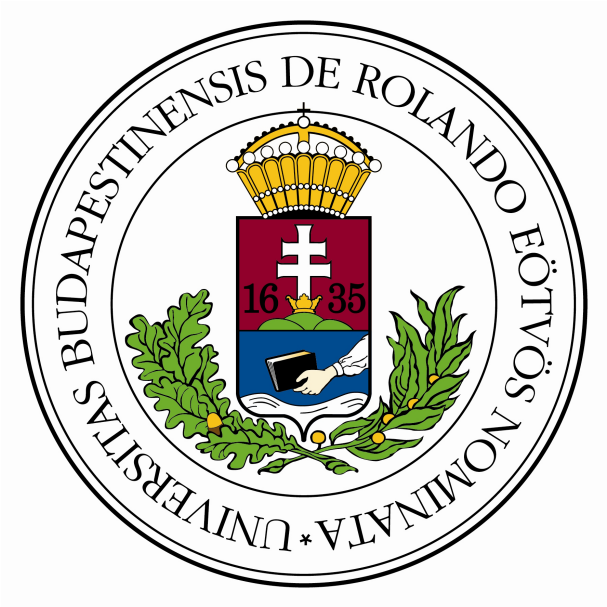

Eötvös Loránd Tudományegyetem
Informatikai Kar

Programozási nyelvek és fordítóprogramok tanszék

Lekérdező nyelv definiálása és prototípus implementálása kódmegértés céljából

Témavezető:
Brunner Tibor
PhD hallgató

Készítette:
Bakos Péter
MSc hallgató

Budapest, 2017

Tartalomjegyzék

1. fejezet

Bevezetés

2. fejezet

Lekérdező nyelv

Lekérdező nyelvvel a szoftverfejlesztéssel foglalkozók már mind találkoztak: relációs adatbázisok adatainak kinyeréséhez, hozzáadásához, törléséhez és módosításához használjuk az SQL (Structured Query Language) nyelvet.Az SQL egy szakterület-specifikus nyelv (angolul domain-specific language, röviden DSL), amely egy bizonyos szakterületre koncentrál (adatbáziskezelés). Az SQL az adatbázis információinak lekérdezéséhez egy néhány utasításból álló parancsot használ. A nyelv komplexitását mutatja, hogy parancsokon belül is lehetnek alparancsok, ezáltal kifinomultabb lekérdezéseket is alkalmazhat a felhasználó. Ezzel ellentétben a LINQ (Language Integrated Query) egy olyan lekérdező nyelv, amely a C# programozási nyelv része. A LINQ-t a Microsoft fejlesztett ki C# nyelv adatstruktúrák információjának kinyerésére az SQL nyelv szintaktikában használatos utasításokkal (Select, Where, Group By, stb.). A forráskódban való kereséshez az idők, ahogy az ipari szoftverek terjedelme egyre nőtt, úgy nőttek a forráskód sorainak száma is. Egy ipari méretű szoftverben a kiigazodás jelentősen komplex feladat. Az egyszerű szöveges keresők nem elég okosak ahhoz, hogy olyan kereséseket végezzenek. Ennek a műveletnek a megkönnyebbítésére az évek során több lekérdező nyelvet is fejlesztettek. Az alábbiakban összehasonlítom a piacon levő már kiforrott lekérdező nyelveket kifejező erő, komplexitás és a felhasználó tanulása szemszögéből.

2.1. jQuery

A JQuery lekérdező nyelv a logikai programozási nyelvek közé sorolható, ami TyRuBa programnyelven alapul [4]. A TyRuBa hasonlít a Prologhoz, azaz egy olyan logikai programozási nyelv, amely rugalmasságot biztosít komplex lekérdezések írásához és szabályokat magasabb rendű kapcsolatok leírásához. A JQuery lekérdező nyelv valójában a TyRuBa nyelv kiterjesztése egy olyan könyvtárral, amikben a forráskódhoz kapcsolódó predikátumok vannak definiálva. Minden predikátumra a nevével hivatkozunk, a paraméterei lehetnek változók (amik elejére kérdőjelet teszünk) vagy attribútum nevek. Például a class(?C, name, HelloWorld) lekérdezés visszaadja az összes olyan osztályt, aminek a name tulajdonsága („property”-je), azaz az osztály neve a HelloWorld szöveggel megegyezik. Komplexebb lekérdezéseket úgy kapunk, ha vesszővel elválasztott lekérdezés-sorozatot írunk. Tegyük fel, hogy keressük az olyan osztályokat, amiknek a létezik olyan függvénye, aminek visszatérési értéke int. Ezt a lekérdezést így írhatjuk le:

```
class(?C, method, ?M),
```

```
method(?M, returnType, int)
```

2.2. .QL

A .QL egy olyan lekérdező nyelv, amely külsőre nagyon hasonlít az SQL nyelvre [2]. A szoftverfejlesztők által jól ismert SELECT-FROM-WHERE hármasra alapoznak, viszont a könnyebb olvashatóság érdekében a szelekció került a lekérdezés végére (FROM-WHERE-SELECT). A hasonlóság azonban kimerül a szintaktikában, a szemantikája teljesen eltérő az SQL nyelvtől. Így a .QL lekérdezések írása könnyen elsajátítható bármely szoftverfejlesztésben minimálisan is jártas felhasználó által az SQL szintaxisát alkalmazva. A .QL a Datalog nyelven alapszik, amely egy egyszerű logikai programozási nyelv. Mivel Datalogban rekurzív lekérdezések is használhatók, így a hívási gráf vagy az öröklődés hierarchiája is egyszerűen, könnyen alkalmazható. A .QL egy objektumorientált lekérdező nyelv, amelyben az osztályok predikátumok, az öröklődés pedig az implikációt jelenti. Ezáltal a lekérdezések újrahasznosíthatók, felhasználhatók akár más projektekben is.

2.2.1. Select utasítások

A Java programozási nyelvben az objektumok egyenlőségvizsgálatára az equals() függvényt használjuk. Ha egy osztály definiál saját equals() függvényt, akkor annak implementálnia kell a hashCode() függvényét is. Ez amiatt szükséges, ha hash alapú adatszerkezetbe rakjuk az adott osztály objektumait, akkor az objektumok ekvivalenciájának vizsgálata ne sérüljön. A lekérdező nyelvek segítségével egyszerűbben felderíthetők az ilyen hibák a forráskódban: keressük meg az olyan osztályokat, amik deklarálják az equals() függvényt, de a hashCode()-ot nem. Ez a lekérdezés a .QL-ben így néz ki:

```
from Class c
where c.declaresMethod("equals") and
      not (c.declaresMethod("hashCode")) and
      c.fromSource
select c.getPackage(), c
```

A Select rész hasonlóan az SQL-hez, kilistázza ezen osztályok csomagnevét és magának az osztálynak a nevét.

2.2.2. Predikátumok

Az elnevezett és parametrizált lekérdezéseket nevezzük predikátumoknak. Ezeket a predikátumokat felhasználhatjuk más lekérdezésekben is. A predikátumok által egyszerűen kaphatunk rendkívül komplex lekérdezéseket anélkül, hogy feláldoznánk az olvashatóságát a lekérdezésnek, hiszen megfelelő névválasztással a predikátumok absztraktabbá, tömörebbé tehetik azokat.

```
predicate between(RefType down, RefType between, RefType up){
   down.hasSuperType*(between) and
   between.hasSuperType*(up)
}
```

A fenti predikátum használatával megkaphatók a *down* és *up* típusok örökölődési hierarchiájában a köztes osztályok. Azaz, a predikátum igaza lesz, ha a *down* osztály a *between-nek* altípusa (leszármazottja), és az *up* a *between-nek* szupertípusa (őse).

2.2.3. Aggregátor függvények

Az SQL nyelv lehetőséget ad különböző összegek, átlagok, számlálások kiszámítására. Ezek az aggregátor függvények használata nehézkes, gyakran csak egymásba ágyazott utasításokkal értelmezhető. A .QL az Eindhoven Kvantor Jelölést használja, ami megkönnyíti az előbb említett függvények írását. Ahogyan SQL-ben, úgy .QL-ben is a SELECT utasításhoz írjuk az aggregátor függvényt. Azonban a GROUP BY utasítás nem szükséges .QL nyelvben. Az aggregátor függvény három részből áll: lokális változók, feltételek, term. Ezek a '—' karakterrel vannak elhatárolva és ezeket előzi meg az aggregátor függvény neve (sum, count, avg, min, max). A lokális változókat felhasználhatjuk a feltételekben, amik értelem szerűen szűrik a találatokat. A term határozza meg, hogy mely értéken szeretnénk aggregálni. A sorrend és azok szerepe nagyban hasonlít a FROM-WHERE-SELECT hármasra.

```
from Package p
where p.hasName("abc.aspectj.ast")
select sum(CompilationUnit cu |
           cu.getPackage() = p |
           cu.getNumberOfLines())
```

Ezzel a lekérdezéssel és a *sum* aggregátor függvény használatával megkapható az *ast* csomagban található forráskód sorainak száma.

2.2.4. Osztályok

.QL nyelvben a predikátumok mellett meghatározhatunk komplexebb entitásokat: osztályokat. Osztályok segítségével definiálhatunk új típusokat a FROM szekcióhoz. Az osztály „konstruktora" megadja, hogy mit várhatunk a típustól: például milyen osztálynak a leszármazottja kell, hogy legyen, vagy milyen függvényeket kell tartalmaznia. Definiálhatunk az osztályhoz olyan függvényt, amellyel megadhatjuk a keresni kívánt típusnak egy adattagját, annak típusára szintén tehetünk megszorítást.

```
class HasIntField {
   Field getIntField() {
      result = this.getAField() and
      result.getType() = int
   }
}
```

Ezzel az osztállyal szűrhetők az *int* típusú adattagokat tartalmazó osztályok. Ezt az osztályt alkalmazhatjuk bármely lekérdezésben.

2.2.5. Generikus lekérdezések

A .QL objektumorientált megközelítésének az előnyeihez sorolható, hogy megírt lekérdezések alkalmazhatók több projektben is. A lekérdezéseket még általánosabbá

teszik a generikus lekérdezések, amiket gyorsan és könnyedén felhasználhatóvá válnak bármely kódbázishoz. Vegyük például a szoftverfejlesztésben gyakran alkalmazott tervezési mintákat (design pattern). A .QL nyelvben lehetőség van a Gyár tervezési minta leírására. A Gyár egy olyan speciális osztály, amely több típusú, de valami által összeköttetésben levő objektumokat állít elő, ezzel nagyobb rugalmasságot adva az objektumok létrehozásához. Mivel a Gyár minta nem konkrét típusokkal írja le működését, így a lekérdezés sem tartalmaz explicite típusokat. De mivel a .QL objektumorientált, ezért egy lekérdezés örökölhet és újra is definiálhat függvényeket, amikkel újrafelhasználhatóság is növekszik. Így már konkrét Gyár osztályokat kereshetünk a lekérdezéssel.

2.3. Program Query Language

A PQL (Program Query Language) kissé eltér a már említett lekérdező nyelvektől, abban hogy csak és kizárólag a forráskódban fellelhető minták keresésére alkalmas [5]. Így egyszerűbben megtalálhatók programozási hibák, figyelmetlenségek, biztonsági rések, memória szivárgások. Ilyen például nem lezárt File típusú objektumok keresése, vagy akár egy adatbázis könyvtár használatakor SQL injekció támadás elleni hibák felismerése. A lekérdezés primitív események szekvenciája. Ezen események objektumokon meghívott függvényekként lehet leírni, ahogyan a Java programozási nyelv szintaxisában megszokhattuk. Rekurzív lekérdezések írására is alkalmas a PQL. Ezt elnevezett allekérdezések segítségével lehet elérni. A felhasználó definiálhat különböző típusú változókat, amiket a lekérdezésben felhasználhat. A keresésben egy változó egy objektumhoz tartozik, a deklarációs részben megadott típusának vagy annak leszármazottjának kell, hogy megfeleljen az adott objektum.

```
query simpleSQLInjection()
uses
   object HttpServletRequest r;
   object Connection c;
   object String p;
matches {
   p = r.getParameter(_);
}
replace c.execute(p)
with Util.CheckedSQL(c, p);
```

A változók reprezentálhatnak adattagokat vagy tagfüggvényeket is, amik szövegesen kell, hogy megegyezzenek a mintában. A „*” karakter helyettesít bármely hosszú karakterláncot. Ha egy adattagot talál a mintában, akkor annak előfordulása a mintában mind ugyanahhoz az objektumhoz kell tartoznia. Lehetőség nyílik a „_” helyettesítő karaktert használni, ami a „*” karaktertől eltérően különböző objektumhoz vagy adattaghoz tartozik. Azonban, az így megtalált szimbólumokat nem lehet vizsgálni vagy visszatéríteni. A PQL lekérdező nyelv változói lehetnek argumentumok (amit egy másik lekérdezésből a jelenlegi lekérdezést meghívva kapott paraméterül), visszatérési érték (a hívó lekérdezés számára visszatérítendő változó), vagy belső változó (amik csak a lekérdezésben használhatók lokálisan).

Az egyszerű parancsokat (függvényhívások) össze lehet kombinálni összetett parancsokká. Az a;b sorozat az „a” megelőzi „b”-t jelöli. Általában, ez azt is jelen-

ti, hogy a két esemény között több esemény is előfordul – a lényeg a két egyedi eseményen alapszik, tehát a szekvenciák általában nem folytonosak. A nyelv lehetőséget kínál arra, hogy egy esemény be nem következését is lehessen definiálni: az a; b;c jelölésben „a” megelőzi „c”-t, úgy hogy a „b” esemény a kettő között nem szerepel. Szintén, lehetőség nyílik az alternáló operátor segítségével („—” karakter) több esemény közül csak az egyik bekövetkezését meghatározni. A within kulcsszó után adható meg, hogy a mintakeresés melyik függvényben történjen. A Java nyelvben a fájlkezeléshez szükséges objektumokat manuálisan kell felszabadítani, különben a rendszer nem kapja vissza az erőforrást a programtól.

```
query forceClose()
uses object InputStream in;
within _._();
matches {
   in = new InputStream();
   ~in.close();
}
executes in.close();
```

A fenti lekérdezés megtalálja az összes olyan függvényt, amelyben a megnyitott fájlokat nem zárja be a programozó. A lekérdezés akkor ad találatot, ha van egy olyan függvény, amiben van egy InputStream objektum viszont a függvény végéig bezárólag nincs meghívva az objektumon a close() tagfüggvény. Mivel nem csak az adott függvény zárhatja le az InputStream-et, hanem az abból hívott függvények is, ezért a within close()-ban a „_” helyettesítő karakter minden függvényhívást is megvizsgál.

A PQL lekérdezések gyakran átvizsgálandó vagy nem kívánt program viselkedést határoznak meg. PQL-ben két lehetőség nyílik az információ logolásra vagy javítási akció végrehajtására. A legegyszerűbb eset az executes klóz. Az ebben leírt függvény akkor fog meghívódni, amikor a keresés találatot kap. A másik mód amikor rögtön javítani szeretnénk a kódot, azáltal hogy kódot helyettesítünk, már meglévő helyére. Ez a replaces részben tehető meg: a replaces kulcsszó utáni kifejezés helyett lesz beillesztve az ezt követő with kulcsszó utáni kódrészlet.

2.4. Soul

A SOUL egy logikai-alapú program lekérdező nyelv [3]. A felhasználó megadja a program karakterisztikáját, struktúráját logikai feltételek segítségével, majd ezt a SOUL már egy konkrét programkód mintájára illeszti. Minden lekérdezés az if kulcsszóval kezdődik és az abban található összes logikai változó első karaktere a kérdőjel. Mint minden logikai nyelvben, itt is predikátumokkal lehet leírni a nyelvben, amit a felhasználó szeretne. A predikátumok előre definiáltak, használatukkal lehet felépíteni egy lekérdezést. Vegyünk egy egyszerű példát: határozzuk meg az olyan függvénypárokat ahol az első valamely mélységben meghívja az utóbbit.

```
if ?outer isStatement,
   ?inner isStatementIn: ?outer
```

A lekérdezés két feltételt tartalmaz. Az első egy unáris predikátum, ami az ?outer változót köti az egyik függvényhez a forráskódban az isStatement predikátum

segítségével. A második feltétele a lekérdezésnek egy bináris predikátum, ami a már előzőleg kötött ?outer változót használja fel, és bevezet egy ?inner változót, amit az isStatementIn predikátum fog kötni egy olyan függvényhez, amit az ?outer változó által kötött függvényen belül hívtak meg. A predikátumokat vesszővel választjuk el egymástól. A forráskód alakjához, struktúrájához, felépítéséhez sokkal közelebb álló kód sablonokat is lehet definiálni a SOUL lekérdezésekben. A kód sablon egy funktort tartalmaz, amit egy paraméter követ. Ez után kapcsos zárójelek között a kódrészlet. A sablon funktora definiálja, hogy mely nyelvi elemek között kell keressen a lekérdezés, majd ezek után a funktoron belüli kódrészlet megtalálása a feladat. A kódrészlet egyfajta keverése a Java és a logikai programok szintaxisának.

```
if jtClassDeclaration(?classDeclaration){
   class ?className{
      private ?fieldDeclarationType ?fieldName;
      ?modifierList ?returnType ?methodName(?parameterList){
         return ?fieldName;
      }
   }
}
```

A fenti lekérdezés az olyan osztályokat keresi meg, ahol az adataggok el vannak rejtve az osztály használóitól és csak getter függvény hívásokkal érhetők el.

2.5. Java Tools Language

Ahogy azt a neve is mutatja a Java Tools Language (JTL) egy Java forráskód keresést megkönnyebbítő lekérdező nyelv [1]. A JTL szintaxisa tömör és intuitív bármely Java nyelvben gyakorlott programozónak. Egy JTL lekérdezés sok esetben pontosan megegyezik egy Java osztály- vagy függvénydefinícióval. Ez növeli a lekérdezés olvashatóságát és nem utolsó sorban a felhasználó magától értetődően tudja JTL mintákat írni. A nyelv két fő adattípusa a Member és a Type. A Member fogalmába tartoznak az osztályok és interface-ek tagjai, beleértve a tagfüggvényeket, adattagokat, konstruktorokat, inicializálókat. A Type adattípusba a Java osztályokat, interface-eket, enum-okat és a Java nyelv primitív típusait (pl. int, float) értjük. Egy JTL program nevesített logikai predikátumok összessége. Egyet kiválasztva a végrehajtott predikátum lesz egy lekérdezés. A predikátumnak kis betűvel kell kezdődjön, a változóknak pedig naggyal.

2.5.1. Egyszerű minták

A Java nyelv kulcsszavai visszaköszönnek a JTL nyelvben, amik ugyanazt a szemantikai jelentést hordozzák, mint amit a forráskódban a Java fordító is értelmez. Például, az int kulcsszó a Java nyelvben állhat egy függvény előtt, mint visszatérési érték, vagy adattag előtt, mint az annak típusát jelezve. A JTL nyelvben ugyanezek a szemantikai jelentésük van. A public minta minden olyan program részletet megtalál, ami publikus láthatóságú osztály vagy például publikus elérésű tagfüggvény, mező. Azonban, vannak a JTL-hez hozzáadott a Java nyelvben nem kulcsszóként ismert natív kulcsszavai. Például az anonymous Type-okra van érzelmezve, amely

névtelen osztályokat keres. A JTL-nek kétféle predikátuma létezik: natív és összetett. A natív predikátumok implementációját nem tartalmazza a nyelv, azaz ha szeretnénk egy ilyen predikátumot kiértékelni, akkor azt csak külső könyvtárak segítségével tehetjük meg. Ezzel ellentétben, az összetett predikátumok definiáltak egy logikai operátorokat használó JTL kifejezés által. A vesszővel elválasztott kulcsszavak a konjugációt jelölik.

```
class {
   public int method;
}
```

Tehát, a public, int mintára az összes olyan publikus függvény fog illeszkedni, aminek a visszatérési értéke int típusú (illetve természetesen az int típusú publikus adattagok is a keresés eredményébe beletartoznak). A vessző elválasztó azonban csak opcionális, elhagyása szintén konjugációt jelent. A diszjunkciót a függőleges vonallal (|) jelezhetjük, míg a negációt a felkiáltó jellel. Következésképpen, a !public ekvivalens kifejezés a protected |private kifejezéssel. Az újrafelhasználást szem előtt tartva, ezeket a mintákat elnevezhetjük, így már csak a definiált kereső kifejezést használhatjuk. Reguláris kifejezések alkalmazása szintén támogatott a JTL nyelvben. Függvények, osztályok, adattagok nevei helyett lehet használni a reguláris kifejezéseket. A függvények paraméterlistájában lehetőség nyílik helyettesítő karakterek használatára. A '*' karakter nulla vagy több paraméternek felel meg, míg a '_' karakter pontosan egy típust helyettesít.

2.5.2. Változók

A változók szerepe a nyelvben hasonlít a Prolog logikai programozási nyelvben használtakhoz. Például ha egy változót kétszer is használunk egy lekérdezésben, akkor annak ugyanazt a típusnevet, függvénynevet vagy osztálynevet kell jelentenie. Az alábbi lekérdezésnek olyan függvények felelnek meg, amiknek a visszatérési értékének típusa megegyezik az argumentumlistájában szereplő valamely paraméter típusával.

```
return_arg := RetType (*, RetType, *);
```

A változók használatának másik területe a predikátumokhoz köthető. A JTL több saját predikátumot definiál, aminek nagy része Java nyelvbeli kulcsszavak. Ilyen az implements[I] predikátum, ami egy Interface típust vár, és visszaadja az összes osztályt, ami implementálja azt. Minden predikátum tartalmaz egy rejtett paramétert, a „vevője" a mintának, amit a This vagy '#' karakterrel lehet hivatkozni. A natív JTL predikátumok közé tartozik többek között: members[M] (igaz, ha M This-nek adattagja, akár definiált, akár örökölt), overriding[M] (igaz, ha This egy olyan függvény, ami felüldefiniálja M-et).

```
container[C] := C.members[This]
```

A container egy olyan JTL által alapértelmezett egyváltozós predikátum, ami akkor teljesül, ha egy osztálynak van a C paraméterű adattagja.

A programozó írhat saját predikátumot is különböző paraméterekkel. Ahogy az a logikai programozásban is lenni szokott, a predikátumok paraméterei nem mások, mint külsőleg elérhető változók. Az alábbi predikátum például egy paramétert fogad.

Amikor egy bizonyos értékkel lesz meghívva, akkor az argumentumnak megfelelő típusú statikus adattagok fognak megfelelni a keresésnek.

```
copty_ctor := constructor(T), T.members[This]
```

A C++ nyelvben egy objektum másolását a másoló konstruktor (copy constructor) végzi. A JTL által ezt a fentebbi lekérdezés szerint írható le. Tehát, egy osztálynak akkor van másoló konstruktora, ha létezik olyan konstruktora, amely egy a ugyanazon típusú objektumot vár paraméterként.

2.6. Összehasonlítás

A vizsgált lekérdező nyelvek közül három logikai programozáson alapul (jQuery, SOUL, JTL), kettő objektum-orientált megközelítést alkalmaz (PQL, .QL). A kódmegértés oka lehet forrás kód refaktorálás. Ez az a folyamat, amikor úgy változtatjuk meg a már meglévő forráskódot, új struktúrát adva, hogy közben a program viselkedése a külső szemlélő számára nem változik meg. Refaktorálás általánosan két esetben szokás alkalmazni: új funkció bevezetésénél (ha a jelenlegi kód nem alkalmas az új implementálásra, vagy megkönnyíti azt), illetve ha bug javítása ezt kívánja meg. Az utóbbi művelethez csak a PQL nyújt lehetőséget, ami szintén a programozó munkáját segíti, hiszen a nyelv alkalmazásával automatikusan lehet az adott hibákat felkutatni és kijavítani.

Vizsgáljuk meg a lekérdező nyelveket a mindennapi kód kereséshez: találjuk meg az összes f nevű függvényt. Majd elemezzük a lekérdezés olvashatóság, kifejező erő, újrafelhasználhatóság szempontjából. jQuery nyelvben ez a lekérdezés az alábbiak szerint írható le:

```
method(?M, name, f)
```

Egyszerű, könnyen olvasható még a jQuery nyelvet nem ismerő számára is. .Qlben ugyanez a kifejezés az SQL szintaxisnak köszönhetően ugyancsak rendkívül jól olvasható.

```
From Method m
Where m.hasName("f")
Select m
```

A SOUL lekérdező nyelvben mivel egy függvény struktúráját kell leírni az olvashatóság szempontjából az ilyen egyszerű lekérdezésekben is elveszhet a felhasználó.

```
if jtMethodDeclaration(?m){
   public ?type f(?parameterList){
      ?statements
   }
}
```

A JTL keressük meg az összes f nevű függvény lekérdezése talán a legtisztább mind közül. Pontosan tükrözi, mit keres a programozó, és azt röviden, tömören lehet leírni.

```
public f(*)
```

A PQL nyelv fejlesztésekor az alapvető programozási hibák és bugok feltárása volt a cél. Ezt úgy érték el, hogy program végrehajtási gráfján lehet keresni, így csak függvényhívás helye határozható meg. Következésképpen, PQL-lel nem lehet kilistázni a forráskódban fellelhető f nevű függvényeket, csak annak hívásait.

3. fejezet

Konklúzió

Irodalomjegyzék

[1] T. Cohen, J. Y. Gil, and I. Maman. JTL: The Java tools language. In OOPSLA, 2006.

[2] O. de Moor, M. Verbaere, and E. Hajiyev. Keynote address: .QL for source code analysis. In SCAM, 2007.

[3] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. The SOUL tool suite for querying programs in symbiosis with Eclipse. In PPPJ, 2011.

[4] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In AOSD, 2003.

[5] M. Martin, V. B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in Object Oriented Programming, Systems, Languages and Applications (OOPSLA'07). ACM, 2005, pp. 365–383.