe vannak beágyazva. Ez azt jelenti, hogy a host nyelven belül bizonyos kódrészleteket a beágyazott nyelv segítségével tudunk megírni. Ilyen beágyazás például az SQL a PL/SQL-ben.

A metaprogramozás lényege, hogy olyan programokat írunk, amik más programokat, esetleg önmagukat manipulálják. Ez a manipuláció történhet akár futási, akár fordítási időben. A C++ nyelvben csak fordítási időben van lehetőség a program manipulációjára, és erre a célra a *template*-eket lehet felhasználni. Ezért szokták a metaprogramozást C++-on belül *template metaprogramozásnak* is hívni. A *template*-ek segítségével történő programozás Turing-teljes.

### 3.2.2. Eszközwálasztás

Ahhoz, hogy egy lekérdezést végrehajtsunk egy forrásszövegen elengedhetetlen a szöveg elemzése. Kézenfekvő megoldás erre a célra egy fordítót választani. Ez a fordító a *CppQuery* projekt esetében a *Clang*. Számos meggyőző indok miatt esett a

választás erre a fordítóra. Az egyik nagy előnye, hogy a *Clang* fejlesztői kezdetektől fogva arra törekedtek, hogy egy moduláris fordítót készítsenek, ami könnyedén felhasználható más projektekben is. Emellett nyílt forráskódú, így igény esetén akár a fordító viselkedésének a módosítására is van lehetőség. A licenc, ami alatt kiadták nagyon liberális, így a projektben szabadon felhasználható. A fejlesztése mögött olyan cégek állnak, mint például a Google és az Apple. A legnyomósabb érv mégis az, hogy a fordító már rendelkezik egy domain specifikus beágyazott nyelvvel, ami segítségével lehet mintaillesztést végezni a szintaxisfákon. Ráadásul képes arra is, hogy futás közben egy ilyen nyelven megírt mintát elemezzen, abból összeállítson egy lekérdezést, és azt futtassa. Azonban ez az eszköz elsősorban a fordító fejlesztői számára készült, hogy az elemzett forrásszöveg szintaxisfájának bizonyos részeit megjelenítsék, valamint azoknak, akik refaktoráló programokat készítenek. Erre a funkcionalitásra alapozva készítettem el az immár nem feltétlen fordítók fejlesztésével foglalkozó programozók számára is használható eszközt.

A *Clang* fordítót alapvetően két feladatra is felhasználom. Az egyik feladat a forrásszövegek elemzése és a szintaxisfa előállítás, a másik feladat a minták illesztése ezeken a szintaxisfákon. A fordító modularitásából fakadóan lehetőségünk van arra, hogy úgy végezzük el a forrásszöveg elemzését, hogy közben ne hozzunk létre tárgykódot. Az általam felhasznált funkcionalitást a fordító teljes egészében csak a C++ API-ján keresztül teszi elérhetővé, ebből kifolyólag a fejlesztést teljes egészében C++ nyelven végeztem.

A szintaxisfán való mintaillesztést a Clang az *ASTMatcher* könyvtáron keresztül teszi elérhetővé. Ez a könyvtár egy beágyazott nyelvet tartalmaz, ami segítségével definiálható a minta. Ez a DSL funkcionális megközelítésen alapszik, ahol predikátumok segítségével tehetünk megszorításokat arra, hogy milyennek kell lennie annak a részfájának a szintaxisfának amit keresünk.

Mivel a fordító kiválasztása után egyértelművé vált, hogy érdemes C++-ban fejleszteni az eszközt, ezért a felhasználói felület tekintetében a *Qt*-re esett a választás. A *Qt* egy több platformon is futó gazdag funkcionalitással rendelkező keretrendszer grafikus alkalmazások fejlesztéséhez. Több ok is szól a döntés mellett. Amellett, hogy több platformot támogat, a legtöbb platformon jól integrálódik az asztali környezetbe a Qt-ban írt alkalmazás. Emellett a natív kódjának köszönhetően jó teljesítménnyel rendelkeznek az ilyen szoftverek. A keretrendszer jó dokumentációval rendelkezik, ezért sokan értenek hozzá.

A szoftver néhány komponense felhasználja a *Boost* könyvtárakat, amik a C++ világban az egyik leggyakrabban használt könyvtárak közé tartoznak. Ezek a könyvtárak számos fordítót támogatnak, és sok olyan funkcionalitást tartalmaz-

nak, amire gyakran szükség van, de a standard C++ könyvtárak nem tartalmazzák. Ezeket a könyvtárakat nem egy konkrét feladatra használtam, hanem arra, hogy sok felmerülő apró, mások által már megoldott feladatot ne kelljen nekem újra megvalósítanom.

A szoftver lefordításához make rendszernek a *CMake*-et választottam. Bár Qt-s projekt esetében magától értetődő lenne a *QtCreator* használata, azonban nem akartam a szoftver esetleges fejlesztőire ráerőltetni egy teljes integrált fejlesztői környezetet. A CMake jó Qt támogatással rendelkezik, és a QtCreator is tud CMake projekteket kezelni. Viszont a forráskód lefordítható anélkül, hogy a QtCreator akár telepítve lenne a fejlesztő gépén. Ennek hála tetszőleges szövegszerkesztővel szerkeszthetővé válik a forráskód.

A kódban található dokumentációk *Doxygen* formátumban íródtak, mivel a C++-t használó projektek körében ez az egyik legelterjedtebb dokumentáció generáló eszköz. Emellett hasznos, hogy képes *markdown* formátumban írt dokumentációból *HTML*-t generálni, és azt hozzácsatolni a forráskódból kinyert dokumentációhoz. A tesztelésre a *Google Test* rendszert választottam. Többek között azért, mivel számos nagy projekt esetében, mint például a *Chrome* böngészőnél is jól bevált.

### 3.2.3. Szemantikus kiegészítés

Annak érdekében, hogy a felhasználó intuitív módon össze tudja állítani a lekérdezést, amit futtatni akar, felhasználóbarát módon kell javaslatokat adni, hogy hogyan folytathatja az adott lekérdezést. Ezt szokták automatikus kiegészítésnek is hívni. Viszont az *ASTMatcher* több mint 200 lekérdező primitívet tartalmaz, ezért a felhasználó érdekében csak azokat a lehetőségeket szabad mutatni, amiknek az adott kontextusban értelmük van. Ezt hívják szemantikus kiegészítésnek. A lekérdezések egy beágyazott nyelv formájában vannak jelen a C++-ban, tehát az a nyelvtan, ami meghatározza, hogy mely környezetben mely primitív lekérdezéseknek van értelme, a típusrendszerben van elkódolva. A C++ fordítási időben limitált lehetőségeket biztosít a típusok vizsgálatára. Ebből kifolyólag egy olyan metaprogram fogja legenerálni az automatikus kiegészítéshez szükséges információkat fordítási időben, aminek a bemenete a beágyazott nyelvet alkotó funktorok lesznek. Az előbb említett metaprogramról írt cikket [3] előadtam az *Informatics 2013* konferencián, valamint behívták az *Acta Electrotechnica Et Informatica* folyóiratba.

Az alapötlet az, hogy minden típusinformáció a fordító rendelkezésére áll, ismeri a konverziós szabályokat. A template specializációkkal történő mintaillesztés és a Substitution Failure Is Not An Error (SFINAE) [6] technika segítségével lehetséges



ezeknek a konverziós szabályoknak a megállapítása és eltárolása fordítási időben. Ezekből a konverziós szabályokból állítható elő az a nyelvtan, amit a szemantikus kiegészítés során fel tudunk használni.

Azért is előnyös a típusinformációkból generálni a nyelvtant, mivel egy kézzel írt nyelvtan hamar elavulna. A Clang verziói között változhat a szintaxisfa szerkezete, valamint új matcherek kerülhetnek hozzáadásra és régi matcherek eltűnhetnek. Azáltal, hogy a nyelvtant a kódból generálom, csökken az az erőfeszítés amit egy új Clang verzióra portoláskor be kell fektetni, valamint kevesebb a hibalehetőség is ilyen esetekben.

Az első nagy kihívás, hogy a szemantikus kiegészítés szövegeken működik, azonban karakterlánc literál nem lehet template paraméter. Ennek ellenére egy ügyes trükk segítségével, amit Sinkovics Ábel fedezett fel, mégiscsak tudunk karakter literálokat kezelni metaprogramokban.

Az ő megoldását viszont módosítanom kellett, mivel jelentős szempont volt, hogy minél egyszerűbben tudjak fordítási idejű karakterláncból futási idejű objektumot generálni. Ábel megközelítésének a lényege az volt, hogy egy makró metaprogram segítségével feldarabolja a karakterláncot karakterek sorozatára, és a fel nem használt karakterek helyére nullákat rak.

```
#define DO(z, n, s) at(s, n),

#define _S(s) \
    BOOST_PP_REPEAT(String::MAX_LENGTH, \
                     DO, s)

template <int N>
constexpr char
    at(char const(&s)[N], int i)
{
    return i >= N ? '\0' : s[i];
}
```

Ábel a Boost MPL [10] metaprogramozáshoz gyakran használt könyvtárban lévő vektor típust használta fel a karakterek tárolására [9] [8]. Én azonban ehelyett a C++11-ben megjelent variadikus (változó paraméterszámú) template-ek paraméter csomagjaiban tároltam a karaktereket. Ez azért előnyös, mert az új inicializációs szintaxis segítségével könnyen és hatékonyan tudok futási idejű karakterláncokat létrehozni.

```
template<unsigned N>
```

```
constexpr unsigned countChars(const char(&str)[N],
                               unsigned i) {
    return (str[i] == 0) ? i : countChars(str, i+1);
}

template <char... cs> struct MetaString {

    static std::string GetRuntimeString() {
        constexpr char contents[] = { cs... };
        constexpr unsigned count = countChars(contents, 0);
        return std::string(contents, count);
    }
};
```

A `countChars` fordítási időben futó függvény meghatározza az utolsó, nem 0 karaktert egy tömbben. A `GetRuntimeString` metóduson visszaadja a `MetaString` típus által reprezentált fordítási idejű szöveg futási idejű változatát.

A következő kérdés, hogy hogyan adjuk meg a bemenetet. Ezt a legkönnyebb úgy megérteni, ha egy konkrét bemenetre írom le, hogy mi történik.

```
typedef typename
    Automata<MATCHER(methodDecl), MATCHER(hasName),
             MATCHER(namedDecl), MATCHER(qualType)
>::result GeneratedAutomata;
```

A metaprogram bemeneteként fel kell sorolni a nyelvet alkotó primitíveket. Ezek a primitívek jelen esetben a `MATCHER` makróba vannak beágyazva. Ez a makró felelős azért, hogy létrehozzon egy három elemű listát, amiben eltárolja az adott objektum típusát, nevét és a példányosítás részleteit. Az utóbbira nincs szükség abban az esetben, ha a metaprogramot szemantikus kiegészítésre használjuk.

```
#define MATCHER(x) variadic::list<                                \
    typename matcher_trait<decltype(x)>::type,                    \
    typename matcher_trait<decltype(x)>::object_type,             \
    MetaString<_S(#x)>                                           \
>
```

A `matcher_trait` metaprogram felelős azért, hogy az adott objektum összetett típusából kikövetkeztesse, hogy mi az a típus, aminek a konverziós szabályait vizsgálni akarjuk a szemantikus kiegészítést elősegítő információk legenerálásakor. Alapvetően a matcherek egy része függvény, még másik részük pedig funktor, ami az

$$M_i = \begin{matrix} & A & B & \dots & \dots \\ \begin{matrix} A \\ B \\ \vdots \\ \vdots \end{matrix} & \begin{pmatrix} true & false & \dots & true \\ false & false & \dots & false \\ \vdots & \vdots & \ddots & \vdots \\ true & true & \dots & true \end{pmatrix} \end{matrix}$$

3.7. ábra. Szemantikus kiegészítés egyik mátrixa

általánosított parancs tervmintát (generalized command pattern [1]) valósítja meg. A `matcher_trait` számos segédmetaprogram segítségével az alapján, hogy az adott matcher függvény-e, valamint ha nem függvény akkor milyen típusú funktor, más és más metafüggvényt hív meg, ami utána elvégzi a tényleges típuskikövetkeztetéseket. A metaprogram bemenete így tehát egy olyan lista lesz, amely három elemű listákat tartalmaz.

A bemenetből generálni fog számos mátrixot a lista tartalma alapján. Mind-egyik mátrix a nyelvi primitívek neveivel van indexelve. Mivel a nyelv, amihez a szemantikus kiegészítést generáljuk funkcionális jellegű, ezért ebben az esetben a nyelvi primitívek mind függvényhívásoknak megfeleltethetők:  $A(\dots, B(\dots), \dots)$ , ahol  $B$  függvény eredményét tovább adjuk  $A$ -nak az  $i$ -edik paramétereként. Legyen az  $i$ -edik mátrix neve  $M_i$ . Az előbbi lekérdezés akkor helyes szemantikusan, ha az  $M_i[A, B]$  értéke igaz. Tehát a  $M_i[A, B]$  hordozza azt az információt, miszerint a  $B$  visszatérési értéke implicit módon konvertálódik-e  $A$ -nak az  $i$ -edik formális paraméterének a típusára. A metaprogram ezeket a mátrixokat fordítási időben generálja, és listában kilapítva sorfolytonos ábrázolásmóddal tárolja. Egy ilyen mátrix látható a 3.7 ábrán. A típuskonverziók helyességét a standard könyvtárbeli `std::is_convertible` trait segítségével állapítom meg. Ahhoz viszont, hogy ezeket a mátrixokat ki lehessen tölteni, minden lehetséges kombinációra meg kell állapítani, hogy létezik-e az adott konverzió. Ehhez fordítási időben a bemenetből le kell generálni az összes lehetséges párosítást.

Az összes párosítás legenerálásának az eredménye, hogy a bemenő nyelvi primitívek számával négyzetesen arányos az előállított párosításokat tartalmazó lista hossza. Ez az `ASTMatcher` könyvtár esetén 230 fölötti szám, tehát a feldolgozandó lista mérete is nagyobb mint 50 000. Ekkora adatmennyiség feldolgozása fordítási időben egyedülálló dolognak számít manapság. A metaprogramok hatékonyságát tapasztalataim szerint leginkább a példányosuló template-ek számában érdemes mérni. Ez nagyban befolyásolja a fordítási időt és a fordításhoz

felhasznált memória mennyiségét is. Azonban jelenleg nem áll rendelkezésre olyan eszköz amivel könnyedén lehetne metaprogramokat profilozni.

Sajnálatos módon az első natív implementációja ennek az algoritmusnak olyanira nem volt hatékony, hogy teljes mértékben lehetetlenné tette a gyakorlati felhasználást. Csupán 12 elemű bemenet esetén a fordító több mint 6 GB memóriát fogyasztott, ráadásul a memóriefogyasztás a bemenet nagyságával exponenciális arányban nőtt. Ennek az egyik oka az volt, hogy a felhasznált technológiákat nem ilyen mértékű bemenetekre és ilyen komplexitású algoritmusokra találták ki.

Az első nagy javulást azzal értem el, hogy a *Boost*ban található *MPL* konténereket leváltottam saját implementációval. Ez azért jelentett nagy javulást, mert a *Boost* konténereket feltölteni és bejárni felépítésükből adódóan csak sok template példányosítás segítségével lehet. Ennek az az oka, hogy a könyvtár egyelőre még nem használja ki a C++11 által nyújtott lehetőségeket, mivel kompatibilis akar maradni a régebbi fordítókkal is.

```
template<typename Head, typename Tail>
struct list {};
```

```
list<double, list<int, list<>>>> foo;
```

Ehhez képest a variadikus template paraméterek segítségével egy sokkal lineárisabb lista konténer is létrehozható, ami segítségével elég egy template példányosítás a lista feltöltéséhez. Egy ilyen metakonténert implementáltam le számos segédfüggvényével együtt. Ez lista implementáció és a hozzá tartozó meta-függvények a *variadic* névtér alatt találhatóak meg.

```
template<typename... Elements>
struct list {};
```

```
list<double, int> foo;
```

További előnye ennek a reprezentációnak, hogy még hagyományos esetben két lista összefűzésénél kénytelenek vagyunk bejárni az egyik listát és egyesével belerakni az elemeket a másik listába, addig az új módszer segítségével konstans template példányosítás segítségével egy teljes összefűzés is elvégezhető. Ezzel a technikával sikerült a második nagy javulást elérni a metaprogram teljesítményében.

```
template <typename... lists> struct concat;

template <typename... Firsts, typename... Seconds>
struct concat<list<Firsts...>, list<Seconds...> > {
```

```

    using type = list<Firsts..., Seconds...>;
};

```

Azonban abban az esetben, ha nem csupán két listát akarunk összefűzni, hanem  $N$  darabot, nincs nyelvi eszköz arra, hogy ezeket az összefűzéseket konstans példányosítással elvégezzük. Ilyenkor a rekurzió miatt valójában  $O(N)$  példányosításra van szükség.

```

template <typename Head, typename... Tail >
struct concat<Head, Tail...> > {
    using type = typename
        concat<Head, typename
            concat<Tail...>::type>::type;
};

```

Habár aszimptotikusan nem csökkenthető a szükséges példányosítások száma, a konstans szorzó tetszőlegesen kicsire csökkenthető, aminek nagyon jelentős teljesítményjavulás lehet az eredménye. Ezt a technikát gyakran alkalmazzák a *Boost* könyvtárban is a várható példányosítások számának a csökkentésére. A technika lényege az, hogy külön megírjuk az összefűzést  $1, 2 \dots k$  darab listára, és csupán a  $k + 1$ . template lesz rekurzív. Ezeket az eseteket meg lehet írni kézzel vagy akár ki is lehet generálni egy makró metaprogram segítségével. Minél nagyobb a  $k$ , annál kisebb konstans szorzóval fog rendelkezni a bemenet függvényében a várható példányosítások száma.

```

template <typename... lists> struct concat;

template <typename... Firsts>
struct concat<list<Firsts...> > {
    using type = list<Firsts...>;

template <typename... Firsts, typename... Seconds>
struct concat<list<Firsts...>, list<Seconds...> > {
    using type = list<Firsts..., Seconds...>;

// Specializations for 3, 4, 5, 6 lists

// Recursive case
template <typename... Firsts, typename... Seconds,
        typename... Thirds, typename... Fourths,

```

```

        typename... Fifths, typename... Sixths,
        typename... Rest>
struct concat<list<Firsts...>, list<Seconds...>,
            list<Thirds...>, list<Fourths...>,
            list<Fifths...>, list<Sixths...>,
            Rest...> {
    using type = typename concat<
        list<Firsts..., Seconds..., Thirds...,
            Fourths..., Fifths..., Sixths...>,
        Rest...>::type;
};

```

Alapvetően mikor egy listán transzformációt végzünk, rekurzívan bejárjuk a listát és egyesével elvégezzük a transzformációt minden elemén. Ez azt jelenti, hogy minden egyes transzformációval csak a bejáráshoz kénytelenek vagyunk felhasználni  $O(n)$  template példányosítást. Azonban a C++11-ben a variadic templatek segítségével képesek vagyunk ezt is konstans példányosítás segítségével megoldani. Ezek után minden meta-algoritmust a mapre próbáltam visszavezetni. Ez jelentette a harmadik nagy javulást a teljesítményben.

```

template <typename list, template <typename Val> class
    func>
struct map;

template <typename... elems, template <typename Val>
    class func>
struct map<list<elems...>, func> {
    using type = list<typename func<elems>::type...>;
};

```

A fentebb leírt módszerek a concat metafüggvény második optimalizációjától eltekintve még nem terjedtek el a metaprogramozók körében, tehát kénytelen voltam ezeket a technikákat magam kitapasztalni. Ezeknek a technikáknak a hatására ugyanaz a meta-algoritmus ami kezdetben 12 hosszú bemenetre 6 GB memóriát használt fel, az most már több mint 230 elemű bement esetén megelégszik 4 GB memóriával. Az összes párosítás algoritmus a fentebb leírt algoritmusokra visszavezetve is igen hosszú leírva, mivel a C++ metaprogramozás egy igen zajos nyelv. Viszont az algoritmust jól kifejező, tömör Haskell kódrészlettel mégis szemléltetni szeretném a mögöttes logikát.

```
pairings x = concat $ map (\a-> map (\b-> (a,b)) x) x
```

Miután kész az összes párosítás és elvégeztük az így kapott listán a szükséges transzformációkat, még mindig kell egy megoldást találni arra, hogy ezeket a fordítási idejű információkat futási időben is elérhetővé tegyünk. Ezért egy factory függvény lesz felelős, aminek a bemenetei template paraméterek lesznek a szokásos futási idejű paraméterek helyett. Mivel azonban a C++ szabvány nem engedi meg template függvények parciális specializációját, ezért ez a függvény egy üres `struct` statikus függvényeként került megvalósításra. A bemenete a függvénynek két csomag azokból a hármasokból, amiket a `MATCHER` makró segítségével tároltam el, valamint egy szám, amit azt írja le, hogy  $F$  által jelölt függvény hányadik paramétereként szerepel a  $G$  által jelölt függvény meghívása.

```
template <typename T> struct RuntimeRuleFactory;

template <typename F, typename FOType, typename FName,
          typename G, typename GOType, typename GName>
struct RuntimeRuleFactory<
    ComposabilityRule<
        variadic::list<F, FOType, FName>,
        variadic::list<G, GOType, GName> > > {

    template <unsigned Param>
    static RuntimeRule GetRuntimeRule() {
        return { FName::GetRuntimeString(),
                 GName::GetRuntimeString(),
                 (ComposabilityHelper<F, G, Param>::value
                  ? ComposeResult::Composable
                  : ComposeResult::NotComposable) };
    }
};
```

A feladat tehát az, hogy az összes párra amit előállított a metaprogram, meghívjuk ezt a factory függvényt, és a függvényhívás eredményét eltároljuk egy futási idejű adatszerkezetben. Az én esetemben a választás az `std::vector`-ra esett. Pontosan ezt csinálja az alábbi kódrészlet, ahol az `IsComposable` egy olyan variadikus paraméter csomag, ami három elemű listák párjait tartalmazza.

```
rules0 = {
```

```

    RuntimeRuleFactory<IsComposable>::template
        GetRuntimeRule<0>() ...
};

```

Itt fog az összes pár kifejtődni egy futási idejű listává, aminek több, mint 50 000 eleme van. Sajnos ilyen méretű inicializáló lista problémát okoz a gcc fordítónak. A 55402-es számú hiba<sup>1</sup> miatt kivárhatatlanul hosszú ideig fordítja a fenti kódot a fordító. Ezt a problémát úgy hidaltam át, hogy a *Clang* fordítót kötelező használni a forráskód lefordításához. Emellett az a fájl, amelyikben a metaprogram példányosul a többitől eltérően minden esetben speciális fordítási paraméterekkel lesz fordítva. Az egyik hozzáadott plusz paraméter letiltja a debug szimbólumok generálását, ami azért fontos mert a metaprogramok eredményeképp számos típus példányosul, amikhez a szimbólumok legyártása szignifikáns memóriaköltséggel és többlet fordítási idővel jár. Ezekből a típusokból nem fog objektum példányosulni, így ez amúgy is fölösleges művelet lenne. Emellett az optimalizációkat is letiltom erre a fájlra, mivel az inline-oló algoritmus is sok memóriát használ fel a metaprogram által generált kód esetében. A fájl nem tartalmaz számításigényes műveleteket, ezért ez az opció nem jelent romlást a felhasználói élményben.

### 3.2.4. Munkamenet kezelés

A munkamenet kezeléséért felelős **Session** osztálynak az osztálydiagramja a 3.4 ábrán látható. Ez az osztály példányosodik minden alkalommal amikor a felhasználó megnyit egy projektet. Ekkor a projektben kiválasztott fájlok elemzésre fognak kerülni. Ez az elemzés a GUI szálról különböző szálon fut és a **Session** objektum vezényli. Ez az objektum a *Clang Tooling* könyvtárát használja, hogy lefuttasson egy **FrontendAction**-t. Ez az akció jelen esetben a **ASTBuilderAction** lesz, aminek az implementációja megtalálható a *session.cpp* fájlban. Ez az osztály tartalmaz egy callbacket ami meghívódik minden egyes fájl esetén, amit elemezni akar a **Session** objektum. Ez a **runInvocation** metódus, ami felelős azért, hogy az adott fájlt elemezze, valamint eltárolja a szintaxisfát, amit az elemzés eredményéül kapott. Minden elemzés előtt és után meghív egy olyan függvényt, amit a **Session** objektum példányosítója állít be, ezáltal kaphat értesítéseket egy fájl elemzéséről, valamint dönthet úgy, hogy egy adott fájl elemzése ne történjen meg. Az elemzés során felmerülhetnek hibák. Ezeket a hibákat egy **TextDiagnosticBuffer** segítségével gyűjti össze az **ASTBuilderAction**. A sikeresen elemzett fájlok szintaxisfái egy vek-

<sup>1</sup>[http://gcc.gnu.org/bugzilla/show\\_bug.cgi?id=55402](http://gcc.gnu.org/bugzilla/show_bug.cgi?id=55402) (2014.05.01.)



torban lesznek eltárolva. Amennyiben volt hiba a fájlok elemzésének során, akkor a `ParseError` kivételt fogja dobni a `Session`.

A lekérdezéseket a `Session` szöveges formátumban kapja meg. Ezeket a szövegeket lekérdezéssé az `ASTMatcher` könyvtár `dynamic` névterében található eszközök segítségével alakítja át. A lekérdezés nem feltétlen helyes, ezért az elemzése közben fellépő hibákról tájékoztatni kell a felhasználót. Ez a `QueryError` kivétel dobásának a segítségével történik meg. A lekérdezés sikeres elemzése után le fogja futtatni az összes korábban eltárolt szintaxisfára az adott lekérdezést, és a találatokat összegyűjti egy halmazba. A találatok összegyűjtéséért a `CollectBoundNodes` osztály felelős. A találatokat `Match` objektumokban tárolom. Ezeken az objektumokon értelmezett a `<` reláció, hogy asszociatív konténerekben is tárolható legyen. A `Session` felhasználója az ilyen `Match` objektumok halmazát kapja meg. Továbbá lehetőség van a legutóbbi lekérdezés taláatait egy szöveges fájlba exportálni.

A felhasználó amikor elemez egy projektet, a projekt elemzéséhez a rendszer headerjeit is kénytelen elemezni a fordító, mert azon headerek hiányában nem tudja értelmezni a forráskódot. Azonban abban az esetben, mikor az egyik lekérdezés helye egy ilyen headeren belül található, ezt a találatot nem akarom a felhasználó felé megjeleníteni, hiszen őt a saját programján belüli találatok érdeklik csak. Ezért a lekérdezés lefuttatásakor abban az esetben, amikor a találatnak a helye egy ilyen rendszerfájlban van, nem kerül eltárolásra az eredmény.

Mivel minden egyes projekt megnyitásakor egy új `Session` objektum jön létre, a régi törlésre kerül. Emiatt a `Session` objektumoknak minden állapotban törölhetőnek kell lenniük anélkül, hogy ezzel a program működésében hibát okoznánk vagy memória szivárogná.

A `Session` példányosítása egy fordítási adatbázis segítségével lehetséges. Ez a fordítási adatbázis egy JSON formátumú fájl, ami leírja, hogy melyik fájl milyen paraméterekkel lett fordítva. Ez azért fontos, mivel egy kód szemantikáját nagyban befolyásolhatják a fordítási paraméterek, hiszen makrókat is lehet definiálni ezeken a paramétereken keresztül. Ilyen fordítási adatbázist tud létrehozni a `CMake`, valamint készíthető is ilyen adatbázis tetszőleges make rendszer esetén a `Bear` eszköz segítségével. Ezeknek az eszközöknek a használatát leírtam a felhasználói dokumentációban. Mivel az eredeti fordítási adatbázis nem feltétlen `Clang` fordító segítségével fordított projektből származik, ezért a fordítási opciókat további Clang specifikus paraméterekkel is ki kell egészíteni. Ezt a célt szolgálja a `ResourceDirSetter` osztály.

### 3.2.5. Felhasználói felület

A felhasználói felület a Qt keretrendszeren alapszik. Az alkalmazás fejlesztése során a Qt5 néven is emlegetett verziót használtam. Többek között azért is, mert új `connect` mechanizmussal rendelkezik, amivel olyan módon lehet összekötni objektumokat, hogy fordítási időben kiderüljön, hogyha nem egyezik a `slot` és a `signal` szignatúrája. Emellett jobban együttműködik a C++11 nyújtotta újdonságokkal, valamint jobb teljesítményt nyújt mint a régi változatok.

A főablakot három fő részre lehet felosztani. A kód nézetre, a lekérdezés nézetre valamint az eredmény nézetre. Ezeknek a nézeteknek és a főablaknak a kapcsolatáról egy osztálydiagram megtekinthető a 3.6 ábrán. A főablak az a `QMainWindow` altípusa. A főablak központi területe a kódnézegető. Az eredmény nézet és a lekérdezés nézet egy-egy `QDockWidget` segítségével lett a főablakra rakva. Ennek hála a felhasználói felület könnyen személyre szabható, valamint külön ablakba kiemelehetőek ezek a komponensek, ha kevés hely áll a programozó rendelkezésére.

A kódnézegető a `QPlainTextEdit` egy specializációja. Többek között hozzáadásra került egy oldalsáv, ami a sorszámokat jeleníti meg a kód mellett. Emellett a `CppHighlighter` osztály segítségével, ami a `QSyntaxHighlighter` osztály specializációja szintaxis kiemelést is végez a kódnézegető. Továbbá képes a találatok megjelenítésére, azaz a szövegben egy bizonyos tartományon további kiemeléseket tud végezni.

A lekérdezés eredményei egy `QTableView` widgetben fognak megjelenni, aminek az elemei nem szerkeszthetőek, de duplán kattinthatóak, aminek a hatására az adott forrásfájl megnyílik a kódnézegetőben és a megfelelő intervallum kijelölésre kerül. A táblában lévő adatok egy külön model objektumban tárolódnak. Ahhoz, hogy a felhasználó szűréseket tudjon végezni az adatokon egy köztes modelt kellett beiktatni a tábla adatait tartalmazó model és a nézet közé, ezt a feladatot `QSortFilterProxyModel` végzi.

A lekérdezést magát szintén egy `QPlainTextEdit`-ből specializált osztály végzi. Hasonlóan a kódnézegetőhöz ez az osztály is végez szintaxiskiemelést a `QueryHighlighter`-nek hála. A legbonyolultabb feladatot a szemantikus kiegészítés jelenti. Ezt a `QueryCompleter` osztály végzi, ami a `QCompleter` osztály specializációja. Ez az osztály az, ami környezetfüggő módon meghatározza, hogy mely függvény hányadik paraméterét szerkeszti éppen a felhasználó. Ennek függvényében meghívja az annak a metaprogramnak az eredményéül kapott típus egy objektumát, amiről az előző fejezetben szó volt. A kapott eredmény alapján frissíti a modelt ami alapján a kiegészítést végzi a `QCompleter` osztály.

A lekérdezéseket lehetőség van elmenteni későbbi használatra egy katalógusba, valamint ebből a katalógusból felhasználható régebben elmentett lekérdezés. A `QueryCatalogWindow` felelős ennek a katalógusnak a megjelenítéséért valamint menedzseléséért. Ahogy az osztály neve is mutatja ez a katalógus egy külön ablakként jelenik meg.

A Qt keretrendszerben az eseménykezelő a fő szálon fut. Ebből kifolyólag amikor egy időigényes műveletet hajtunk végre a fő szálon, akkor az eseménykezelőre nem jut a végrehajtás, ezért a felhasználói felület nem fog reagálni a bemenetekre. Emiatt az időigényes műveleteket külön szálon érdemes futtatni. Ilyen művelet a forrásfájlok elemzése is. Ahhoz, hogy egy progress bar tudjon a GUI mutatni, miközben egy másik művelet fut, fontos volt, hogy a másik művelet egy külön szála kerüljön. Ebből adódóan egy `QThread` objektum által menedzselt szálon történik a forrásfájlok elemzése. A szálak közti kommunikáció sorok segítségével történik.

A fájlok megnyitásához a Qt-ban megtalálható, előre megírt dialogokat használom. Egy projekt megnyitásához a projekt fordítási adatbázisát tartalmazó JSON fájlra kell rátallózni. Ennek a fájlnek a megnyitása után felugrik a `FileSelectorDialog` ablak, ahol az adatbázis által tartalmazott fájlok vannak felsorolva. Ezekből a fájlokból kell kiválasztania a felhasználónak, hogy melyek azok, amiket ténylegesen elemezni szeretne.

A felhasználói felület fejlesztése során minden szöveget a `tr` makróval vettem körbe. Ez azért előnyös, mert bár a felhasználói felület nyelve jelenleg angol, ugyanakkor a későbbiekben ha lefordításra kerülne más nyelvekre, akkor minimális erőfeszítés árán lehet majd többnyelvűvé tenni az alkalmazást, hála a Qt ilyen célra készített eszközeinek.

### 3.2.6. Fizikai és logikai elrendezés

A `Session` és `GUI` komponensek osztályai mind a `CppQuery` névtérben találhatóak. Emellett a lokálisan használt típusok anonim névtérben vannak. A metaprogramhoz felhasznált variadikus metakonténerek és a hozzá tartozó metafüggvények mind a `variadic` névtérben találhatóak. A szemantikus kiegészítést generáló metaprogram nincs névtérben, csupán egyetlen típust tartalmaz, ami felhasználásra kerül belőle. Ez a típus ráadásul egyetlen forrásfájlban kerül felhasználásra, ahol objektum példányosodik belőle. Ez a példányosítás szintén egy anonim névtéren belül történik meg.

A projekt gyökérkönyvtárában számos mappa és fájl található. A `documentation` mappában találhatóak azok a markdown fájlok, amiket felhasználva a Doxygen ki-generálja a dokumentációkat. Ennek a mappának az almappáit rekurzívan bejárja a

Doxygen dokumentációs fájlok után kutatva. A következő könyvtár a `gui`, amiben a felhasználói felülettel kapcsolatos forrásfájlok találhatóak. Az `include` mappában vannak a csak headerekből álló komponensek, valamint azok a header fájlok, amiket több komponens is alkalmaz. Ennek a mappának az almappája például a `meta_lib`, ahol a `variadic` névtér fájljai is megtalálhatóak, valamint az `automata_generator`, ami a szemantikus kiegészítést végző metaprogram fájljait tartalmazza. Az `src` könyvtárban vannak a `gui`n kívüli többi komponens forrásfájljai. A `settings` mappa tartalmazza a beállítófájlokat az alapértelmezett beállításokkal. A fordítási folyamat során ezek a fájlok átmásolásra kerülnek a létrejövő futtatható állomány mellé. Végül a `tests` mappa tartalmazza a futtatható teszteket. A gyökérkönyvtárban megtalálható még a `CMake` fájl, ami a fordításért felelős, a Doxygen konfigurációs fájlja, valamint a licencet tartalmazó fájl is.

### 3.2.7. Fejlesztői környezet

A program lefordításához szükséges a *Boost* könyvtár. Ezen felül opcionális függőség a *Doxygen*, amiből legalább 1.8.0-ás verzió szükséges, amennyiben telepítve van. Ha a Doxygen telepítve van, abban az esetben lehetőség van a forráskódban lévő strukturált dokumentációból, valamint a programhoz tartozó további fejlesztői dokumentációkból HTML oldalakat generálni, ezért a Doxygen használata erősen javasolt. Szintén opcionális függőség a *Google Test* teszt keretrendszer. Amennyiben elérhető, akkor lehetővé válik a unit tesztek futtatására. Minden fejlesztőnek erősen ajánlott ezért ennek a függőségnek a telepítése. Kötelező függősége a Qt keretrendszer is, mivel a felhasználói felület ezt használja. Legalább 5.0-ás verzióra van szükség, mivel az új `connect` szintaxist használja a kód. Mivel a fordításhoz *CMake* build rendszert használok, ezért a *CMake* is kötelező függőség, amiből legalább 2.8.12-es verzió szükséges. Kötelező függőség még a *Clang* fordító is, mivel mind könyvtárként, mind pedig fordítónak ezt használja a projekt. Ebből csak a 3.4-es verzió felel meg. A program csak szabványos C++ és Qt moc kódokat tartalmaz, ezért minden olyan platformon lefordítható, ahol a függőségei lefordíthatóak. A forrásszöveg szerkesztéséhez tetszőleges szerkesztő vagy integrált fejlesztői környezet alkalmazható. A forráskód elérhető a *GitHub*on [15]. A forráskód *BSD* licenc [16] alatt lett kiadva.

```
git clone https://github.com/Xazax-hun/CppQuery.git
cd CppQuery
mkdir build && cd build
cmake ..
```

```
make
make test
make docs
```

Ha a szükséges fejlesztési környezet jelen van, akkor a fordításhoz elsőként le kell tölteni a forráskódot a Github tárolóból vagy átmásolni a CD mellékletéről. Ezután hozzuk létre azt a mappát, ahol a fordítást el szeretnénk végezni, majd adjuk ki a `cmake` parancsot, aminek a paramétere az a könyvtár legyen, ahova a átmásoltuk a kódot. A `cmake` sikeres futása után elérhetővé válik a `make` parancs amivel lefordíthatjuk a kódot. Ennek az eredményeképpen létre fog jönni a `CppQuery` és a `unittest` bináris fájlok. Továbbá ezen fájlok mellé oda fog másolódni a `settings.ini` és a `catalog.txt`. Az utóbbi két fájl a `CppQuery` binárisal együtt csomagolva továbbítható a felhasználók számára. A `make test` parancssal lefutatható a `unittest` program, ami a teszteseteket tartalmazza. A `make docs` parancs hatására létrejön egy `docs/html` mappa, amin belül a programban lévő megjegyzésekből generált *HTML* formátumú dokumentáció található.

### 3.2.8. Továbbfejlesztési lehetőségek

Jelenleg az alkalmazás egy szálon végzi a forrásfájlok elemzését. Több párhuzamos szálon történő elemzés drasztikusan csökkentené az elemzésre fordított időt. Emellett jelenleg az összes fordítási egység szintaxisfája a memóriában van a program teljes futása alatt. Jobban skálázódna az alkalmazás nagy projektekre, hogyha a szintaxisfák ki lennének mentve a merevlemezre és mindegyik szintaxisfa csak akkor kerülne beolvasásra, amikor azon egy lekérdezést le kell futtatni, majd a lekérdezés lefutása után egyből ki lenne takarítva a memóriából. A lekérdezések futtatása is egy szálon fut, nagy teljesítménynövekedés érhető el akár ennek a párhuzamosításával is.

## 3.3. Tesztelés

A szemantikus kiegészítést végző modul jól tesztelhető unit tesztek segítségével és ez a modul a legösszetettebb az egész kódbázisban, ezért az a funkcionalitás a *Google Test* könyvtár segítségével van tesztelve. A tesztek csak akkor fordulnak le, ha a fejlesztő gépén jelen van a könyvtár, ezért azoknál a fejlesztőknél akik ezt a funkcionalitást módosítani kívánják erősen ajánlott a *Google Test* telepítése. A többi funkcionalitást, úgymint a munkamenetet és a felhasználói felületet kézzel teszteltem. Mivel a lekérdezések elemzéséért és lefordításáért, valamint a forrásfájlok

elemzéséért a *Clang* felelős, ami rendelkezik számos tesztesettel, ezért ezek a funkcionalitások nem igényelnek tőlem további tesztelést.

Mivel a szemantikus kiegészítést végző modul szinte teljes egészében metaprogramokból áll, elkerülhetetlen a tesztelés. Az ok, hogy a C++ nyelvben a template-ek lustán példányosulnak, és ameddig nem példányosulnak, addig csak szintaxis ellenőrzést végez rajtuk a fordító. Tehát csak abban az esetben bizonyosodhatunk meg arról, hogy egyáltalán típushiba nélkül lefordul egy metaprogram, hogy azt használjuk is.

```
TEST(SimpleMatcherInterfaceTest, Third) {
    typedef typename Automata<MATCHER(stmt),
                               MATCHER(ifStmt),
                               MATCHER(hasCondition)
    >::result GeneratedAutomata;
    GeneratedAutomata automata;

    std::set<std::string> expected { "ifStmt",
                                     "hasCondition", "stmt" };
    auto tmp = automata.GetComposables("ifStmt");

    std::set<std::string> result(tmp.begin(), tmp.end());

    EXPECT_EQ(expected, result);
}
```

Minden szemantikus kiegészítésért felelős teszteset tartalmazza a szemantikus kiegészítésért felelős metaprogram által generált típus egy példányát. A tesztesetek nem használnak semmilyen globális állapotot. A teszteset elején egy halmaz tartalmazza azt az eredményt, amit elvár. A teszteset végén pedig az elvárt elemeket tartalmazó halmaz kerül összehasonlításra a kapott elemek által definiált halmazzal. Ez azért előnyös, mert így abban az esetben is elbukik a teszteset ha a vártnál több lehetséges kiegészítést ajánl fel a metaprogram, illetve akkor is, hogyha kevesebbet.

Rendelkezek unit tesztekkel, továbbá hagyományos függvényekkel és funktorokkal példányosított metaprogramra is, hogy amennyiben egy változás miatt elbuknak a tesztesetek, egyértelmű legyen, hogy a hiba oka a fő metaprogramban keresendő, vagy pedig a `matcher_traits` modulban van. További tesztesetekkel rendelkezem még a `MetaString` típusról. Összesen 5 teszt van, amikben 9 teszteset található. Ahogy a szemantikus kiegészítés dokumentációjában leírtam, számos módosítást kellett végeznem a modulban annak érdekében, hogy nagy bemenetekre is vállalható

erőforrásigénye legyen a fordítónak, mikor a metaprogramot lefordítjuk. Ezeknek a módosításoknak az elvégzésénél nagy segítséget jelentett a tesztek jelenléte.

A munkamenet és a felhasználói felület tesztelésekor elsősorban az alkalmazás saját forráskódját használtam. Ez egy kényelmes megoldás, mivel a *CMake* build rendszer miatt könnyen előállítható volt fordítási adatbázis. Emellett viszont mégis egy valós projektről van szó, ami olyan gyakran használt könyvtárakat is felhasznál mint a *Qt* és a *Boost*. Ebből kifolyólag valós képet kaptam annak a tekintetében is, hogy hogyan állja meg a helyét az alkalmazás éles helyzetben. A tesztelés során figyeltem arra, hogy minél több szélsőséges esetet kipróbáljak, mint például hibás forráskód elemzése, hibás lekérdezés futtatásának a megkísérlése valamint hibás fordítási adatbázis használata.

A saját kódján kívül egy másik nyílt forráskódú projekten is teszteltem az eszközt. A választásom az *rAthena* [17] projektre esett több okból kifolyólag. Egyrészt ez egy C nyelven írt projekt, így azt is le tudtam tesztelni, hogy C környezetben mennyire használható a *CppQuery*. Másrészt a projekt mérete közel kétszázezer sor, ha az üres sorokat és a megjegyzéseket nem számoljuk. Az alkalmazásom reszponzív volt ekkora projekt esetében is. Ebből a projektből a *Build Ear* [14] segítségével nyertem ki a fordítási adatbázist. Mivel a fordítónak a *make* rendszer ebben az esetben relatív útvonalakkal adta át a forrásfájlok helyét a fordítónak, ezért a saját programomban át kellett írnom az útvonalak kezelését, hogy ezt az esetet is lefedje. Összességében ezért hasznosnak bizonyult egy másik projekten is az eszköz kipróbálása.

Érdekes, hogy még az *rAthena* forráskódjának a töredéke csupán a *CppQuery* forráskódja a sorok számát tekintve, mégis az utóbbi kód elemzése során a program memóriefelhasználása többszöröse az *rAthena* elemzéséhez képest. Ennek egyik oka a számos *Qt* és *Boost* header aminek következtében a *CppQuery* esetén a fordítási egységek mérete jelentősen nagyobb *rAthena*hoz képest. Ráadásul a C++ kódok elemzése nehezebb a C kódok elemzésénél, ezáltal több memóriát is igényel. Tetézi a helyzetet a számos metaprogram, amit a *CppQuery* kódja felhasznál.

A lekérdezések által adott találatokat néhány próbalekérdezés után kézzel ellenőriztem. Mivel az általam kipróbált lekérdezésekre kapott találatok helyesek voltak, valamint az *ASTMatcher* könyvtár sok esetet kimerítő tesztesetekkel rendelkezik, úgy éreztem, hogy a program ezen része nem igényel ennél részletesebb tesztelést.

## 4. fejezet

# Összefoglalás

Egy új fejlesztői munkafolyamatot lehetővé tévő, hiánypótló alkalmazást sikerült kifejlesztenem, amit nem mindennapi eszközökkel valósítottam meg. Mivel a megvalósítás során nagy adatmennyiséget feldolgozó metaprogramokkal kellett dolgoznom, ezért olyan metaprogramozási technikákat voltam kénytelen alkalmazni, amelyek jelenleg még nincsenek bent a köztudatban. A fejlesztést megnehezítette néhány, a *GCC C++* fordítójában fellelhető hiba, aminek következtében a forráskód jelenleg csak a *Clang* fordító segítségével fordítható le. A program nagyban kihasználja a C++11-es szabvány nyújtotta újításokat az optimális működés érdekében.

Az alkalmazást kipróbáltam önmagán, valamint egy közel kétszázezer soros nyílt forráskódú projekten is. A tesztjeim során úgy találtam, hogy a program felhasználóbarát és reszponzív. A mindennapi fejlesztés támogatására alkalmas. A szemantikus kiegészítés jelentős segítséget nyújt mind a lekérdezőnyelv tanulásában mind a lekérdezések megírásában. Skálázhatóság tekintetében a memóriefogyasztás jelenti a legnagyobb akadályt, azonban a továbbfejlesztési lehetőségek között szereplő megoldás segítségével ez az akadály is elhárítható lesz.

Remélem, mind a program maga, mind pedig a program fejlesztése közben kidolgozott módszerek hasznosnak fognak bizonyulni a fejlesztők számára. Főleg, mivel a teljes forráskód bárki számára elérhető BSD licenc alatt egy Github tárolóban.



# Irodalomjegyzék

- [1] Alexandrescu, A.: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley (2001)
- [2] Fowler, M.: *Domain-Specific Languages*, Addison-Wesley, 2010.
- [3] Horváth, G., Kozár, G., Szűgyi, Z.: *Generating type-safe script languages from functional APIs*, in Proc. of the Twelfth International Conference on Informatics, pp. 122–126.
- [4] Lattner, C.: *LLVM and Clang: Next Generation Compiler Technology*, The BSD Conference, 2008.
- [5] Stroustrup, B.: *The C++ Programming Language* Addison-Wesley Publishing Company, fourth edition, 2013.
- [6] Vandervoorde, D., Josuttis, N. M.: *C++ Templates: The Complete Guide*, Addison-Wesley Professional, 2002
- [7] Clang AST matcher library  
<http://clang.llvm.org/docs/LibASTMatchers.html> (2013. jún. 11.)
- [8] Porkoláb, Z., Sinkovics, Á.: *Domain-specific Language Integration with Compile-time Parser Generator Library*, In Proc. 9th international conference on Generative programming and component engineering (GPCE 2010). ACM, October 2010, pp. 137-146.
- [9] Sinkovics, Á., Abrahams, D.: *Using strings in C++ template metaprograms*  
<http://cpp-next.com/archive/2012/10/using-strings-in-c-template-metaprograms/>  
(2013. jún. 02.)
- [10] Abrahams, D., Gurtovoy, A.: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond* Pearson Education, Inc., 2005

- [11] Clang LibTooling, <http://clang.llvm.org/docs/LibTooling.html> (2012. okt. 20.)
- [12] CppDepend, <http://www.cppdepend.com/> (2012. okt. 20.)
- [13] OpenGrok, <http://opengrok.github.io/OpenGrok/> (2014. máj. 03.)
- [14] Build Ear, <https://github.com/rizsotto/Bear> (2014. máj. 02.)
- [15] CppQuery, <https://github.com/Xazax-hun/CppQuery> (2014. máj. 03.)
- [16] BSD License, <http://opensource.org/licenses/bsd-license.php> (2014. máj. 03.)
- [17] rAthena, <https://github.com/rathena/rathena> (2014. máj. 03.)