**UNIVERSITETET I OSLO**
**Institutt for informatikk**

# Implementasjon av OQL mellom Java og ObjectStore

## Masteroppgave
(60 studiepoeng)

Øivind Hagen

**28**. april 2008

# Forord

Jeg vil rette en stor takk til Stein Krogdahl og Ragnar Normann som har vært mine veiledere i denne oppgaven. De har gitt mange grundige tilbakemeldinger, og alltid tatt seg god tid til veiledningstimene.

En stor takk skal også rettes til Mette Westby som har lest oppgaven og gitt konstruktive og oppmuntrende tilbakemeldinger.

28 april 2008, Øivind Hagen

# Innhold


1 Introduksjon......1
2 Bakgrunn......5
- 2.1 Om OQL......5
- 2.2 Om ObjectStore og Java......7
  - 2.2.1 Sesjoner, databaser og transaksjoner......7
  - 2.2.2 Persistente klasser og objekter i ObjectStore......9
  - 2.2.3 Hvordan man lagrer et objekt i en ObjectStore-database......13
- 2.3 Om Java-bindingen......14
  - 2.3.1 Java-grensesnitt (Interfaces)......14
  - 2.3.2 Java-unntak (Exceptions)......15
  - 2.3.3 Koblingen mellom ODMGs objektmodell og Java......15

3 Implementasjonen......17
- 3.1 Verktøy og programvare......17
- 3.2 Scanneren......18
  - 3.2.1 Scannergeneratoren JFlex......18
  - 3.2.2 Scanner for OQL......21
    - 3.2.2.1 Egendefinert kode......21
    - 3.2.2.2 Innstillinger og navngitte regulære uttrykk......22
    - 3.2.2.3 Handlinger for hvert leksem......23
- 3.3 Parseren......24
  - 3.3.1 Parsergeneratoren CUP......25
  - 3.3.2 Parser for OQL......27
    - 3.3.2.1 Java-kode i inputfila til parsergeneratoren......27
    - 3.3.2.2 BNF-grammatikk og konkrete syntakstrær......28
    - 3.3.2.3 Abstrakte syntakstrær......31
- 3.4 Binding av parametere......37
- 3.5 Semantisk sjekk og eksekvering av spørringer......39
  - 3.5.1 DeclarationImport......40
  - 3.5.2 ExprLiteral......41
  - 3.5.3 ExprObjectConstruction......41
  - 3.5.4 ExprStructConstruction......42
  - 3.5.5 ExprCollectionConstruction......44
  - 3.5.6 ExprQueryParam......44
  - 3.5.7 ExprSelect......45
  - 3.5.8 ExprCast......45
  - 3.5.9 ExprCommon......47
- 3.6 Implementasjon av grensesnittene i Java-bindingen......49
  - 3.6.1 Implementasjon av Database-grensesnittet......49
  - 3.6.2 Implementasjon av Transaction-grensesnittet......50
  - 3.6.3 Implementasjon av Implementation-grensesnittet......51

3.6.4 Implementasjon av OQLQuery-grensesnittet............................52
3.6.5 Implementasjoner av DCollection-grensesnittene......................53
3.6.6 Implementasjon av DMap-grensesnittet...................................54
3.7 Bygging og testing av implementasjonen.........................................54
3.8 Eksempelprogram.................................................................................56
4 Kritikk av ODMG 3.0-standarden............................................................61
4.1 Manglende skille mellom scanner og parser......................................61
4.2 Bruk av eksempler som ikke støttes av grammatikken.......................61
4.3 Utilstrekkelig tegnsett i strengliteraler...............................................62
4.4 Database-grensesnittet mangler status-metoder.................................62
4.5 Database-grensesnittet mangler metode for å tømme databaser.........62
4.6 Java-bindingen støtter ikke listing av rot-objekter.............................63
4.7 Java-bindingen bruker ikke generics i samlingsklasser......................63
4.8 Java-bindingen definerer structer for overfladisk..............................64
4.9 Array-indekser kan ikke brukes i dot-notasjon..................................65
4.10 Nøkkelord kan ikke brukes i dot-notasjon.......................................65
5 Oppsummering og videre arbeid...............................................................67
5.1 Kjente feil og mangler i implementasjonen.......................................67
5.2 Videre forskning og utvikling............................................................70
6 Appendikser.............................................................................................71
6.1 Tabell med koblingen mellom ODMGs objektmodell og Java..........71
6.2 Beskrivelse av OQL-leksemer i JFlex...............................................72
6.3 Beskrivelse av OQL-syntaks i CUP...................................................74
6.4 Kildekode...........................................................................................86
6.5 Unntakshierarkiet i ODMG 3.0.........................................................87
7 Bibliografi.................................................................................................89

# 1 Introduksjon

OQL (Object Query Language) er et spørrespråk i samme stil som SQL, og det brukes til å hente ut data fra objektorienterte databaser etter kriterier som brukeren spesifiserer. Språket er en del av standardene fra *Object Data Management Group* (*ODMG*). Disse standardene ble laget i 90-årene for å få gode standarder for objektorienterte databasehåndteringsystemer (OODBMS).

En stor del av ODMG-standardene bygger videre på tidligere standarder. Spesielt bygger OQL på erfaringene fra SQL, samtidig som det er bedre planlagt fra et teknisk synspunkt. SQL ble først implementert, og deretter ble standardene utarbeidet. For OQL ble derimot standardene utarbeidet først.

I denne oppgaven er mesteparten av språket OQL, slik det er definert i standarden ODMG 3.0 ([1]), implementert mot DBMS'et ObjectStore for vertspråket Java. Denne standarden inneholder retningslinjer og grensesnitt som beskriver hvordan dette skal gjøres for både Java, C++ og SmallTalk. Implementasjonen omfatter også flere deler av ODMG 3.0 som OQL støtter seg på.

ODMG består av representanter fra mange store og små selskaper, bl.a. Sun Microsystems, Ericsson og CERN. Mange av representantene kommer fra virksomheter som jobber med OODBMS og objektorienterte programmeringsspråk (heretter kalt OO-programmeringsspråk).

De hadde sitt første møte hos Sun Microsystems høsten 1991 og ga ut første versjon av standarden, ODMG 1.0, i 1993. Etter dette har versjon 1.1, 1.2, 2.0 og 3.0 blitt publisert. Versjon 3.0 kom i 1999, og er foreløpig den siste. Denne versjonen blir ikke regnet som fullstendig, og målet er å videreutvikle standarden til versjon 4.0. Denne utviklingen har imidlertid stoppet opp.

Den foreløpig siste standarden har altså vært ferdig i mange år, men ifølge Wikipedia ([12]) finnes det ingen komplett implementasjon av OQL. Dette er grunnet høy kompleksitet i språket og i rammeverket rundt.

Det kan se ut som at dette er i ferd med å endre seg. Apache OJB (tilgjengelig på http://db.apache.org/ojb) er en implementasjon av ODMG 3.0 hvor mesteparten av funksjonaliteten i OQL er på plass. OJB arbeider imidlertid ikke mot noe OODBMS, men oversetter objekter til rader i relasjonsdatabaser. I skrivende stund har den to kjente feil som er under utbedring.

En annen interessant implementasjon er OQL-SERF ([2]). Denne baserer seg

på versjon 2.0 av ODMG-standarden, og fokuserer på evolusjon av skjemaer i objektorienterte databaser. I denne er deler av OQL implementert mot DBMS'et PSE Pro 2.0, som er en enbrukerutgave av ObjectStore.

Når man utvikler systemer med større mengder data som skal lagres over tid, er det vanlig å lagre disse dataene i databaser gjennom et DBMS. Valget faller ofte på et relasjons-DBMS (RDBMS), ettersom slike er mest utbredt og kjent.

Men om man skriver i OO-programmeringsspråk og har behov for å lagre hele objekter, er det tungvint å lagre dem i relasjonsdatabaser. For å kunne gjøre dette, må tabeller opprettes for alle klassene som har objekter som skal lagres. Videre må det skrives kode for å oversette innholdet i objekter til rader i slike tabeller og omvendt.

Dette problemet kalles ofte *impedance-mismatch*, og kommer av at RDBMS bygger på relasjonsmodellen ([3] og [4]), som er annerledes enn de forskjellige objektmodellene som OO-programmeringsspråk bygger på.

Det finnes riktignok flere rammeverk som forenkler store deler av denne oversettelsen. JBoss Hibernate og Oracle TopLink er to slike rammeverk (begge omtalt i [9]). Koding av grunnleggende database-operasjoner som å lagre, slette, oppdatere og hente ut objekter blir kraftig redusert og forenklet av disse rammeverkene. Samtidig kan man bruke spørrespråk som følger med rammeverkene for å hente ut data etter spesifikke kriterier.

Men det er i slike tilfeller ofte bedre å lagre objekter i OODBMS. Hovedfordelen er at datamodellen er den samme i programmeringsspråket og i databasen. Man lagrer altså objektene i databasen som de er, uten å oversette disse til å passe inn i en annen datamodell.

ObjectStore er et slikt OODBMS. Det ble påbegynt i 1988 av Object Design ([13]), og kjøpt av Progress i 2002. OQL er i denne oppgaven implementert mot dette OODBMS'et, og implementasjonsprosessen blir gjennomgått i dette dokumentet. Oppgaven påpeker også noen feil og mangler i implementasjonen og standarden.

Implementasjonen i denne oppgaven kan beregne resultater av OQL-spørringer, og de fleste funksjoner er implementert. Blant funksjoner som ikke er implementert er gruppering i select-uttrykk, hvilket ville tatt for lang tid å implementere.

Derimot kan implementasjonen verifisere syntaksen til alle gyldige OQL-spørringer. Kildekoden som er skrevet skulle være rimelig forståelig, hvilket i noen tilfeller går litt på bekostning av ytelse.

ODMG 3.0-standarden forutsetter at DBMS'et som OQL implementeres mot understøtter en viss funksjonalitet. I de tilfellene hvor slik funksjonalitet mangler eller fungerer annerledes enn forutsatt, er dette diskutert, og løsninger foreslått. For eksempel er transaksjonsmodellen som er definert i ODMG 3.0 annerledes enn transaksjonsmodellen i ObjectStore. Dermed følger ikke transaksjonsmodellen i denne implementasjonen ODMG 3.0-standarden.

Språket ODL (Object Definition Language) er også definert i ODMG 3.0, men er ikke implementert i denne oppgaven. ODL brukes til å definere skjemaer i objektorienterte databaser. Informasjonen i slike skjemaer skal brukes av OQL under eksekvering av spørringer. Som følge av denne avgrensningen blir skjemaer i databaser som bruker denne implementasjonen definert på en annen måte.

Implementasjonen som denne oppgaven har resultert i burde greit kunne videreutvikles til en komplett ODMG 3.0-implementasjon. Den kan også brukes som et utgangspunkt for å implementere ODMG 3.0-standarden mot andre OODBMS, f.eks. db4objects (http://www.db4o.com).

Implementasjonen inneholder 7054 linjer manuelt skrevet kildekode (i tillegg til kildekoden generert av verktøyene JFlex og CUP). 2511 av disse linjene utgjør 175 tester som ble kjørt hver gang implementasjonen ble bygget, for å sjekke at funksjonalitet ikke hadde blitt ødelagt av endringer.

For å lettere forstå hvordan implementasjonen er gjort, behandles bakgrunnsinformasjon for oppgaven i kapittel 2. I kapittel 3 diskuteres implementasjonsprosessen. Kapittel 4 omtaler feil og mangler i ODMG 3.0, og kapittel 5 oppsummerer hva som gjenstår å implementere.

# 2 Bakgrunn

Dette kapittelet gir en del bakgrunnsinformasjon som kan være nyttig for å forstå hvordan en OQL-implementasjon er gjort. Det er hovedsaklig en introduksjon til de teknologier og den terminologi som brukes i resten av dokumentet.

Jeg bruker engelsk terminologi der jeg mener dette er til det bedre. Det antas at leseren har en viss kunnskap om SQL og DBMS.

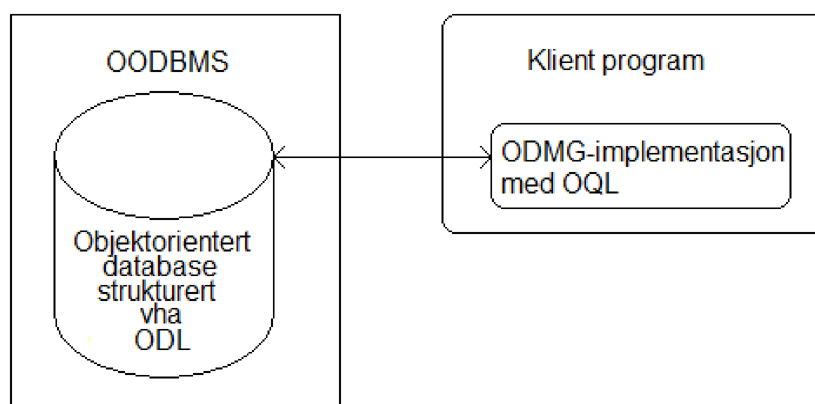
## 2.1 Om OQL

Vi skal her se nærmere på OQL og de teknologiene dette språket støtter seg på. Dette omfatter bl.a. hvordan et OODBMS fungerer.

OQL er altså et spørrespråk som brukes til å hente ut objekter fra objektorienterte databaser etter kriterier brukeren spesifiserer. Språket er definert i ODMG 3.0-standarden ([1]), som inneholder definisjonen av både ODMGs objektmodell og en beskrivelse av språkene ODL og OQL. Den dynamiske og statiske semantikken for disse språkene og den underliggende datamodellen blir forklart i presis engelsk.

For å bidra til å holde implementasjoner portable, beskriver ODMG 3.0 hvordan OQL og ODL skal implementeres for vertspråkene C++, Java og Smalltalk. Disse beskrivelsene kalles *språkbindinger*.

OQL er ment å være en del av et større grensesnitt som brukes mot en database i et OODBMS. Man skal definere hvilke klasser en database skal kunne håndtere (skjemaet i databasen) i ODL. Som tidligere nevnt er ikke ODL implementert som del av denne oppgaven.




*Illustrasjon 1 : Slik brukes en ODMG-implementasjon*

Når skjemaet i en database er definert, kan man koble seg til denne databasen ved å bruke en ODMG-implementasjon i et klientprogram (se illustrasjon 1). Deretter kan man lagre objekter i databasen og utføre spørringer som henter objekter ut av databasen.

ODMG designet OQL slik at det har så mange likhetstrekk med SQL som mulig, samtidig som det skulle være vesentlig ryddigere. Begge støtter select-uttrykk, gruppering og aggregater. Matematiske uttrykk og førsteordens logikk er også med i språkene. Man kan oppnå relativt like resultater i SQL og OQL ved å følge en ganske lik fremgangsmåte.

SQL og OQL benyttes mot forskjellige underliggende datamodeller. SQL er bygget på relasjonsmodellen, og henter ut rader fra tabeller. OQL er bygget på objektmodellen definert i ODMG 3.0, og henter ut objekter fra *ekstensjoner*.

En ekstensjon er et samlingsobjekt som holder på alle objekter fra samme klasse. Hvis klasse `A` er en subklasse av klasse `B`, vil ekstensjonen til klasse `B` inneholde alle instanser av klasse `A` og `B`. Ekstensjonen til klasse `A` er altså et subsett av ekstensjonen til klasse `B`.

Dataene som blir behandlet av SQL og OQL blir altså strukturert på forskjellige måter. Man kan for eksempel strukturere informasjon om personer som rader i tabeller eller som objekter i ekstensjoner.

På samme måte som man kan definere prosedyrer og funksjoner i f.eks. PL/SQL, kan man definere navngitte spørringer og variable i OQL. Man kan for eksempel definere en spørring i OQL som henter ut personer over `x` år, og kalle denne spørringen med argumenter:

```
define query eldreEnn(int alder) as
    select p
    from person p
    where p.alder > alder;
select *
from eldreEnn(40)
```

I objektorienterte databaser som ikke er tomme er det alltid ett eller flere navngitte objekter, bl.a. alle aktuelle ekstensjoner, som man bruker som utgangspunkt når man navigerer i databasen. Disse objektene kalles *rot-objekter*. Alle andre objekter i databasen nåes ved å følge pekere fra disse.

OQL kan referere til slike rot-objekter. Hvis et rot-objekt heter `Kunderegister`, kan man hente ut dette objektet ved å skrive følgende spørring:

```
Kunderegister
```

### *2.2 Om ObjectStore og Java*

For å forstå hvordan OQL er implementert og hvordan det kan brukes mot ObjectStore, er det viktig å vite litt om hvordan ObjectStore er bygget opp og fungerer. Det er også viktig å vite litt om hvordan kommunikasjonen mellom ObjectStore og Java foregår.

Java-programmer kommuniserer med ObjectStore-databaser gjennom *ObjectStore Java Interface*, heretter kalt *OSJI*. OSJI er en samling Java-klasser, hvorav noen tar seg av kommunikasjon med ObjectStore.

Andre klasser i OSJI beskriver samlingsobjekter som er kompatible med ObjectStore (dvs, de kan lagres i en ObjectStore-database). Disse klassene er i samme stil som de man finner i `java.util.Collection`-rammeverket.

De delene av OSJI som er viktige for oppgaven blir beskrevet i dette kapitlet. Det meste av denne informasjonen er hentet fra [10].

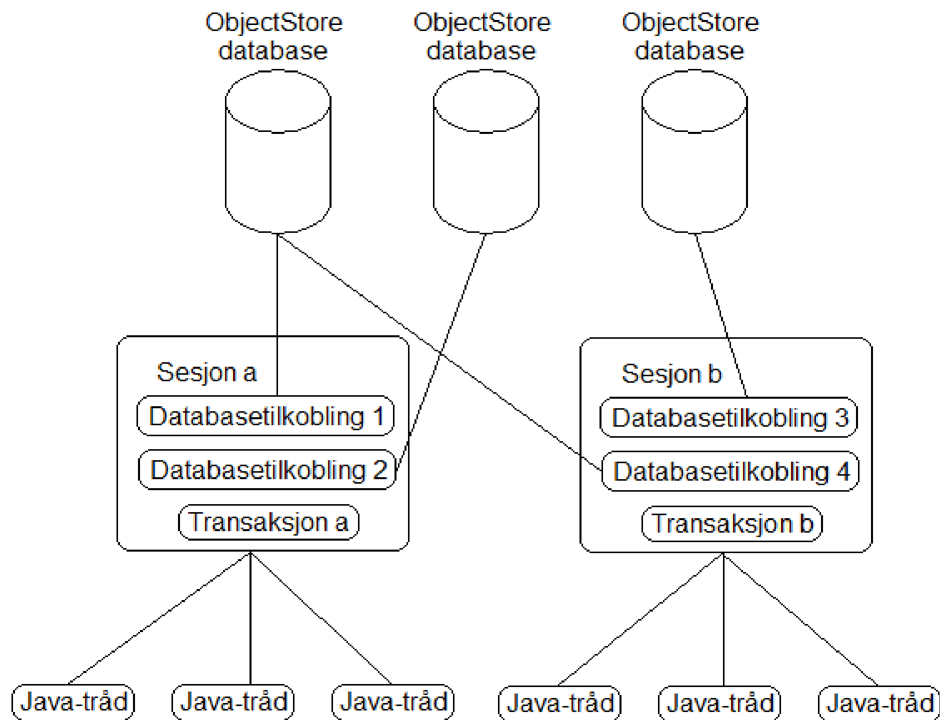
#### 2.2.1 Sesjoner, databaser og transaksjoner

Et `Session`-objekt representerer en *sesjon*, som er en kontekst for kommunikasjon med en eller flere ObjectStore-databaser. En slik sesjon kan inneholde flere databasetilkoblinger, men kan ikke kjøre flere enn én transaksjon om gangen.

En Java-tråd (heretter kalt en tråd) som skal jobbe mot en ObjectStore-database må være tilknyttet en sesjon. Samme sesjon kan være knyttet til flere tråder, men én tråd kan kun være knyttet til én sesjon. Ettersom samme sesjon kun kan kjøre én transaksjon om gangen, så kan ikke flere transaksjoner kjøre parallelt i samme tråd.

Et `Database`-objekt er selve tilkoblingen til en ObjectStore-database. For å kunne åpne og lukke databasetilkoblinger, må tråden som gjør dette være tilkoblet en sesjon. En database må være åpnet i skrivemodus om man skal kunne lagre, slette eller oppdatere objekter i den. Hvis man kun skal utføre leseoperasjoner på objekter, er det tilstrekkelig at databasen åpnes i lesemodus.

En transaksjon samler operasjoner mot en eller flere databaser til en atomær operasjon. Man kan kun aksessere objekter i databaser fra en tråd som er koblet til en sesjon med en transaksjon som har startet. To transaksjoner kan kjøre samtidig i hver sin sesjon og dermed hver sin tråd. Illustrasjon 2 viser samspillet mellom transaksjoner, sesjoner, tråder og databaser.




*Illustrasjon 2 : Samspill mellom tråder, sesjoner, transaksjoner og databaser*

Data låses på sidenivå, hvilket betyr at når et objekt aksesseres, låses hele siden objektet ligger i. Dermed låses som oftest flere objekter når kun ett aksesseres. Hvor mange bytes som utgjør en side bestemmes av operativsystemet ObjectStore-installasjonen kjører på.

Hvis en transaksjon aborterer, blir endringene i de persistente objektene (dvs. de som er lagret i databasen) rullet tilbake. Hvis endringer i de transiente objektene (dvs. de som ikke er lagret i databasen, men kun eksisterer i vertsprogrammet) skal rulles tilbake, er det vertsprogrammets ansvar å gjøre dette.

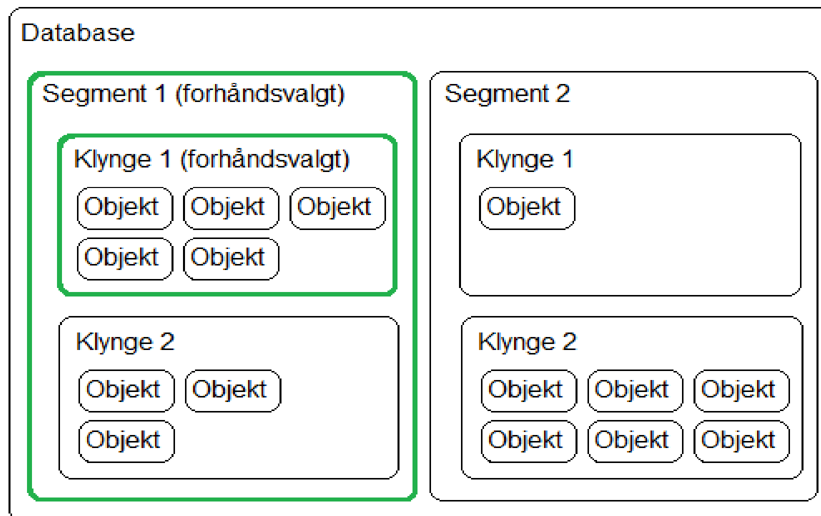
Det er vanlig at OODBMS gir mulighet til å dele opp databaser slik at objekter som ofte aksesseres sammen kan lagres i nærheten av hverandre. ObjectStore implementerer denne strategien ved å dele opp databaser i segmenter som igjen deles opp i klynger. Slike klynger inneholder objekter.

Man kan fordele objektene man skal lagre i forskjellige databaser, forskjellige segmenter i databaser og forskjellige klynger i segmenter. Dermed får hvert objekt en adresse som består av databasenavn, segmentnummer, klyngenummer og adresse i klyngen.

Objekter som ofte aksesseres sammen burde lagres i samme klynge. Objekter som ikke aksesseres sammen fullt så ofte kan likevel lagres i samme

segment. Objekter som ikke er relatert til hverandre kan gjerne lagres i forskjellige segmenter.

Slik optimering er ikke påkrevd. Alle ObjectStore-databaser har et forhåndsvalgt segment og en forhåndsvalgt klynge hvor objekter lagres dersom annet ikke spesifiseres (se illustrasjon 3).




*Illustrasjon 3 : Databaser, segmenter, klynger og objekter*

### 2.2.2 Persistente klasser og objekter i ObjectStore

For at objekter skal kunne lagres i ObjectStore, må visse kriterier tilfredsstilles. Vi skal se nærmere på disse kriteriene i dette kapittelet.

Det er tre typer Java-klasser å forholde seg til når man jobber med ObjectStore fra Java:

1. Den ene typen klasse beskriver objekter som skal kunne lagres i ObjectStore. Disse klassene og objektene de beskriver er *Persistence Capable* (heretter kalt *PC*), og utgjør selve datamodellen i en applikasjon. En applikasjon som holder orden på salg av biler, vil typisk ha en `Kunde`- og en `Bil`-klasse som er PC.

   At et objekt er PC, betyr ikke at det er lagret i databasen, men at det *kan* lagres i databasen. Et PC-objekt som ikke har blitt lagret i databasen er *Transient*. Transiente objekter forsvinner når Java-programmet de eksisterer i terminerer.

2. Den andre typen klasse beskriver objekter som *ikke* skal kunne lagres i ObjectStore, men som likevel skal kunne *behandle* objekter som er

PC. Disse klassene og objektene de beskriver er *Persistence Aware* (heretter kalt *PA*), og inneholder typisk forretningslogikk.

I eksemplet med applikasjonen som holder orden på salg av biler vil det typisk finnes objekter hvis ansvar er å legge til nye kunder og registrere salg av biler. Slike objekter *behandler* objekter som er PC, og må dermed være PA.

3. Objekter av den tredje typen klasse skal hverken kunne lagres i en ObjectStore-database eller behandle objekter som skal kunne lagres i en ObjectStore-database. Komponent-objekter i grafiske brukergrensesnitt og objekter som tar seg av kryptering og komprimering er typiske eksempler på objekter av slike klasser.

Man kan gjøre klasser PC enten manuelt ved å skrive koden til klassen etter bestemte regler, eller automatisk ved å postprosessere bytekoden til klassen etter den har blitt kompilert. Det sistnevnte gjøres ved hjelp av et program som følger med ObjectStore.

Selve endringen i klassen består av å la klassen implementere et bestemt Java-grensesnitt og å legge til to medlemsvariable som holder orden på identiteten og tilstanden til objekter av klassen. Det vil også settes inn metodekall som sørger for at innholdet i et objekt av klassen blir hentet ut fra databasen før objektet aksesseres.

Man kan ikke gjøre en klasse PC dersom en medlemsvariabel er typet med en klasse som ikke er PC. Anta at en klasse `Person` har en medlemsvariabel av typen `java.util.HashMap`. Bytekoden til denne klassen kan ikke postprosesseres, ettersom `java.util.HashMap` ikke er PC.

Slike klasser kan likevel bli PC hvis alle medlemsvariabler som er typet med klasser som ikke er PC blir deklarert med Java-nøkkelordet `transcient`. Medlemsvariabler som blir deklarert med dette nøkkelordet skal *ikke* lagres i databasen, og trenger dermed ikke å være PC.

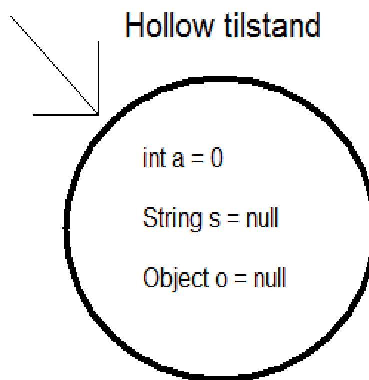
Man kan gjøre klasser PA på samme måter som man kan gjøre klasser PC, dvs. enten manuelt eller automatisk. Det er færre endringer som skal til for å gjøre en klasse PA. PA-objekter må hente ut innholdet til PC-objekter når disse aksesseres. Et PA-objekt må også endre tilstanden til hvert PC-objekt det skriver til.

Men et PA-objekt har ikke forskjellige tilstander slik som et PC-objekt, og trenger derfor ikke ha en medlemsvariabel som holder orden på dette. Det har heller ikke en adresse i en database, og trenger ikke å ha en medlemsvariabel for dette heller.

Ettersom klasser som skal postprosesseres ikke er PA eller PC før de har blitt postprosessert, er det viktig å tenke på rekkefølgen man postprosesserer filer i. Anta to klasser, `A` og `B`, som skal postprosesseres til å bli PC. Hvis `A` har en peker til `B`, og `B` igjen har en peker til `A`, så må bytekoden til disse klassene postprosesseres samtidig. Dette kalles å postprosessere filer i samme *Batch* (side 219 i [10]).

Når en klasse først er PC, kan man lagre objekter av klassen i databasen. Når man har hentet ut et slikt objekt fra en database, så går dette objektet igjennom tre tilstander. Disse tilstandene kalles *Hollow*, *Active* og *Stale* (side 7 og 8 i [10]).

1. Når man har hentet ut et objekt fra en database, er det først i Hollow tilstand. Det betyr at dataene objektet inneholder ikke har blitt hentet ut fra databasen enda (se illustrasjon 4). Man har altså en peker på et objekt med udefinerte verdier. Hvis man skal aksessere dette objektet, så må dette gjøres fra et objekt som er PC eller PA.



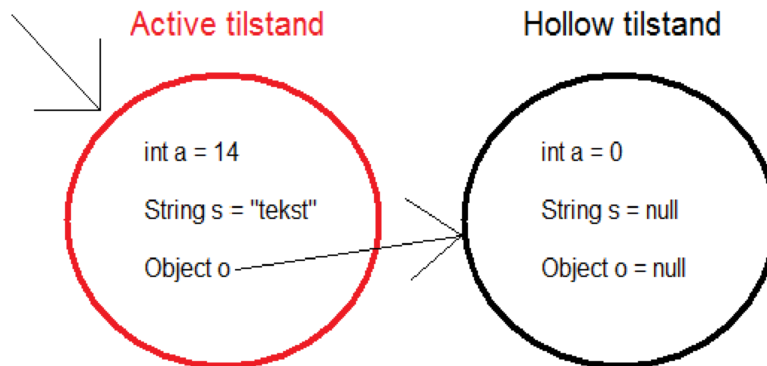

*Illustrasjon 4: Et objekt i Hollow tilstand har udefinerte verdier*

2. Når man aksesserer objektet, vil det endre tilstand til Active. Det betyr at innholdet til objektet leses fra databasen og kopieres inn i objektet. Dermed har objektet i vertsprogrammet samme innhold som det objektet det representerer i databasen.

   Samtidig markeres det at objektet ikke har blitt endret siden det ble hentet ut av databasen. Hvis man skriver til objektet, vil dette markeres i en medlemsvariabel som holder orden på objektets tilstand. Dette gjør at kun de objekter som har blitt skrevet til i løpet av en transaksjon lagres i databasen når transaksjonen fullfører.

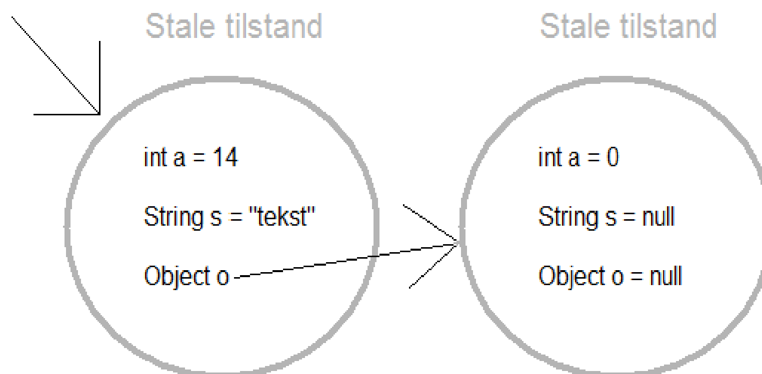
   Ofte peker et objekt i Active tilstand på andre objekter som er lagret i databasen. Slike objekter opprettes i Hollow tilstand når de kobles til det aktive objektet, og går over til Active tilstand hvis de aksesseres

(se illustrasjon 5). Dermed blir kun innholdet til objektene som aksesseres kopiert ut av databasen.




*Illustrasjon 5 : Et objekt i Active tilstand har blitt synkronisert med det tilsvarende objektet i databasen, og peker ofte på andre objekter i Hollow tilstand*

3. Når en transaksjon fullfører, endrer de persistente objektene i transaksjonen tilstand til Stale (hvis man ikke har stilt inn OSJI på en annen måte). Et objekt i Stale tilstand kan hverken leses fra eller skrives til.




*Illustrasjon 6 : Objekter i Stale tilstand kan ikke aksesseres*

ObjectStore kan kun adressere *objekter*, ettersom disse har en egen medlemsvariabel for adresse når de er PC. Primitive typer som `int`, `long` og `boolean`, kan derfor ikke lagres *frittstående* i ObjectStore-databaser. De må enten være medlemsvariabler i et PC-objekt, eller pakkes inn i såkalte *wrapper*-objekter.

For eksempel kan `int`-primitiver pakkes inn i `Integer`-objekter, som igjen kan lagres. Objekter av disse wrapper-klassene blir behandlet spesielt av ObjectStore, ettersom de kan lagres selv om klassene ikke er postprosessert.

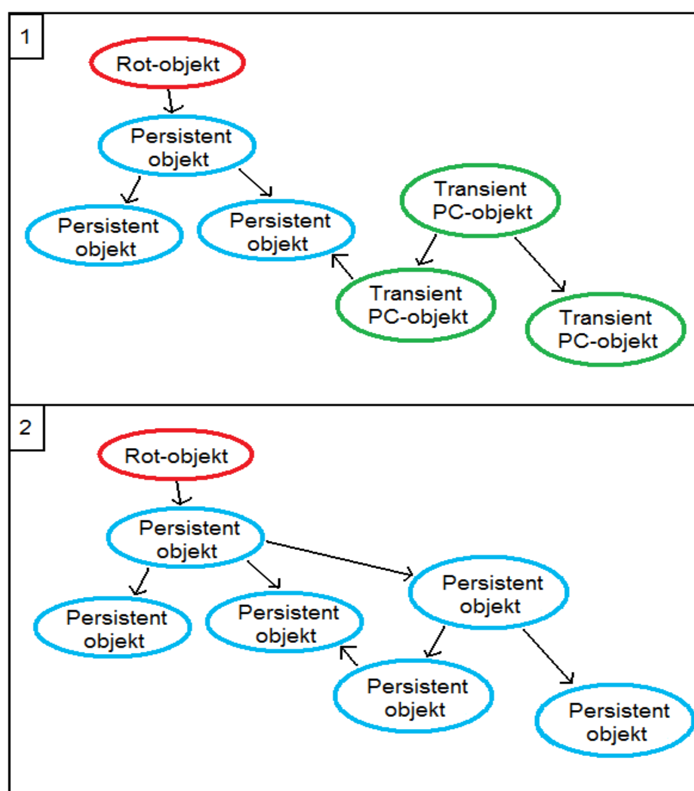
### 2.2.3 Hvordan man lagrer et objekt i en ObjectStore-database

Lagring av objekter i en objektorientert database fungerer på en helt annen måte enn lagring av rader i en relasjonsdatabase. Vi skal nå se på hvordan dette gjøres, både generelt i OODBMS og i ObjectStore.

I en objektorientert database som ikke er tom, har man ett eller flere rot-objekter, som har et navn som er unikt i hele databasen. Objekter som er rot-objekter, eller kan nåes ved å følge pekere fra ett eller flere rot-objekter, blir bevart i databasen.

Å lagre et nytt objekt kan dermed gjøres på én av to måter:

1. Det nye objektet lagres som et rot-objekt ved at det får et unikt navn i databasen.
2. Det nye objektet pekes på av et objekt som allerede er lagret i databasen.




*Illustrasjon 7 : Transitive Persistence, også kjent som Persistence by Reachability*

Man kan la mange transiente PC-objekter peke på hverandre (del 1 av illustrasjon 7), og la et objekt som er lagret i databasen peke på ett av disse objektene (del 2 av illustrasjon 7). ObjectStore vil da lagre alle objektene i databasen når transaksjonen fullfører. Dette kalles *Transitive persistence* eller *Persistence by reachability*.

## 2.3 Om Java-bindingen

OQL er i denne oppgaven implementert for vertspråket Java. Koblingen mellom Java og OQL blir kalt *Java bindingen* (side 239 i [1]). Vi skal se nærmere på denne bindingen i dette kapittelet.

### 2.3.1 Java-grensesnitt (Interfaces)

Java-bindingen definerer følgende Java-grensesnitt:

- `Implementation`
- `Database`
- `Transaction`
- `OQLQuery`
- `DCollection`
- `DArray`
- `DBag`
- `DList`
- `DMap`
- `DSet`

`Implementation`-grensesnittet er definert slik at man skal kunne bruke implementasjoner fra forskjellige leverandører på samme måte. De fleste av metodene instansierer objekter som implementerer de andre grensesnittene i Java-bindingen.

`Database`-grensesnittet definerer generell databasefunksjonalitet, bl.a. åpning av databaser i lesemodus, skrivemodus og eksklusivt modus. Å åpne en database i eksklusivt modus låser hele databasen for andre brukere.

Videre definerer dette grensesnittet metoder for å opprette, hente ut og slette rot-objekter, samt metoder for å legge objekter i ekstensjoner.

`Transaction`-grensesnittet definerer metoder for å starte og abortere transaksjoner, samt å låse objekter.

`OQLQuery`-grensesnittet definerer metoder relatert til OQL-spørringer. Spørringer kan også utføres fra objekter av klasser som implementerer `DCollection`-grensesnittet.

`DBag`-, `DSet`-, `DArray`- og `DList`-grensesnittene utvider `DCollection`-grensesnittet. Disse grensesnittene definerer forskjellige samlingsklasser.

`DMap`-grensesnittet utvider *ikke* `DCollection`-grensesnittet. Grensesnittet definerer samlingsklasser med hashmap-funksjonalitet (også kjent som *dictionarys* i andre språk).

### 2.3.2 Java-unntak (Exceptions)

ODMG har definert et hierarki av Java-unntak (se appendiks 6.5). Vi ser på et par eksempler på hvor slike unntak skal brukes. All informasjon er hentet fra [1].

`ODMGException` oppstår i objekter av klasser som implementerer Java-grensesnittene definert i ODMG 3.0. Hvis man for eksempel forsøker å åpne en database som ikke finnes, vil dette oppdages i en implementasjon av `Database`-grensesnittet, som kaster et unntak av en subklasse av `ODMGException`.

`ODMGRuntimeException` oppstår når noe går galt utenfor implementasjoner av Java-grensesnittene i ODMG 3.0. Et eksempel er når et objekt av en klasse som ikke er PC blir forsøkt lagret i en database. I slike tilfeller oppdages feilen i OSJI eller i databasen.

Det defineres også fire unntaksklasser for unntak relatert til tolking av OQL-spørringer og binding av objekter til parametrene i disse. Det vil f.eks. kastes et unntak hvis det forsøkes å binde fire objekter til en spørring med tre parametere.

### 2.3.3 Koblingen mellom ODMGs objektmodell og Java

De forskjellige typene primitiver og objekter definert i ODMGs objektmodell skal representeres av objekter og primitiver i Java.

For eksempel skal `long`-literaler i denne objektmodellen behandles som `int`-primitiver eller `Integer`-objekter i Java. `Timestamp`-literaler skal

behandles som `java.sql.Timestamp`-objekter i Java.

En tabell over hva de forskjellige primitivene og objektene i OQL skal representeres av i Java er vedlagt i appendiks 6.1.

# 3 Implementasjonen

## 3.1 Verktøy og programvare

I dette kapittelet skal vi se på de viktigste verktøyene og programmene som er brukt i oppgaven. Vi lister dem opp med noen få opplysninger her i begynnelsen av kapittelet, og ser nærmere på noen av dem senere.

### JFlex

JFlex er en scannergenerator, og likner mye på *Lex*. Den er skrevet i Java og genererer Java-kode, og er gratis tilgjengelig på http://www.jflex.de. Der ligger også brukermanualen ([5]) og annen dokumentasjon. Jeg har valgt å bruke JFlex på grunn av tidligere erfaring med dette verktøyet.

### CUP

CUP er en parsergenerator som er skrevet i Java og genererer Java-kode. En parser generert av CUP kan parsere såkalte LALR-grammatikker. Jeg har valgt å bruke CUP på grunn av tidligere erfaring. Den er gratis tilgjengelig og er dokumentert på http://www2.cs.tum.edu/projects/cup.

### ObjectStore

Implementasjonen ble testet mot en ObjectStore-installasjon (versjon 6.1, Service Pack 2) som kjører på en av Linux-tjenerne på nettverket til UiO. Hver gang implementasjonen skulle testes, ble den kompilert og lastet opp til denne tjeneren.

### Eclipse

Eclipse er en gratis plattform for utvikling av bl.a. Java-programmer. Den er tilgjengelig på http://www.eclipse.org. Jeg har valgt å bruke Eclipse fordi jeg har brukt det i flere år og synes det er enkelt å bruke.

Ofte skal man utføre en oppgave som Eclipse ikke har innebygde verktøy for, f.eks. programmering av grafiske brukergrensesnitt. I slike tilfeller kan man finne tilleggsmoduler som har funksjonaliteten man trenger. Mange slike tilleggsmoduler er gratis.

Det medfølger et innebygget kjøremiljø som gjør at man kan kjøre både grafiske og terminalbaserte programmer i Eclipse. Man kan selv velge hvilket

underliggende JDK (Java Software Development Kit) eller JRE (Java Runtime Environment) dette kjøremiljøet skal bruke.

JDK er en programpakke fra Sun Microsystems (http://www.sun.com). Denne brukes bl.a. til å kompilere Java-kode samt å kjøre Java-programmer og tester. Det finnes flere variasjoner og utgaver, og den jeg har brukt er tilgjengelig på http://java.sun.com.

Som tidligere nevnt har ObjectStore et program som postprosesserer Java-bytekode. Dette programmet krever JDK 1.4.2 for å kjøre, og fungerer ikke mot bytekode kompilert av nyere JDK utgaver, og det er altså den jeg har hentet fra siden angitt over.

#### Apache Ant

Det finnes flere verktøy for å bygge Java-programmer. Jeg har valgt å bruke Apache Ant, ettersom jeg tidligere har brukt det sammen med JFlex, CUP og Eclipse. Ant er tilgjengelig gratis på http://ant.apache.org.

#### JUnit

JUnit er et rammeverk som brukes til å teste programmer skrevet i Java. Det kan lastes ned gratis fra http://www.junit.org. Jeg har valgt å bruke JUnit både på grunn av tidligere erfaring og fordi Eclipse har gratis tilleggsmoduler som fungerer sammen med JUnit.

## *3.2 Scanneren*

Når kildekode skal tolkes, er første steg vanligvis å dele opp teksten som utgjør denne kildekoden i en sekvens av leksemer. Et slikt leksem omfatter alle formene et symbol i språket (et nøkkelord, et literal eller liknende) kan ha.

Jeg bruker JFlex til å generere scanneren for OQL. I dette kapittelet ser vi på hvordan JFlex fungerer, og hvordan scanneren er implementert.

### 3.2.1 Scannergeneratoren JFlex

JFlex fungerer som andre scannergeneratorer ved at den benytter seg av en inputfil med definisjoner og innstillinger. Ut fra denne genererer JFlex kildekode for en scanner og en tilhørende symboltabell.

Inputfiler som brukes av JFlex er delt opp i tre deler. Den første delen består av egendefinert kode som skal brukes av scanneren. Den andre delen består

av innstillinger og navngitte regulære uttrykk. Den tredje delen består av regler for hva scanneren skal gjøre for hver leksem-type når den finner en.

Grunnelementene i et programmeringsspråk er gjerne definert som en samling av leksemer. Når disse settes i rekkefølge utgjør de setninger i språket. Eksempler på leksemer fra OQL er `select`, `array` og alle heltall.

Som andre scannergeneratorer kjenner JFlex igjen slike leksemer som sekvenser av tegn som godtas av regulære uttrykk. Disse regulære uttrykkene må man selv utforme i inputfila til JFlex.

Når tekst blir sendt til den ferdige scanneren, deles denne teksten opp i leksemer. For de fleste leksemer som kjennes igjen opprettes et `Symbol`-objekt, som parseren bruker til videre prosessering. Denne prosessen kalles leksikalsk analyse.

En gitt tekst kan inneholde sekvenser av tegn som ikke godtas av noen av de regulære uttrykkene man har definert. I så fall fører scanning av den gitte teksten til syntaksfeil.

Eksempler på navngitte regulære uttrykk man kan skrive i JFlex følger.

<table>
<tr><td>abc = ('a'|'b'|'c')</td><td>Dette uttrykket godtar en a, b eller en c. Uttrykket er navngitt abc, og kan senere brukes i andre regulære uttrykk eller til å kjenne igjen leksemer.</td></tr>
<tr><td>flereabc = ('a'|'b'|'c')*</td><td>Alle sekvenser bestående av a'er, b'er og c'er, samt den tomme sekvensen.</td></tr>
<tr><td>abcfirkant = [a|b|c]</td><td>En a, b eller en c. Ved bruk av [] istedenfor (), trenger man ikke å omslutte hvert tegn med anførselstegn.</td></tr>
<tr><td>nulltilni = [0-9]</td><td>Alle siffer fra 0 til 9. F.eks. vil 1, 2 og 3 godtas. 10 godtas ikke, siden det består av to siffer.</td></tr>
<tr><td>tallogbokstaver = [a-zA-Z0-9]+</td><td>Alle sekvenser bestående av siffer fra 0 til 9 og små og store bokstaver fra a til z.</td></tr>
<tr><td>tall = [0-9]*</td><td>Alle siffer fra 0 til 9 null eller flere ganger etter hverandre. 1423, 000 og 99999 er eksempler som godtas.</td></tr>
<tr><td>uttrykkmeduttrykk = {abc|nulltilni}*</td><td>Alle sekvenser av sekvensene som godtas av uttrykkene abc og nulltilni. 1423, 000, 9a9b9c9a9 og aaa er eksempler som godtas.</td></tr>
</table>

Når man har definert navngitte regulære uttrykk for leksemene, må man definere hva som skal gjøres for hver leksem-type scanneren kan kjenne igjen. Vi ser på følgende eksempel:

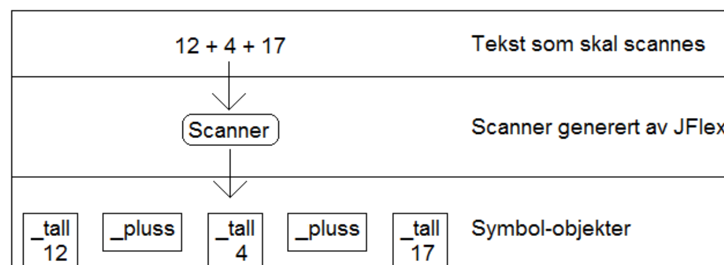
```
//navngitte regulære uttrykk
tall = [0-9]*
//regler for hva som skal gjøres når regulære uttrykk matches
{tall}    { return new Symbol(sym._tall, yytext());}
"+"       { return new Symbol(sym._pluss);}
```

De to reglene i dette eksemplet definerer hva scanneren skal gjøre med leksemer som godtas av de regulære uttrykkene `tall` og `"+"`. Uttrykket `"+"` har kun én form, og trenger derfor ikke å navngis slik som uttrykket `tall`. I begge tilfellene skal det opprettes et `Symbol`-objekt.

Symboltabellen JFlex genererer inneholder i dette tilfellet definisjoner av symbolene `_tall` og `_pluss`. Merk at alle `Symbol`-objekter er av samme klasse. Om et `Symbol`-objekt er av typen `_tall` eller `_pluss` indikeres av en medlemsvariabel.

Når den ferdige scanneren leser tekst som godtas av det regulære uttrykket `tall`, vil den generere et `Symbol`-objekt av typen `_tall`. Dette `Symbol`-objektet vil også inneholde teksten leksemet består av, som hentes ut ved et kall på `yytext()`.

Illustrasjon 8 viser resultatet av å scanne uttrykket `12 + 4 + 17` etter reglene definert over.




*Illustrasjon 8 : Fra tekst til `Symbol`-objekter*

Det oppstår ofte situasjoner hvor tekst godtas av flere regulære uttrykk samtidig. JFlex vil da bruke det regulære uttrykket som godtar lengst tekst.

Hvis to regulære uttrykk godtar like lang tekst, vil JFlex bruke det som står høyest opp i listen over uttrykk. I slike tilfeller er det viktig å tenke på rekkefølgen disse uttrykkene defineres i. Vi ser på følgende eksempel:

```
//navngitte regulære uttrykk
identifier = [a-zA-Z]+
//regler for hva som skal gjøres når regulære uttrykk matches
"if"         { return new Symbol(sym._if, yytext());}
{identifier} { return new Symbol(sym._identifier, yytext());}
```

Både uttrykket `identifier` og uttrykket `"if"` godtar teksten `if`. Ettersom regelen for `"if"` er definert før regelen for `{identifier}`, vil teksten `if` føre til at det opprettes et `Symbol`-objekt av typen `_if`. På denne måten kan nøkkelord i språket skilles fra variabelnavn.

### 3.2.2 Scanner for OQL

I denne implementasjonen heter inputfila til JFlex `oql.lex` (vedlagt i appendiks 6.2). Som utgangspunkt for denne har jeg brukt en obligatorisk oppgave fra kompilatorkurset ved UiO fra våren 2007. Vi skal nå se nærmere på de tre delene `oql.lex` består av.

#### *3.2.2.1 Egendefinert kode*

Den første delen av inputfila begynner med en deklarasjon av hvilken pakke den ferdige scanneren skal legges i, og hvilke biblioteker som skal importeres:

```
package org.odmg.impl;
import java_cup.runtime.*;
import com.odi.odmg.*;
```

Disse tre linjene utgjør første del av inputfila. For å markere at første del slutter og andre del begynner, skrives følgende på en egen linje:

```
%%
```

#### 3.2.2.2 Innstillinger og navngitte regulære uttrykk

Den andre delen av inputfila består av innstillinger og regulære uttrykk. Vi ser først på innstillingene:

```
%class Lexer
%public
%cup
%unicode
%line
%column
```

Den ferdige scanner-klassen skal hete `Lexer` og være `public`. Den skal fungere sammen med CUP, og være i stand til å scanne unicode-tekst.

Etterhvert som scanneren jobber seg igjennom en gitt tekst, skal den holde orden på hvilken linje og hvilken kolonne den har kommet frem til. Dette er for å kunne rapportere hvor en syntaksfeil er når scanneren kommer til tekst som ikke matcher noen leksemer.

Resten av del to består av navngitte regulære uttrykk, f.eks. `WhiteSpace`, `StringLiteral` og `TimeStampLiteral`.

De fleste av disse uttrykkene er definert i ODMG 3.0 på EBNF-form. EBNF brukes vanligvis til å definere hvordan sekvenser av leksemer kan se ut, men i ODMG 3.0 defineres også selve leksemene på denne formen. Vi ser for eksempel på definisjonen av strengliteraler (side 131 i [1]):

```
stringLiteral     ::= " { character } "
character         ::= letter
                    | digit
                    | special-character
letter            ::= A | B | ... | Z | a | b | ... | z
digit             ::= 0 | 1 | ... | 9
special-character ::= ? | _ | * | % | \
```

Disse reglene beskriver at strengliteraler består av tekst omgitt av anførselstegn. Reglene for de andre leksemene er i samme stil, men ikke fullt så omfattende.

ODMG 3.0 definerer altså ikke noe klart skille mellom hva som hører hjemme i scanneren og hva som hører hjemme i parseren. Jeg har derfor måttet skrive om noen av EBNF-reglene til regulære uttrykk. Vi ser nærmere på noen av disse uttrykkene.

`StringLiteral`-leksemet beskriver strenger. Disse består av null eller flere tegn, omgitt av to anførselstegn:

```
StringLiteral = "\"" [a-zA-Z0-9?_*%\\ ]* "\""
```

`WhiteSpace`-leksemet omfatter alle blanke tegn som forekommer mellom de andre leksemene. Slike leksemer er ikke definert i standarden, men er nødvendige som skille mellom andre leksemer. Blanke tegn innebærer mellomrom, tabulator, carriage-return, line-feed, new-line og kombinasjonen av carriage-return og new-line:

```
WhiteSpace = [ |\t|\f|\r|\n|\r\n]
```

`TimeLiteral`-leksemet beskriver klokkeslett. Disse består av apostrof, time, kolon, minutt, kolon, sekund og apostrof:

```
TimeLiteral = "'" [0-9]+ [:] [0-9]+ [:] ([0-9]+ |[0-9]+
"." [0-9]+)"'"
```

Sekunder kan godt skrives som flyttall (uten eksponent). Det er ikke noe i scanneren som hindrer brukeren i å skrive klokkeslett som ikke eksisterer. Man kan fint skrive `'25:74:93'` uten at scanneren oppfatter dette som syntaksfeil.

Scanneren teller heller ikke hvor mange siffer som skrives inn. Dermed kan man skrive f.eks. `'12345:12345:12345'` uten at scanneren oppfatter dette som en feil. Likevel vil slike feil oppdages når en slik streng blir forsøkt konvertert til et `java.sql.Time`-objekt.

Dette utgjør andre del av inputfila. For å markere at tredje del begynner, skrives følgende på en egen linje:

```
%%
```

#### 3.2.2.3 Handlinger for hvert leksem

Den tredje delen av inputfila består av regler for hva scanneren skal gjøre når den matcher et leksem. Det vanligste vil være at teksten leksemet består av skal pakkes inn i et `Symbol`-objekt som brukes av parseren til den videre prosesseringen.

Det kan også være at teksten leksemet består av må modifiseres før dette gjøres. For eksempel fjernes de omsluttende anførselstegnene fra et `Stringliteral`-leksem før det pakkes inn i et `Symbol`-objekt.

Ofte trenger ikke teksten leksemet består av å være med i Symbol-objektet. Dette gjelder alle leksemer som bare har én form, dvs. nøkkelord og operatorer.

I visse tilfeller skal scanneren *ikke* opprette et Symbol-objekt for leksemet den gjenkjenner. Mellomrom, linjeskift, tabulator o.l. blir oftest ignorert i scannere, bortsett fra som skille mellom leksemer.

De fleste av reglene gjelder nøkkelord og operatorer som f.eks. select, boolean, <=, + og –. For slike leksemer skal det opprettes Symbol-objekter som *ikke* inneholder teksten leksemet består av. For ordens skyld har jeg definert reglene i alfabetisk rekkefølge. De første reglene er:

```
"abs" { return new Symbol(sym._abs);}
"all" { return new Symbol(sym._all);}
"and" { return new Symbol(sym._and);}
```

abs, all og and er nøkkelord i OQL. For nøkkelordet abs opprettes et nytt Symbol-objekt av typen _abs.

Resten av reglene gjelder for navngitte regulære uttrykk. Her skal det stort sett opprettes Symbol-objekter med innhold. Vi ser nærmere på to av disse reglene:

```
{QueryParam} { return new Symbol(sym._queryparam, yytext());}
```

Når scanneren gjenkjenner et QueryParam-leksem, opprettes et Symbol-objekt av typen _queryparam som inneholder teksten leksemet består av.

Når scanneren kjenner igjen et WhiteSpace-leksem (mellomrom, linjeskift o.l.) skal den ikke gjøre noe. Handlingen for WhiteSpace er altså tom:

```
{WhiteSpace} { }
```

## 3.3 Parseren

En scanner produserer en sekvens av symboler ut fra en gitt tekst. Parseren undersøker om denne sekvensen følger en gitt grammatikk. I så fall kan symbolene i sekvensen utgjøre bladnoder i et *konkret syntakstre*.

Konkrete syntakstrær blir sjelden bygget, ettersom de gjerne har mye mer informasjon enn hva man trenger for å utføre den videre prosesseringen som skal gjøres.

Det er derfor vanlig å utelate mye av denne informasjonen og lage såkalte *abstrakte syntakstrær*, som kun inneholder den informasjonen man trenger.

Resultatet av en parsering sørger man som oftest for at er et slikt abstrakt syntakstre. Å gjøre om tekst til et abstrakt syntakstre kalles syntaktisk analyse.

### 3.3.1 Parsergeneratoren CUP

Før vi ser på parseren for OQL skal vi se nærmere på CUP, og som et eksempel se på hvordan man kan generere en veldig enkel parser med dette verktøyet.

En parser generert av CUP kan lett brukes sammen med en scanner generert av JFlex. Parseren vil analysere sekvensen av `Symbol`-objekter den får fra scanneren, og generere et abstrakt syntakstre ut fra regler brukeren har definert.

Vi ser tilbake på eksemplet fra JFlex, hvor regler for to regulære uttrykk blir definert med følgende linjer:

```
{tall}    { return new Symbol(sym._tall, yytext());}
"+"       { return new Symbol(sym._pluss);}
```

For disse uttrykkene skal det opprettes `Symbol`-objekter av typene `_tall` og `_pluss`. Symboltypene man har definert i JFlex må brukes med samme navn i CUP. Dette er fordi CUP bruker symboltabellen JFlex genererer.

De forskjellige symboltypene representerer terminaler i parseren, og defineres i inputfila til CUP på følgende måte:

```
terminal  String  _tall;
terminal          _pluss;
```

Terminalen `_tall` inneholder data i form av en streng. Denne strengen består av sekvensen av sifre som leses.

Terminalene settes videre sammen til ikke-terminaler ut fra BNF-grammatikken. Dermed kan man bygge opp setninger.

For å kunne lage alle additive uttrykk med heltall, definerer vi ikke-terminalen `ADDITIVT` og en enkel BNF-grammatikk:

```
non terminal  Uttrykk  ADDITIVT;

ADDITIVT
  ::= _tall:t
    {:RESULT=new Uttrykk(t);:}
    | ADDITIVT:a _pluss _tall:t
    {:RESULT=new Uttrykk(a, t);:}
  ;
```

Et ADDITIVT-uttrykk kan altså bestå av en _tall-terminal, eller av en ADDITIVT-ikke-terminal etterfulgt av terminalene _pluss og _tall. ADDITIVT-ikke-terminaler skal inneholde objekter av typen Uttrykk.

Parseren henter ut ett Symbol-objekt om gangen fra sekvensen som scanneren har generert, og tolker dette som en terminal. Dette kalles *shifting* (side 198 i [6]). Deretter undersøker den om en regel i BNF-grammatikken kan matches. I motsatt fall shifter den enda et Symbol-objekt.

Hvis parseren shifter en _tall-terminal, matches den øverste av de to reglene for ADDITIVT-uttrykket. Java-koden mellom {: og :} utføres mot bestanddelene av regelen, hvilket i dette tilfellet er Symbol-objektet som representerer _tall-terminalen.

Strengen i Symbol-objektet som representerer _tall-terminalen refereres til med bokstaven t, og blir sendt til konstruktøren i Uttrykk-klassen. En spesiell variabel ved navn RESULT settes til Uttrykk-objektet som opprettes. Dermed har man bygget en ikke-terminal av typen ADDITIVT ut fra en _tall-terminal.

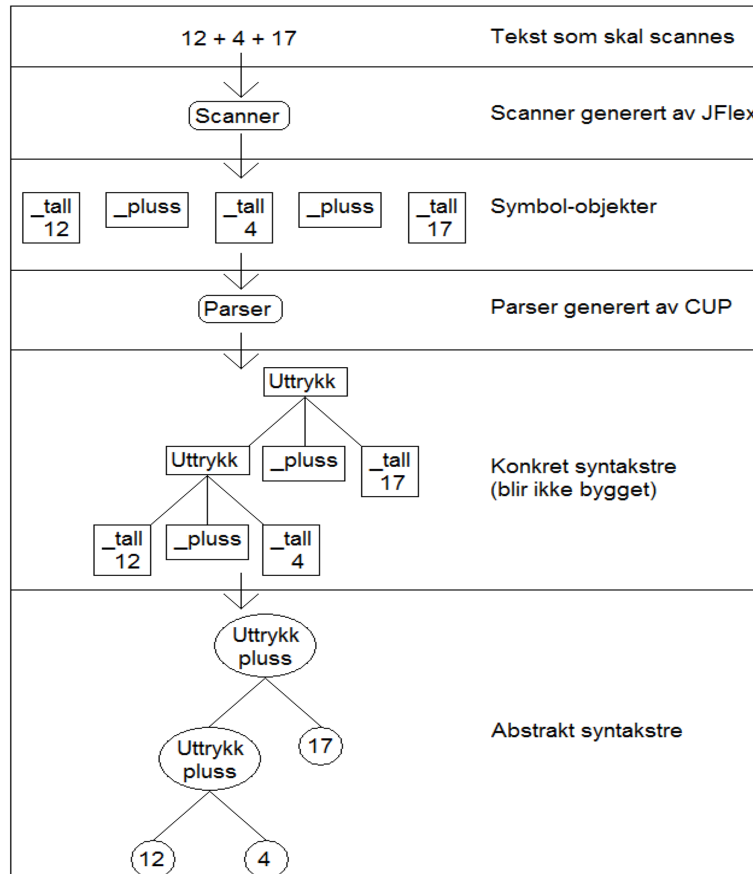
Dette ADDITIVT-uttrykket inneholder det nyopprettede Uttrykk-objektet, på samme måte som _tall-terminalen inneholdt en streng med sifre. Å bygge opp slike ikke-terminaler kalles *redusering* (se side 198 i [6]).

Hvis parseren shifter _pluss og _tall etter dette, kan parseren matche neste regel. Dermed kan det opprettes et nytt Uttrykk-objekt ut fra bestanddelene av ikke-terminalen ADDITIVT og terminalene _pluss og _tall. Dette Uttrykk-objektet blir til innholdet i en ny ADDITIVT-ikke-terminal.

Uttrykk er en Java-klasse som man må programmere selv. I dette enkle eksempelet kan den se slik ut:

```
class Uttrykk{
        int tall;
        Uttrykk leftChild;
        public Uttrykk(String tall){
                this.tall = Integer.parseInt(tall);
        }
        public Uttrykk(Uttrykk leftChild, String tall){
                this.tall = Integer.parseInt(tall);
                this.leftChild = leftChild;
        }
}
```

Ved hjelp av BNF-regler som i eksemplet over vil parseren trinnvis forsøke å bygge opp et konkret syntakstre. Hver gren av det konkrete syntakstreet som blir ferdig inneholder kun de objektene som skal være med i det abstrakte syntakstreet, ifølge Java-koden man har lagt til i hver BNF-regel. Illustrasjon 9 viser hele prosessen fra scanning av en tekst til et ferdig abstrakt syntakstre.




*Illustrasjon 9 : Fra tekst til abstrakt syntakstre*

## 3.3.2 Parser for OQL

Vi ser nå på hvordan parseren for OQL er implementert. For å lage denne parseren har jeg tatt utgangspunkt i en obligatorisk oppgave fra kompilatorkurset ved UiO fra våren 2007.

### 3.3.2.1 Java-kode i inputfila til parsergeneratoren

På samme måte som JFlex benytter CUP seg av en fil med diverse definisjoner, og ut fra denne fila kan CUP generere en parser. Denne fila heter `oql.cup` i denne implementasjonen, og begynner i vårt tilfelle med

følgende Java-kode:

```
package org.odmg.impl;
import org.odmg.impl.syntaxtree.*;
import java_cup.runtime.*;
```

Parseren skal ligge i samme pakke som resten av implementasjonen. Den skal bruke klassene i `org.odmg.impl.syntaxtree` når den bygger abstrakte syntakstrær. I tillegg skal den benytte noen av klassene som følger med CUP.

Videre defineres terminalene og ikke-terminalene som brukes i BNF-grammatikken til OQL. Etter definisjonen av disse kan man definere assosiativitet og presedens (side 117 i [6]). Dette er ikke nødvendig i vårt tilfelle, ettersom assosiativitet og presedens fremkommer av grammatikken.

#### 3.3.2.2 BNF-grammatikk og konkrete syntakstrær

Etter Java-koden i begynnelsen av inputfila, følger grammatikken til OQL. Ettersom denne grammatikken er definert som EBNF i ODMG 3.0, må den skrives om til ren BNF for å kunne brukes i CUP. Vi ser på en slik omskrivning:

```
queryProgram ::= declaration {; declaration} [; query]
               | query
```

Dette er hentet fra side 126 i ODMG 3.0-standarden ([1]). Det er kun semikolonene i første linje som er terminaler. Alt som står mellom "`{`" og "`}`" kan settes inn null eller flere ganger. Alt som står mellom "`[`" og "`]`" kan settes inn null eller én gang. Beskrivelsene av ikke-terminalene `declaration` og `query` er ikke vist.

Vi ser at et `queryProgram` kan settes sammen på en av tre måter:

1. Én eller flere ikke-terminaler av typen `declaration` adskilt av terminalen `;`
2. Én eller flere ikke-terminaler av typen `declaration` adskilt av terminalen `;`, etterfulgt av `;` og en ikke-terminal av typen `query`
3. Én ikke-terminal av typen `query`

Utdraget blir skrevet som BNF-grammatikk i CUPs syntaks på denne måten:

```
QUERYPROGRAM ::= DECLARATIONS
               | DECLARATIONS _semi QUERY
               | QUERY
               ;
DECLARATIONS ::= DECLARATION
               | DECLARATIONS _semi DECLARATION
               ;
```

Jeg har valgt å skrive ikke-terminaler med store bokstaver, og terminaler med små bokstaver og understrek foran. I dette eksempelet har jeg utelatt reglene for hvordan abstrakte syntakstrær skal bygges opp. Mer om abstrakte syntakstrær i kapittel 3.3.2.3.

Grammatikken er definert i ODMG 3.0 på en slik måte at man enklere forstår semantikken bak grammatikken. Jeg har valgt å skrive om grammatikken slik at den blir mer kompakt, og dermed enklere å arbeide med.

Noen av ikke-terminalene i EBNF-grammatikken hører hjemme i scanneren. Dette ble beskrevet i kapittel 3.2.2.2.

På side 92 i [1] er det eksempler på konstruksjon av structer og objekter i OQL. Disse eksemplene støttes merkelig nok ikke av grammatikken, hvilket må anses som en feil i ODMG 3.0. Vi ser nærmere på ett av eksemplene:

```
stats(select stat(a: age, s: sex)
          from Persons
          where name = "Pat")
```

Denne spørringen skal opprette et `stats`-objekt, som er et `DBag`-objekt som skal inneholde `stat`-structer. Definisjonen av `stats`-objektet og `stat`-structene gjøres i OQL, og finnes også på side 92 i [1], men er ikke vesentlig for denne diskusjonen.

Den delen av EBNF-grammatikken som utgjør reglene for oppretting av objekter er som følger:

```
objectConstruction ::= identifier ( fieldList )
fieldList          ::= field { , field }
field              ::= identifier : expr
```

Utdraget er hentet fra side 130 i [1]. Et select-uttrykk blir representert av ikke-terminalen `selectExpr`. I eksemplet over har vi altså et `objectConstruction`-uttrykk med et `selectExpr`-uttrykk i parentes.

Det lar seg ikke gjøre å parsere spørringen i dette eksemplet med denne grammatikken, ettersom ikke-terminalen `fieldList` ikke kan avledes til

selectExpr (se EBNF-grammatikk på side 130 i [1]).

Dermed må grammatikken endres slik at man kan bruke selectExpr i tillegg til fieldList mellom parentesene i objectConstruction-uttrykk. Endringen gjøres ved å endre regelen for objectConstruction til følgende:

```
objectConstruction ::= identifier ( fieldList )
                     | identifier ( selectExpr )
```

Dette skrives videre om fra EBNF til BNF i oql.cup. Dermed er dette problemet løst.

Ved omskrivningen fra EBNF til ren BNF oppstår det mange såkalte shift-reduser-konflikter og én såkalt reduser-reduser-konflikt. En shift-reduser-konflikt oppstår når det er riktig for parseren å både redusere med en gang, og å shifte inn flere Symbol-objekter, ut fra den gitte grammatikken. En reduser-reduser-konflikt oppstår når parseren kan matche flere regler samtidig, og dermed kan redusere på forskjellige måter.

Det er 57 slike konflikter til sammen. De fleste av shift-reduser-konfliktene følger av reduser-reduser-konflikten. Ingen av shift-reduser-konfliktene får konsekvenser, ettersom CUP alltid velger å shifte fremfor å redusere, hvilket forhindrer feilaktige reduksjoner i vårt tilfelle. Vi ser nærmere på den aktuelle reduser-reduser-konflikten.

Problemet kommer av at man kan bruke nøkkelordet distinct i OQL på to måter:

1. Som et frittstående uttrykk som fjerner duplikater fra et samlingsobjekt:

   ```
   distinct(bag(1,1,2,3))
   ```

2. Som en del av et select-uttrykk som returnerer et samlingsobjekt med Struct-objekter der hvert Struct-objekt er unikt i samlingen:

   ```
   select distinct p.biler, p.navn from person p
   ```

Konflikten oppstår når man skriver en spørring på denne formen:

```
select distinct (p.biler), p.navn from person p
```

I dette uttrykket kan distinct (p.biler) tolkes både som selvstendig uttrykk, og delt opp slik at distinct hører til det omliggende select-uttrykket og (p.biler) er et selvstendig uttrykk. Parseren kan altså redusere uttrykket på to gyldige måter, og har en reduser-reduser-konflikt.

Det står ingenting i standarden om presedens i denne sammenhengen. Jeg har valgt å la parseren tolke slike spørringer slik at `distinct` tilhører select-uttrykket. Jeg mener at det sjelden gir mening å få duplikater i resultatet av et select-uttrykk hvor `distinct` blir brukt.

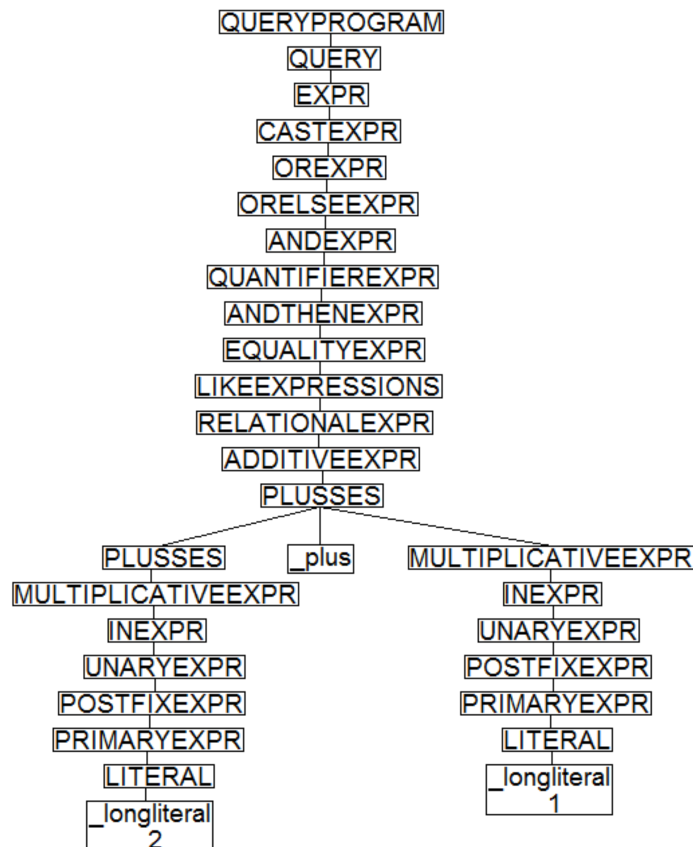
#### 3.3.2.3 Abstrakte syntakstrær

Vi skal her se på hvordan abstrakte syntakstrær blir bygget opp. Vi skal også se på klassene til objektene som brukes som noder i disse trærne.

Abstrakte syntakstrær inneholder som beskrevet tidligere mye mindre informasjon enn konkrete syntakstrær. Når man definerer hva man vil ha med i de abstrakte syntakstrærne velger man bort alt som ikke er vesentlig for den videre prosesseringen. Vi ser på et eksempel med et additivt uttrykk i OQL:

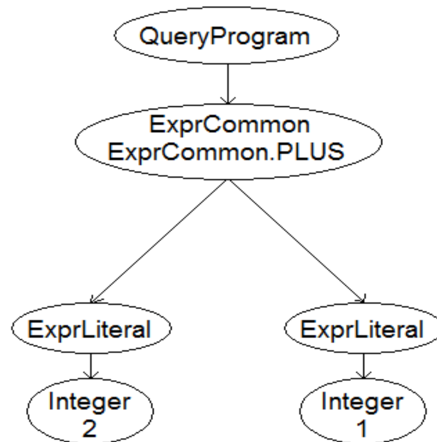
```
2 + 1
```

Et konkret syntakstreet for dette enkle regnestykket vises i illustrasjon 10.




*Illustrasjon 10 : Konkret syntakstre for uttrykket 2 + 1*

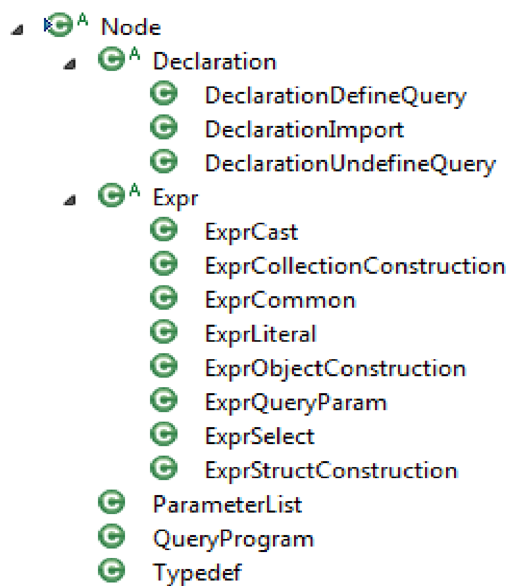
Ved å kun ta med nødvendig informasjon fra dette treet, sitter vi igjen med det abstrakte syntakstreet i illustrasjon 11, som er et tre av Java-objekter.




*Illustrasjon 11 : Abstrakt syntakstre for uttrykket 2 + 1*

`Integer`-objektene blir instansiert med innholdet fra `Symbol`-objektene som representerer `_longliteral`-terminalene. `ExprLiteral`-objektene stammer fra ikke-terminalen `LITERAL` i BNF-grammatikken.

`ExprCommon`-objektet peker på de to literalene, og har en variabel satt til verdien `ExprCommon.PLUS`, som betyr at operandene skal legges sammen. `QueryProgram` er alltid roten til hele det abstrakte syntakstreet.

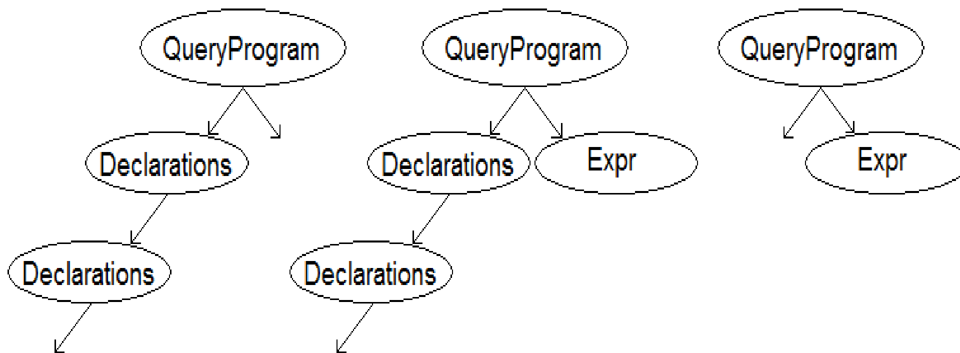



*Illustrasjon 12: Klassene til objektene som brukes til å bygge opp abstrakte syntakstrær. De som er merket med A er abstrakte klasser*

Illustrasjon 12 viser hierarkiet av klasser som beskriver objekter som kan brukes til å bygge opp abstrakte syntakstrær.

Alle klassene til objekter som brukes til å bygge opp et slikt tre er altså subklasser av den abstrakte klassen `Node`.

En spørring kan bestå av både deklarasjoner og et uttrykk, eller bare en av delene. Dette representeres i det abstrakte syntakstreet som vist i illustrasjon 13.




*Illustrasjon 13 : Et abstrakt syntakstre har en av disse tre formene*

Deklarasjoner representeres av objekter av subklasser til den abstrakte klassen `Declaration`. Det finnes tre typer deklarasjoner i OQL:

1. Den ene typen deklarasjon importerer Java-klasser, og binder disse til variabelnavn slik at de kan brukes i spørringer. Deklarasjoner av denne typen representeres av `DeclarationImport`-objekter.

2. Den andre typen definerer navngitte spørringer (vi så et eksempel på dette i kapittel 2.1). Slike navngitte spørringer skal ifølge ODMG 3.0 lagres i databasen de brukes i, slik at de kan brukes i andre spørringer senere.

   Dette er ikke implementert i denne oppgaven, men burde være forholdsvis greit å implementere. Deklarasjoner av denne typen representeres av `DeclarationDefineQuery`-objekter.

3. Den tredje typen sletter navngitte spørringer fra databasen. Deklarasjoner av denne typen representeres av `DeclarationUndefineQuery`-objekter.

   Ettersom navngitte spørringer ikke er implementert i denne oppgaven, er heller ikke funksjonaliteten for å fjerne slike lagrede spørringer implementert. Det burde være helt trivielt å implementere dette når definering og lagring av navngitte spørringer er gjort.

Navngitte spørringer kan ha parametere. Slike parametere blir representert av ParameterList-objekter i det abstrakte syntakstreet.

Et uttrykk representeres av et objekt av en subklasse av den abstrakte Expr-klassen. Vi ser nærmere på noen av disse klassene.

Literalene i OQL omsluttes av objekter av en egen klasse. Denne klassen heter ExprLiteral.

OQL-spørringer kan ha parametere på formen $1, $2, $3 osv. Slike representeres av ExprQueryParam-objekter.

ExprCommon er den klassen som holder på de fleste operatorene i OQL. Dette gjelder OQL-uttrykk for addisjon, subtraksjon, konvertering av lister til sett, fjerning av duplikater i samlingsobjekter osv.

OQL støtter typekasting av både primitiver og objekter. Typekasting representeres av ExprCast-objekter.

Ved typekasting angis det hvilken type et uttrykk skal kastes til. På samme måte defineres det hvilke typer objekter og literaler som godtas som argumenter til navngitte spørringer. Slike typer representeres av Typedef-objekter.

Man kan opprette samlingsobjekter av typen DCollection i spørringer. Et eksempel på et uttrykk som oppretter et slikt samlingsobjekt er list(1,2,4,3). Et slikt uttrykk representeres av et ExprCollectionConstruction-objekter.

Det går også an å opprette objekter i spørringer. Et slikt uttrykk blir representert av et ExprObjectConstruction-objekt.

Objektmodellen i ODMG 3.0 definerer primitiver og objekter som har en bestemt oppbygning. Det defineres også Struct-objekter, som har en friere oppbygning. Slike Struct-objekter skal kunne opprettes på nesten samme måte som andre objekter. Vi ser på to eksempler:

```
struct(navn:"Ola", fodselsnr: "14054512345")
struct(bilmerke:"Toyota", bilnummer:"DK35234")
```

Her opprettes to Struct-objekter med forskjellig oppbygning. De representeres av ExprStructConstruction-objekter.

Select-uttrykk kan ha flere former. De kan være med eller uten where-klausul, gruppering, sortering og duplikater. Alle formene til select-uttrykk representeres av ExprSelect-objekter.

For å få parseren til å bygge et abstrakt syntakstre av objekter av disse klassene, må man legge til Java-kode i BNF-grammatikken til CUP. Hvordan man gjør dette ble beskrevet i detalj i kapittel 3.3.1. Vi ser på noen eksempler:

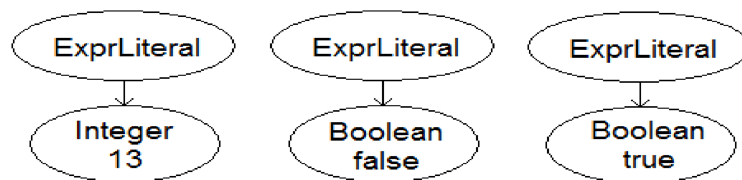
```
non terminal Expr LITERAL;
```

Denne regelen er definert i øverste del av inputfila. Ikke-terminalen `LITERAL` skal representeres som et `Expr`-objekt. Vi ser videre på et utdrag av BNF-grammatikken for ikke-terminalen `LITERAL`:

```
LITERAL
  ::= _true
    {:RESULT=new ExprLiteral(new Boolean("TRUE"));:}
    | _false
    {:RESULT=new ExprLiteral(new Boolean("FALSE"));:}
    | _longliteral:l
    {:RESULT=new ExprLiteral(new Integer(l));:}
    ...
```

`LITERAL` kan bl.a. være en av terminalene `_true`, `_false` og `_longliteral`. Hvis parseren kommer til en terminal av typen `_true` eller `_false`, opprettes det et nytt `Boolean`-objekt som pakkes inn i et `ExprLiteral`-objekt. Hvis parseren kommer til en terminal av typen `_longliteral` med verdien `13`, opprettes et `ExprLiteral`-objekt med et `Integer`-objekt med verdien `13`.

Ikke-terminaler av typen `LITERAL` skal representeres som `Expr`-objekter. Det er tilstrekkelig at de representeres som `ExprLiteral`-objekter, ettersom `ExprLiteral`-klassen er en subklasse av `Expr`-klassen. Illustrasjon 14 viser hvordan de tre eksemplene vi nettopp har sett på representeres i et abstrakt syntakstre.




*Illustrasjon 14 : Tre forskjellige `ExprLiteral`-objekter*

Vi ser på et mer sammensatt uttrykk:

```
non terminal Expr OREXPR;
```

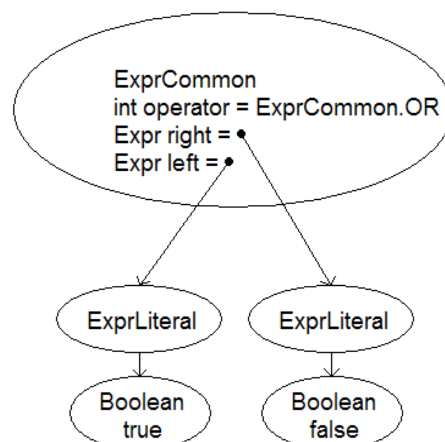
En ikke-terminal av typen `OREXPR` skal representeres av et `Expr`-objekt. Vi ser videre på et utdrag av BNF-grammatikken til `OREXPR`:

```
OREXPR
  ::= ORELSEEXPR:oe
    {:RESULT=oe;:}
    | ORELSEEXPR:oe _or OREXPR:o
    {:RESULT=new ExprCommon(oe, o, ExprCommon.OR);:};
```

To ting kommer frem av dette utdraget:

1. Et `OREXPR` kan avledes til et `ORELSEEXPR`. Det betyr at `OREXPR` kan avledes til f.eks. uttrykket `true orelse false`. Hvis man ser på hele grammatikken til OQL, ser man også at `ORELSEEXPR` kan avledes videre til ikke-terminalen `LITERAL`, som igjen kan avledes til primitiver.

2. Et `OREXPR` kan avledes til to uttrykk med operatoren `or` mellom. Hvis parseren kommer til f.eks. uttrykket `true or false`, blir det opprettet et nytt `ExprCommon`-objekt som holder på venstre- og høyredelen. Det blir også satt en variabel i `ExprCommon`-objektet som sier at operatoren i dette uttrykket er `or`. Det abstrakte syntakstreet for dette uttrykket er vist i illustrasjon 15.




*Illustrasjon 15 : Abstrakt syntakstre for uttrykket* `true or false`

## 3.4 Binding av parametere

Før en spørring kan eksekveres må eventuelle parametere i spørringen bindes til objekter. Dette gjøres ved å traversere det abstrakte syntakstreet rekursivt og å hente ut alle `ExprQueryParam`-objekter.

Jeg har valgt å legge disse objektene i en lenket liste for hvert av parameternumrene `$1`, `$2`, `$3` osv. Disse lenkede listene blir igjen lagt i et `HashMap`-objekt.

Vi ser på et eksempel:

```
select *
from person
where skatteprosent > $1
and lonn * $1 > $2
```

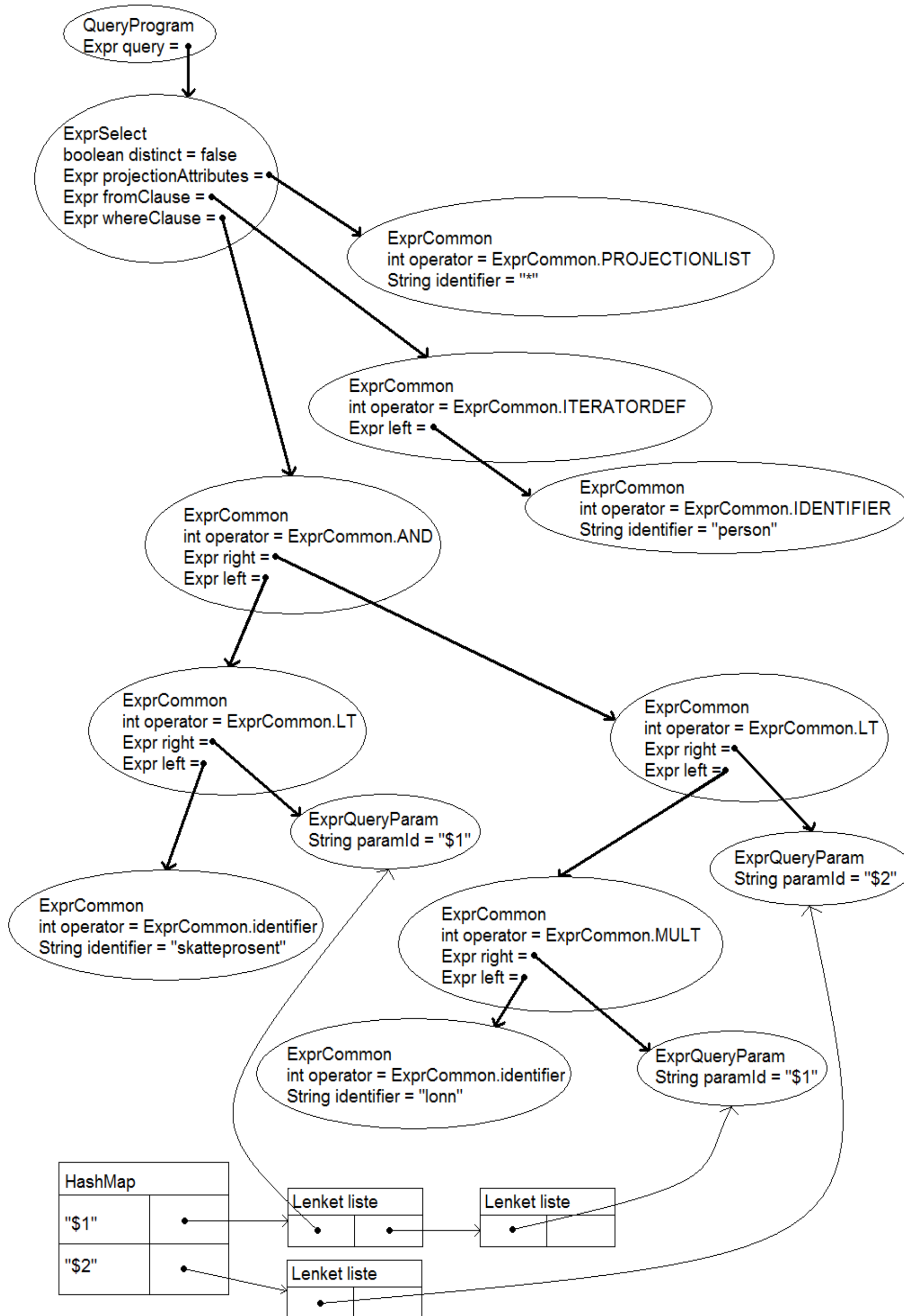
Her spør vi etter personer som har en skatteprosent over et visst nivå, og som samtidig betaler skatt over en viss sum. Etter å ha bygget opp det abstrakte syntakstreet og hentet ut alle `ExprQueryParam`-objektene, kommer vi frem til objektstrukturen vist i illustrasjon 16.

Det sjekkes deretter at samlingen av parametere inneholder alle parameternummer fra `$1` til `$n`, hvor `n` er et heltall. En spørring som f.eks. bare inneholder parametrene `$1`, `$2` og `$4` er ugyldig.

Når objekter skal bindes til parametrene i en spørring, hentes de lenkede listene ut i stigende rekkefølge. Anta at man skal binde `0.26` til den første parameteren i spørringen over. Man pakker inn primitivet `0.26` i et `Double`-objekt som man sender som argument til `bind()`-metoden i spørringen.

Ettersom dette er første kall på `bind()`, hentes den lenkede listen som peker på `ExprQueryParam`-objektene med `paramId="$1"`. Denne listen traverseres, og hvert av `ExprQueryParam`-objektene får `Double`-objektet bundet til seg.

Ved neste kall på `bind()`, hentes listen med `ExprQueryParam`-objekter med `paramId="$2"`, og blir behandlet på samme måte som forrige liste. Deretter kan man eksekvere spørringen. Etter spørringen er eksekvert, nullstilles parametrene. Spørringen kan brukes flere ganger, men man må binde objekter til parametrene igjen etter hver gang spørringen eksekveres.



*Illustrasjon 16 : Slik forberedes binding av objekter til parametere*

## 3.5 Semantisk sjekk og eksekvering av spørringer

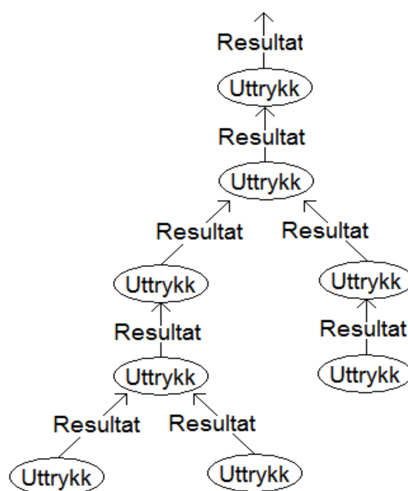
Etter at OQL-koden har blitt strukturert til et abstrakt syntakstre, og alle parametere har blitt bundet til objekter, er neste steg å eksekvere selve spørringen. Semantiske regler blir sjekket underveis.

Det første som skjer er at deklarasjoner av variable og navngitte spørringer blir behandlet. Disse blir lagt i et `HashMap`-objekt som hvert uttrykk får tilgang til. Slike `HashMap`-objekter holder på samlinger av navngitte objekter.

Når et uttrykk skal finne ut hvilket objekt en variabel refererer til, undersøker det først om variabelnavnet finnes i dette `HashMap`-objektet. Hvis det ikke finnes der, undersøkes det om det finnes et rot-objekt i databasen med dette navnet.

De fleste uttrykkene i det abstrakte syntakstreet arbeider kun mot de direkte tilkoblede uttrykkene ved hjelp av rekursive metoder som utføres i postfix rekkefølge. Noen uttrykk bruker iterative metoder. `HashMap`-objektet som holder på deklarerte variable o.l sendes med den rekursive metoden nedover i treet.

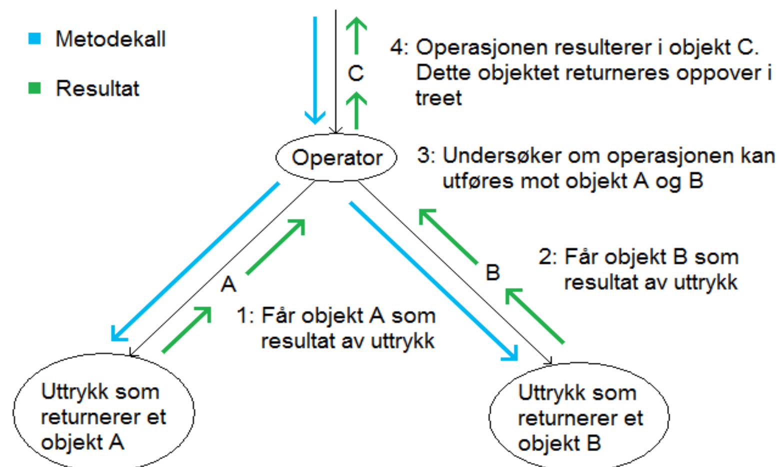
Når nye variable deklareres på vei nedover i treet, legges de til i `HashMap`-objektet. Hvis det defineres en ny variabel som har samme navn som en variabel som allerede ligger i `HashMap`-objektet, overskrives den gamle variabelen. Spørringer må dermed bruke unike variabelnavn for å få gyldige resultater.




*Illustrasjon 17 : Resultater av uttrykk flyter oppover i det abstrakte syntakstreet etterhvert som den rekursive metoden blir ferdig med å beregne uttrykkene*

Resultatene returneres oppover i det abstrakte syntakstreet etter hvert som den rekursive metoden fullfører beregninger (se illustrasjon 17).

Den semantiske sjekken utføres altså underveis i hvert uttrykk. Når et uttrykk skal beregne et resultat, hentes først resultatene av de underliggende uttrykkene. Deretter undersøkes det om operasjonen i uttrykket kan utføres med disse operandene. I så fall utføres operasjonen, som resulterer i et objekt. Dette objektet blir returnert tilbake til uttrykket over i det abstrakte syntakstreet. Illustrasjon 18 viser hele denne prosessen.




*Illustrasjon 18 : Typisk beregning av resultatet av et uttrykk*

Vi ser nærmere på hvordan de forskjellige typene uttrykk og deklarasjoner fungerer i detalj.

### 3.5.1 DeclarationImport

Man kan importere og bruke Java-klasser i OQL-spørringer. Dette er nødvendig for eksempel når man skal opprette objekter i spørringer.

Anta at en spørring skal benytte seg av funksjonalitet i `java.lang.Math`-klassen for å beregne cosinus av et tall. En slik spørring kan se slik ut:

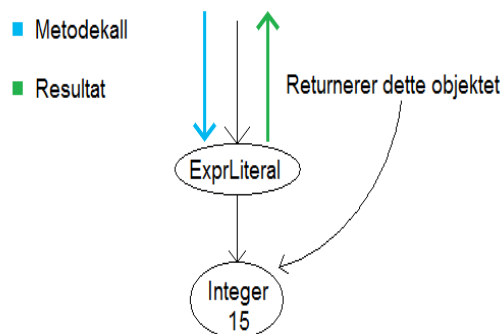
```
import java.lang.Math as Math; Math.cos(2.4)
```

Resultatet av denne spørringen blir et `Double`-objekt med verdien av `cosinus(2.4)`. Merk at klasser ikke kan importeres for fremtidige spørringer. De må importeres i alle spørringer de skal brukes i.

### 3.5.2 ExprLiteral

Et literal utgjør et eget uttrykk som representeres av et `ExprLiteral`-objekt som peker på objektet som representerer literalet. Når den rekursive metoden kommer til et `ExprLiteral`-objekt, returneres objektet som representerer literalet.

Hvis vi har et heltall-literal, f.eks. `15`, er dette pakket inn i et `Integer`-objekt. Dette `Integer`-objektet returneres oppover i treet som resultat av uttrykket (se illustrasjon 19).




*Illustrasjon 19 : Beregning av resultatet av et `ExprLiteral`-objekt*

Et literal kan være en streng, et tegn, et heltall (flere varianter), et flyttall (flere varianter), et tidspunkt, et klokkeslett, en dato, udefinert eller `null`. Hvis literalet er udefinert, returneres et `Undefined`-objekt. `Undefined` er en klasse i implementasjonen. Hvis literalet er `nil`, returneres Java-verdien `null` uten å bli pakket inn i et objekt.

### 3.5.3 ExprObjectConstruction

Man kan instansiere transiente objekter i spørringer. Denne funksjonaliteten er implementert ved å bruke *Java-reflection-API* ([7] og [8]). Anta at man skal opprette et `Person`-objekt. Det komplette navnet til `Person`-klassen er `eksempel.Person`. Man begynner med å importere denne klassen med følgende deklarasjon i OQL-spørringen:

```
import eksempel.Person as Person;
```

Ut fra denne deklarasjonen vil kompilatoren legge `Class`-objektet til `Person`-klassen i `HashMap`-objektet hvor alle deklarerte variable ligger. Dermed kan uttrykk referere til dette `Class`-objektet med navnet `Person`.

Anta at `Person`-klassen ser slik ut:

```
class Person{
    public String fornavn;
    public int alder;
}
```

Merk at medlemsvariablene må være deklarert som `public` for at Java-reflection-API skal ha tilgang til til disse. OQL-spørringen kan da fortsette med følgende uttrykk:

```
Person(fornavn: "Ola", alder: 43)
```

Dette uttrykket blir representert i det abstrakte syntakstreet av et `ExprObjectConstruction`-objekt. Det første som skjer under evalueringen av dette uttrykket er at `Class`-objektet som representerer `Person`-klassen blir hentet ut fra `HashMap`-objektet som holder på deklarerte variable.

Ved å bruke Java-reflection-API opprettes et tomt `Person`-objekt via dette `Class`-objektet. Det må derfor finnes en konstruktør uten argumenter i `Person`-klassen, og alle andre klasser som det skal opprettes objekter av.

Videre hentes hvert av parene variabelnavn og variabelverdi ut av argument-listen. Første par består av variabelnavnet `fornavn` og verdien `"Ola"`. Strengen `"Ola"` blir lagt i medlemsvariabelen `fornavn` i det nye `Person`-objektet, igjen ved bruk av reflection. Dette kan kun gjøres hvis medlemsvariabelen `fornavn` er av typen `String`. Dermed blir typesjekkingen i slike uttrykk utført av Java-reflection-API.

Videre legges heltallet `43` i medlemsvariabelen `alder`. Ettersom dette utgjør hele lista over argumenter, er `Person`-objektet ferdig laget, og kan returneres som et resultat av uttrykket.

### 3.5.4 ExprStructConstruction

Structer er en egen type objekter som brukes i mange tilfeller i implementasjonen. De kan opprettes eksplisitt i OQL-spørringer ved hjelp av egne uttrykk. Slike uttrykk representeres av `ExprStructConstruction`-objekter i det abstrakte syntakstreet.

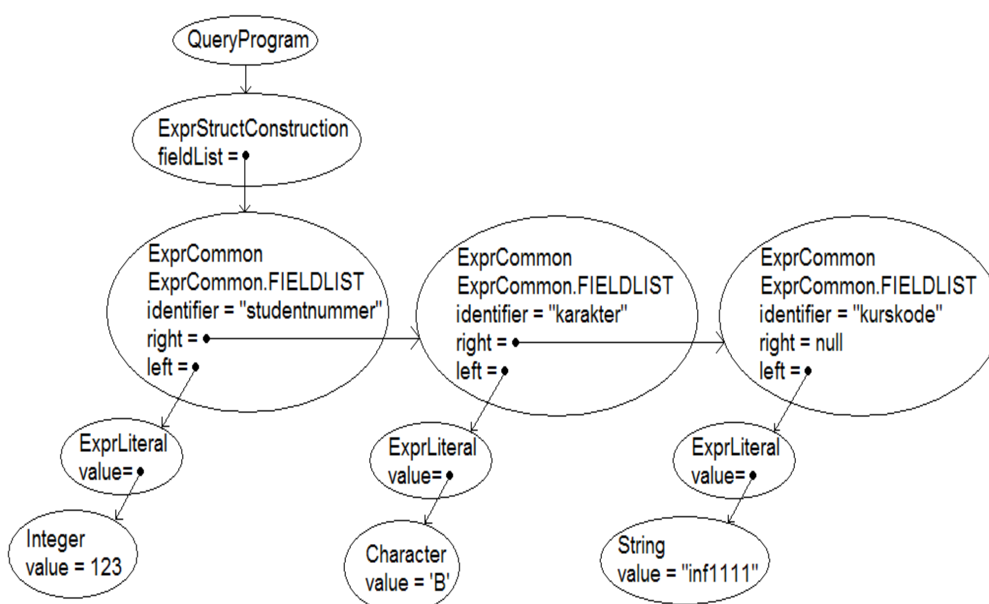
Hvis man f.eks. skal opprette et `Struct`-objekt med feltene `studentnummer`, `karakter` og `kurskode`, kan man skrive følgende OQL-spørring:

```
struct(studentnummer: 123, karakter: 'B', kurs: "inf1111")
```

Etter at dette uttrykket er scannet og parsert, har parseren laget det abstrakte syntakstreet i illustrasjon 20.

Ved beregning av resultatet av denne spørringen utføres et rekursivt metodekall som jobber seg ned til `ExprStructConstruction`-objektet. Dette objektet holder på en liste av navngitte uttrykk (`ExprCommon`-objektene).

`ExprStructConstruction`-objektet itererer igjennom listen for å finne ut lengden til denne. I dette eksempelet ser vi at listen består av tre navngitte uttrykk.




*Illustrasjon 20 : Abstrakt syntakstre for uttrykket `struct(studentnummer: 123, karakter: 'B', kurskode: "inf1111")`*

Det opprettes så to vektorer av samme lengde som listen. Disse skal holde på navnene og verdiene til feltene i det ferdige `Struct`-objektet. Navn og verdier legges inn i disse vektorene ved å iterere igjennom listen av uttrykk en gang til. De to ferdige vektorene er vist i illustrasjon 21.

| String[] navn: |
|---|
| "studentnummer" |
| "karakter" |
| "kurskode" |

| Object[] verdier: |
|---|
| Integer(123) |
| Character('B') |
| String("inf1111") |

*Illustrasjon 21 : Vektorer som brukes til å opprette et `Struct`-objekt*

Disse to vektorene sendes som argumenter til konstruktøren i Struct-klassen. Dermed opprettes et Struct-objekt som returneres oppover til QueryProgram-objektet, som videre returnerer dette objektet som svar på spørringen.

### 3.5.5 ExprCollectionConstruction

Man kan opprette samlingsobjekter i spørringer ved hjelp av egne uttrykk. Et slikt uttrykk representeres av et ExprCollectionConstruction-objekt i et abstrakt syntakstre.

Man kan kun opprette samlingsobjekter av typene DSet, DBag, DArray og DList ved hjelp av disse uttrykkene. De kan opprettes med eller uten innhold. Vi ser på noen eksempler på slike OQL-uttrykk:

```
set()
```

Her opprettes et tomt DSet-objekt.

```
bag('a','b','c','a')
```

Her opprettes et DBag-objekt som inneholder {'a','b','c','a'}

```
list(1 .. 5)
```

Her opprettes et DList-objekt som inneholder {1,2,3,4,5}.

Som de fleste andre uttrykk i OQL kan man bruke uttrykk som deler av større uttrykk:

```
array( 1 + 2, (4 * 3) / 6 , 3 + 4)
```

Her opprettes et DArray-objekt med verdiene {3,2,7}.

Etter at et samlingsobjekt har blitt opprettet, returneres det oppover i treet som resultat av uttrykket.

### 3.5.6 ExprQueryParam

Parameterne i OQL-spørringer representeres av ExprQueryParam-objekter i det abstrakte syntakstreet. Ved binding av et objekt til en parameter kobles dette objektet til alle ExprQueryParam-objektene som representerer denne parameteren.

Under eksekveringen av spørringen returnerer et ExprQueryParam-objekt det bundne objektet som resultat. Dette fungerer på samme måte som med

ExprLiteral-objekter.

### 3.5.7 ExprSelect

Et select-uttrykk representeres av et ExprSelect-objekt i et abstrakt syntakstre. Et slikt uttrykk har et sett av klausuler og en innstilling som sier om duplikater er tillatt i resultatet av uttrykket. Denne innstillingen og klausulene utgjør trinn i beregningen. Resultatet av uttrykket er det samlingsobjektet man sitter igjen med etter alle trinnene er utført.

Først undersøkes det om duplikater skal fjernes fra resultatet, dvs. om uttrykket begynner med select distinct. I så fall tas det utgangspunkt i at resultatet skal være et DSet-objekt (dette kan endres til DList senere i beregningen hvis uttrykket skal sorteres). I motsatt fall tas det utgangspunkt i at resultatet skal være et DBag-objekt.

Dette samlingsobjektet blir først fylt med det kartesiske produktet av samlingsobjektene i from-klausulen. Hvert objekt i samlingsobjektet er et Struct-objekt.

Hvis select-uttrykket har en where-klausul, sendes samlingsobjektet fra from-klausulen videre til denne. Where-klausulen inneholder et boolskt uttrykk som beregnes mot alle Struct-objektene i samlingsobjektet. Hvis uttrykket resulterer i false eller undefined, blir det aktuelle Struct-objektet fjernet fra samlingsobjektet.

Etter samlingsobjektet har vært igjennom from-klausulen og eventuelt where-klausulen sendes det videre til select-klausulen. Her blir alle Struct-objektene endret slik at de inneholder de feltene som er spesifisert i denne klausulen.

De to siste klausulene er group-by-klausulen og order-by-klausulen. Disse er ikke implementert. Group-by-klausulen ble for omfattende å implementere i forhold til tiden til rådighet, og order-by-klausulen skal fungere i select-uttrykk både med og uten gruppering.

Resultatet av uttrykket er dermed samlingsobjektet som har vært igjennom select-klausulen, from-klausulen og eventuelt where-klausulen.

### 3.5.8 ExprCast

Typekasting av et uttrykk representeres av et ExprCast-objekt i et abstrakt syntakstre. Et slikt ExprCast-uttrykk består av uttrykket som skal kastes og typen det skal kastes til.

Hvordan resultatet av typekasting beregnes er avhengig av hva slags uttrykk som skal typekastes og til hva slags type. Vi ser først på typekasting av primitiver.

Hvis et primitivt heltall skal typekastes til et primitivt flyttall, gjøres dette ved å opprette et nytt flyttall-objekt for så å gi dette verdien til heltallet. Man kan også gjøre om flyttall til heltall, men da går muligens noe presisjon tapt. Å typekaste tall til strenger lar seg også gjøre ved at verdien til tallet konkateneres med en tom streng.

Å gjøre om strenger eller tall til tegn lar seg ikke gjøre direkte. Man kan derimot gjøre om hva som helst til en streng, og bruke `charAt()`-metoden i `java.lang.String`-klassen for å hente ut første tegn i denne strengen. Denne metoden kan brukes direkte i en OQL-spørring.

Å typekaste et objekt fra en type til et annet gjøres ved å returnere hele `ExprCast`-objektet som resultatet av `ExprCast`-uttrykket. Det gjøres på denne måten ettersom det ikke har noen effekt å kaste et objekt av type `A` til type `B` for så å returnere det som et generelt Java-objekt.

Når et uttrykk i OQL får et `ExprCast`-objekt som resultat fra et av uttrykkene under, kan det behandle innholdet i dette `ExprCast`-objektet som en instans av typen `ExprCast`-objektet definerer. Vi ser på et eksempel.

Anta at man har en klasse `Person` og en klasse `Ansatt`, som er en subklasse av `Person`. `Person` har en metode `toString()`, som gjør om et `Person`-objekt til en formatert streng. Denne metoden blir redefinert i `Ansatt`-klassen.

Anta at det eksisterer et rot-objekt av typen `Ansatt` i en database, og at dette objektet er navngitt `Ola`. Hvis man skal bruke `toString()`-metoden i `Person` mot dette `Ansatt`-objektet, så skriver man følgende uttrykk:

```
((Person)Ola).toString()
```

Man får da et `ExprCast`-uttrykk hvor typen det skal kastes til er `Person`, og objektet som skal typekastes er rot-objektet `Ola`. Dette `ExprCast`-uttrykket returnerer seg selv som resultat til dot-notasjon-uttrykket rundt.

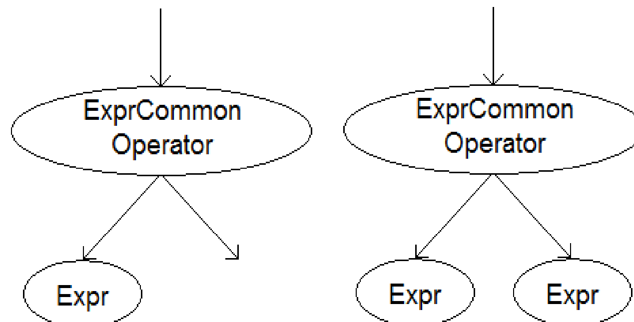
Dette dot-notasjon-uttrykket ser at det har fått et `ExprCast`-uttrykk som resultat, og behandler rot-objektet `Ola` som et `Person`-objekt. Dette gjøres ved å bruke Java-reflection-API med klassen `Person` istedenfor klassen `Ansatt` mot dette objektet.

### 3.5.9 ExprCommon

De aller fleste uttrykkene i OQL blir representert av `ExprCommon`-objekter. Mange uttrykk har en av de følgende to formene:

1. `operator uttrykk`
2. `uttrykk operator uttrykk`

Disse uttrykkene blir representert som vist i illustrasjon 22.




*Illustrasjon 22 : `ExprCommon`-objekter har vanligvis en av disse formene*

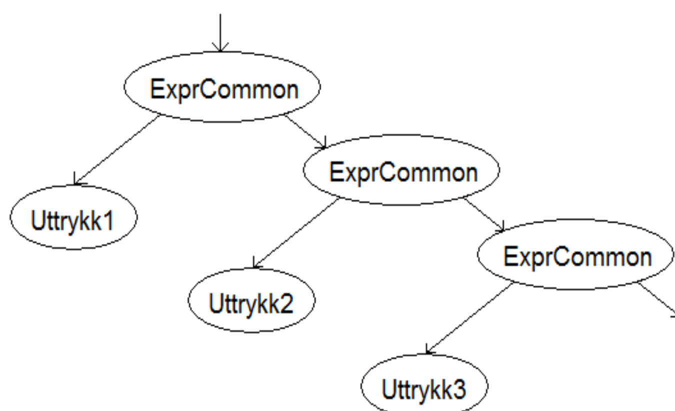
Resultatet av slike uttrykk blir beregnet rett frem ved at operatoren i uttrykket blir brukt mot resultatene av de underliggende uttrykkene. Dette resulterer i et objekt som returneres oppover i det abstrakte syntakstreet.

`ExprCommon`-objekter brukes også til å representere lister på en av de følgende formene:

1. `uttrykk1, uttrykk2, uttrykk3`
2. `navn1: uttrykk1, navn2: uttrykk2`

Den første formen av lister blir representert som vist i illustrasjon 23. Den andre formen blir representert på tilsvarende måte.

Slike lister blir stort sett traversert iterativt istedenfor rekursivt. `ExprStructConstruction`- og `ExprObjectConstruction`-objekter benytter seg av slike lister.

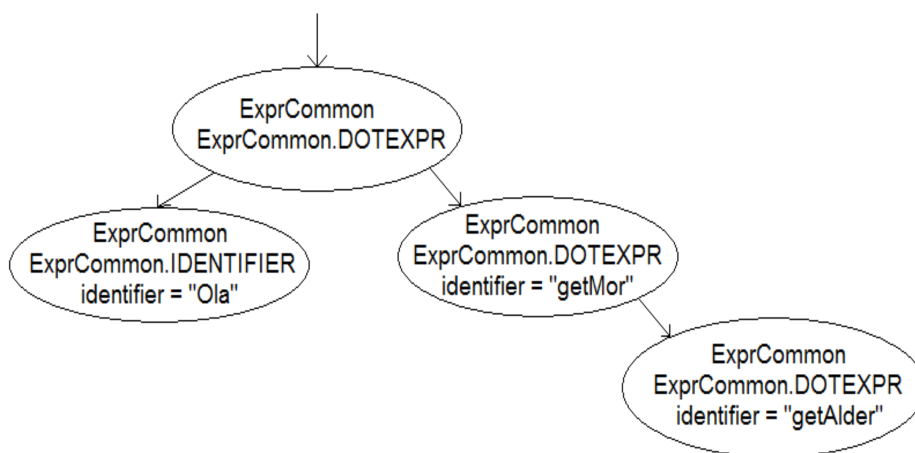



*Illustrasjon 23 : `ExprCommon`-lister representeres på denne måten*

`ExprCommon`-objekter representerer også uttrykk for å hente ut rot-objekter og uttrykk med dot-notasjon. Anta et `Person`-objekt `Ola`, og at `Person`-klassen har en metode `getMor()` og en metode `alder()`. For å beregne alderen til moren til `Ola`, kan man skrive:

```
Ola.getMor.alder()
```

Det er valgfritt å skrive parenteser etter metodekall når metoden ikke har parametere. Dette OQL-uttrykket representeres som vist i illustrasjon 24.




*Illustrasjon 24 : Dot-notasjon representert av `ExprCommon`-objekter*

Først hentes rot-objektet `Ola` ut av databasen, deretter kalles `getMor()`-metoden på dette objektet. Resultatet av kallet på `getMor()`-metoden er et nytt `Person`-objekt. Metoden `getAlder()` blir kalt for dette objektet. Resultatet blir et heltall som returneres som resultat av hele uttrykket.

### *3.6 Implementasjon av grensesnittene i Java-bindingen*

De foregående kapitlene har beskrevet hvordan OQL-tolkeren har blitt implementert og hvordan resultater av uttrykk beregnes. Vi går nå videre til å se på hvordan Java-grensesnittene fra ODMG 3.0 er implementert, og hvordan OQL-tolkeren passer inn i noen av disse implementasjonene.

Transaksjonsmodellen er vanskelig å implementere i henhold til standarden. Jeg har valgt å forenkle denne delen av implementasjonen, ettersom hovedmålet med oppgaven er å implementere funksjonalitet i OQL.

Klassene som implementerer de forskjellige grensesnittene har fått prefikset OS, som er kort for ObjectStore.

#### 3.6.1 Implementasjon av Database-grensesnittet

`Database`-grensesnittet implementeres av `OSDatabase`-klassen. Denne klassen har en medlemsvariabel `db` av typen `com.odi.Database` fra OSJI, som tar seg av tilkoblinger mot ObjectStore-databaser.

Metodene `bind()`, `unbind()` og `lookup()` har som oppgave å opprette, slette og gjenfinne rot-objekter. De implementeres rett frem ved kall på `db.createRoot()`, `db.destroyRoot()` og `db.getRoot()`. Når et objekt blir forsøkt bundet til et navn som allerede er i bruk, eller når et rot-objekt hvis navn ikke finnes skal gjenfinnes eller ødelegges, kastes et unntak fra ODMGs unntakshierarki (se appendiks 6.5).

Metoden `open()` har som oppgave å koble `Database`-objektet til en database. Først undersøkes det at `Database`-objektet ikke allerede er tilkoblet en database. Deretter blir den kallende Java-tråden koblet til en ny sesjon hvis den ikke allerede er tilkoblet en annen sesjon. Til sist blir databasen forsøkt åpnet i lesemodus, skrivemodus eller eksklusivt modus, avhengig av hvilke argumenter som ble sendt med `open()`-metoden.

Metoden `close()` kaller videre på `close()` i den underliggende medlemsvariabelen `db`. Deretter settes `db` til Java-verdien `null`, og sesjonen som tråden er tilkoblet blir avsluttet.

Metodene `makePersistent()` og `deletePersistent()` legger objekter i og fjerner objekter fra ekstensjoner. Når et objekt skal lagres, opprettes `OSHashSet`-objekter som blir ekstensjonene til klassen og hver superklasse som objektet er med i (unntatt klassen `java.lang.Object`).

Anta at vi har en klasse `eksempel.Arbeidstaker` som er en subklasse av `eksempel.Person`, som igjen er en subklasse av `java.lang.Object`. Følgende skjer når man skal lagre et `Arbeidstaker`-objekt ved et kall på `makePersistent()`:

1. Det undersøkes om et `OSHashSet`-objekt for ekstensjonen til `Arbeidstaker`-klassen allerede eksisterer. I så fall er det lagret som et rot-objekt ved navn `arbeidstaker` i databasen. Jeg har valgt å la navnet til ekstensjonen være siste ledd i klassenavnet til objektet som skal lagres, gjort om til små bokstaver. Hvis det ikke finnes, opprettes et nytt `OSHashSet`-objekt som bindes til dette navnet.

2. Etter punkt 1. vil det uansett finnes et `OSHashSet`-objekt for `Arbeidstaker`-klassen. `Arbeidstaker`-objektet blir lagt til i dette `OSHashSet`-objektet.

3. Ettersom objekter som er med i `Arbeidstaker`-klassen også er med i `Person`-klassen, skal `Arbeidstaker`-objektet også legges i ekstensjonen til `Person`. Dette gjøres ved å gjenta punkt 1. og 2. for `Person`-klassen.

4. `Person`-klassen er en direkte subklasse av `java.lang.Object`, som ikke skal ha en ekstensjon. Dermed har `Arbeidstaker`-objektet blitt lagret i alle ekstensjoner det skal lagres i.

På tilsvarende måte slettes objekter fra ekstensjoner ved kall på `deletePersistent()`. Hvis ekstensjonen til klassen til et objekt ikke finnes, så har ikke objektet blitt lagret før. `deletePersistent()` returnerer så fort den ikke finner en ekstensjon den leter etter.

Hvis ODL hadde vært en del av implementasjonen, ville `makePersistent()` og `deletePersistent()` blitt implementert annerledes. Ekstensjoner ville da blitt definert i ODL, og `makePersistent()` og `deletePersistent()` ville legge objekter til og slette objekter fra disse ekstensjonene. Det ville altså ikke bli opprettet nye ekstensjoner automatisk ved kall på `makePersistent()`.

### 3.6.2 Implementasjon av Transaction-grensesnittet

`Transaction`-grensesnittet implementeres av `OSTransaction`-klassen.

Transaksjonsmodellen i ObjectStore er annerledes enn transaksjonsmodellen i ODMG 3.0-standarden, hvilket fører til at transaksjoner ikke er

implementert i henhold til ODMG 3.0-standarden.

Hovedproblemet er at ObjectStore bruker sesjoner til å organisere både transaksjoner og databasetilkoblinger. Hvis to tråder er tilkoblet samme sesjon, og en av disse trådene starter en transaksjon, vil den andre tråden automatisk tilkobles denne transaksjonen.

Dermed er det fristende å starte alle transaksjoner i hver sin sesjon, slik at ingen tråder tilkobles transaksjoner implisitt. Men ettersom tilkoblinger til databaser er bundet til sesjoner, vil slike tilkoblinger brytes dersom alle transaksjoner startes opp i en ny sesjon.

En tenkt løsning på dette problemet er å lagre informasjon om databasetilkoblinger utenfor sesjoner, slik at disse kan åpnes bak kulissene når en sesjon starter. Dette ville muliggjøre at transaksjoner kan startes i hver sin sesjon.

Jeg har valgt å ikke implementere denne løsningen, ettersom hovedmålet i oppgaven er å implementere funksjonalitet i OQL. Dermed fungerer implementasjonen av transaksjoner litt annerledes enn hva som er definert i ODMG 3.0.

Når en tråd forsøker å koble seg til transaksjonen til en annen tråd, kobler den seg til sesjonen denne transaksjonen er koblet til. Dermed er disse to trådene koblet til samme sesjon, og vil dele fremtidige transaksjoner.

Når en tråd kobler seg fra en transaksjon, vil den koble seg fra sesjonen til denne transaksjonen. Dermed mister den også åpne databasetilkoblinger.

Det finnes også to metoder for å låse objekter til en transaksjon. Låsing foregår som oftest implisitt, men kan også foregå eksplisitt ved hjelp av disse metodene. Den ene metoden returnerer `true` eller `false`, ettersom låsingen gikk bra eller ikke. Den andre metoden returnerer ingenting, men kaster et kjøretids-unntak hvis låsingen ikke ble gjennomført.

Sammenlikning av `OSTransaction`-objekter gjøres ved kall på `equals()`. Denne metoden sjekker at to `OSTransaction`-objekter er koblet til samme underliggende ObjectStore-transaksjon.

### 3.6.3 Implementasjon av Implementation-grensesnittet

`Implementation`-grensesnittet implementeres av `OSImplementation`-klassen.

Metodene `newDArray()`, `newDList()`, `newDBag()`, `newDSet()`, `newDMap()`, `newOQLQuery()`, `newDatabase()` og `newTransaction()` er såkalte factory-metoder. De oppretter og returnerer objekter av klasser som implementerer de respektive grensesnittene. For eksempel vil `newTransaction()` opprette og returnere et `OSTransaction`-objekt i denne implementasjonen.

Metoden `currentTransaction()` undersøker om en transaksjon er igang for den kallende tråden. Hvis ingen transaksjoner er igang, returneres Java-verdien `null`. Hvis en transaksjon er igang, blir det underliggende `com.odi.Transaction`-objektet for denne transaksjonen pakket inn i et nytt `OSTransaction`-objekt som returneres.

Merk at dette innebærer at følgende uttrykk resulterer i `false`:

```
currentTransaction()==currentTransaction()
```

Dette er fordi det underliggende `com.odi.Transaction`-objektet pakkes i et nytt `OSTransaction`-objekt hver gang metoden kalles. `OSTransaction`-objekter skal derfor som oftest sammenliknes ved å bruke `equals()`-metoden.

Metoden `getDatabase()` undersøker hvilken ObjectStore-database et objekt tilhører. Hvis objektet tilhører en database, vil OSJI finne frem det underliggende `com.odi.Database`-objektet som representerer en tilkobling til denne databasen. Dette `com.odi.Database`-objektet pakkes inn i et `OSDatabase`-objekt som returneres.

Hvis et objekt er lagret i en database, kan man få en strengrepresentasjon av adressen objektet har i denne databasen ved å kalle `getObjectId()`-metoden.

### 3.6.4 Implementasjon av OQLQuery-grensesnittet

`OQLQuery`-grensesnittet implementeres av `OSOQLQuery`-klassen. Denne klassen har en medlemsvariabel av typen `Interpreter`, som er selve OQL-tolkeren.

Det er tre metoder i `OSOQLQuery`-klassen. Alle disse benytter seg av `Interpreter`-medlemsvariabelen:

1. Metoden `create()` bygger opp et abstrakt syntakstre av en OQL-spørring. Dermed blir syntaksen i spørringen nødvendigvis verifisert.

2. Metoden `bind()` binder et objekt til en parameter.

3. Metoden `execute()` eksekverer spørringen. Det første som skjer er at det sjekkes at alle definerte parametere er bundet til et objekt. Deretter beregnes resultatet av spørringen. Videre nullstilles parametrene i OQL-spørringen, slik at disse kan bindes til nye objekter ved neste spørring. Til sist returneres resultatet av spørringen.

### 3.6.5 Implementasjoner av DCollection-grensesnittene

ODMG 3.0 definerer Java-grensesnitt for fem samlingsklasser. Fire av disse utvider `DCollection`-grensesnittet. Implementasjonene til disse fire heter:

- `OSDArray`
- `OSDList`
- `OSDSet`
- `OSDBag`

Alle disse implementasjonene utvider samlingsklasser fra OSJI.

`DCollection` definerer fire metoder som omfatter OQL-spørringer. For hver spørring som utføres, opprettes et `Interpreter`-objekt som beregner resultatet av spørringen. Vi ser nærmere på disse metodene:

1. `select()`-metoden returnerer et `Iterator`-objekt som kan iterere igjennom resultatene av spørringen.

2. `query()`-metoden returnerer et `DCollection`-objekt som inneholder objektene spørringen resulterer i.

3. `selectElement()`-metoden returnerer objektet en spørring resulterer i, uten å pakke dette inn i et samlingsobjekt.

4. `existsElement()`-metoden returnerer den boolske verdien `true` hvis objektet som en spørring resulterer i finnes i samlingsobjektet spørringen utføres fra.

I alle disse metodene kan man referere til samlingsobjektet man spør fra med variabelnavnet `this`. Hvis man f.eks. har et `OSDArray`-objekt `array` vil følgende metodekall returnere dette `OSDarray`-objektet:

```
array.query("this");
```

`OSDArray` og `OSDList` definerer samlingsobjekter med rekkefølge. `OSDArray`-objekter har en fastsatt lengde, i motsetning til `OSDList`-objekter.

Konkatenering av to `OSDList`-objekter gjøres ved å opprette et nytt `OSDList`-objekt, legge til innholdet fra det første `OSDList`-objektet, og deretter legge til innholdet fra det andre `OSDList`-objektet.

`OSDBag` og `OSDSet` definerer samlingsobjekter uten rekkefølge. `OSDSet`-objekter kan ikke ha flere forekomster av samme objekt.

Man kan kalle metoder for å undersøke om et `OSDSet`-objekt er et subsett, ekte subsett, supersett eller ekte supersett av et annet `OSDSet`-objekt. Disse metodene følger vanlige regler for sett under sammenlikningen.

`OSDSet` har også metoder for union, differanse og snitt, som følger vanlige regler for sett. Disse metodene oppretter *nye* `OSDSet`-objekter som inneholder resultatet av operasjonen. Kall på disse metodene endrer dermed ikke eksisterende `OSDSet`-objekter.

Tilsvarende metoder finnes også i `OSDBag`, med den forskjell at disse metodene skal følge vanlige regler for bager.

### 3.6.6 Implementasjon av DMap-grensesnittet

`DMap`-grensesnittet implementeres av `OSDMap`-klassen. Ettersom `DMap`-grensesnittet utvider `java.util.Map` uten å definere noen egne metoder, er det tilstrekkelig at en klasse som skal implementere `DMap`-grensesnittet utvider en klasse fra OSJI som implementerer `java.util.Map`.

## *3.7 Bygging og testing av implementasjonen*

Under arbeidet med oppgaven ble hele implementasjonen bygget og testet hver gang ny funksjonalitet ble implementert.

Selve byggingen foregår i tre trinn som er definert i et Ant-byggescript:

1. Bytekode fra forrige gang implementasjonen ble bygget slettes.
2. Kildekode til scanneren og parseren genereres ved hjelp av CUP og JFlex.
3. Kildekoden til scanneren, parseren, en samling JUnit-tester og resten av implementasjonen kompileres.

Den ferdigbygde implementasjonen lastes opp til en Linux-tjener på UiO-nettverket som kjører en ObjectStore-installasjon. Her blir også bytekode postprosessert slik at den fungerer mot ObjectStore.

Videre kjøres et script som utfører alle JUnit-testene (175 til sammen) for å forsikre om at endringer siden forrige kompilering ikke har ødelagt for fungerende kode. Denne fremgangsmåten kalles *Test Driven Development* ([14]).

All postprosessering og kjøring av kode gjøres av forskjellige script. Alle scriptene ligger i følgende zip-fil:

http://folk.uio.no/~oivindha/OQL.zip

Som utgangspunkt for disse scriptene er det brukt et script fra følgende pdf-dokument:

http://www.uio.no/studier/emner/matnat/ifi/INF5100/h05/undervisningsmateriale/HANDOUTS/L02-Oracle-and-ObjectStore-start-guide.pdf

Testene for de forskjellige metodene er adskilt fra hverandre. Metoder for addisjon, konkatenering av strenger, oppretting av samlingsobjekter osv. testes dermed hver for seg. Hvis f.eks. testen for konkatenering av strenger feiler, vet man hvor man skal begynne å lete etter feil.

Vi ser nærmere på en slik JUnit-test:

```
public void testExprCommonAND(){
  assertEquals(
    new Interpreter( "true and true" ).execute(),
    new Boolean( true ) );
  assertEquals(
    new Interpreter( "true and false" ).execute(),
    new Boolean( false ) );
  assertEquals(
    new Interpreter( "false and true" ).execute(),
    new Boolean( false ) );
  assertEquals(
    new Interpreter( "false and false" ).execute(),
    new Boolean( false ) );
}
```

Her testes det at `and`-operatoren gir riktig svar for alle kombinasjoner av boolske operander. Ettersom disse boolske operandene er literaler, forutsetter disse testene at literaler fungerer som de skal. Det er altså ofte slik at når man skal teste funksjonalitet så må man støtte seg på annen funksjonalitet.

## 3.8 Eksempelprogram

For å illustrere funksjonaliteten i implementasjonen er det laget et eksempelprogram. Det består av klassene `Eksempel`, `Person` og `Bil`. Utgangspunktet for programmet er eksempelprogrammet i kapittel 2 i [10].

For å kjøre programmet må man postprosessere bytekoden til klassene programmet består av. Videre må det settes en rekke miljøvariabler som bl.a. forteller hvor de forskjellige ObjectStore-bibliotekene ligger. Dette gjøres av script som ligger i følgende zip-fil:

http://folk.uio.no/~oivindha/OQL.zip

Programmet kan åpne, lukke, opprette og tømme databaser. Videre kan man opprette og slette `Bil`- og `Person`-objekter. Biler kan eies av personer, og slike eierforhold kan registreres og slettes. En person kan ha en mor, en far og flere barn. Morskap og farskap kan registreres og slettes. Se illustrasjon 25 for hele brukergrensesnittet.

```
Meny:
------
 1) Koble til database
 2) Koble fra database
 3) Opprett ny database
 4) Nullstill database
 5) List opp personer
 6) List opp biler
 7) Registrer ny person
 8) Registrer ny bil
 9) Registrer eierforhold mellom bil og person
10) Slett eierforhold mellom bil og person
11) Registrer foreldre
12) Slett farskap og morskap
13) Slett person
14) Slett bil
15) Utfor OQL-sporring
A) Avslutt
```

*Illustrasjon 25 : Menyen i eksempelprogrammet*

Alle `Bil`- og `Person`-objekter blir lagt i ekstensjoner når de blir opprettet. Dette gjøres ikke *automatisk* av OQL-implementasjonen, men ved at eksempelprogrammet *manuelt* kaller på metoder i OQL-implementasjonen som gjør dette.

Man kan utføre OQL-spørringer mot disse ekstensjonene (se valg 15 i illustrasjon 25). Man kan også utføre OQL-spørringer som hverken benytter seg av ekstensjonene eller de andre objektene i databasen, så lenge spørringen er gyldig. `"1 + 1"` er en slik spørring.

Ettersom ODL ikke er implementert i denne oppgaven, er det eksempelprogrammets oppgave å sørge for at reglene i det tenkte skjemaet overholdes. OQL-spørringer brukes bak kulissene for å forsikre at det ikke lagres to personer med samme navn eller to biler med samme bilnummer.

Først registrerer vi noen personer og biler. Deretter registrerer vi familieforhold og eierforhold mellom personer og biler. De registrerte `Person`- og `Bil`-objektene er listet opp i de følgende to tabellene. Ettersom OQL ikke støtter bokstavene æ, ø og å, skrives f.eks. *rød* som *rod* og *blå* som *bla*. (Det er en triviell utvidelse å la OQL tillate norske tegn i strenger).

Personer:

| Navn | Alder | Gift | Lonn | Gjeld | Skatte prosent | Vekt | Mor | Far | Barn | Biler |
|---|---|---|---|---|---|---|---|---|---|---|
| linda | 25 | false | 455000 | 1624000 | 0.38 | 67.0 | | karl | | dk12345 |
| ola | 65 | true | 460000 | 0 | 0.34 | 87.4 | | | mari<br>karl | |
| mari | 45 | true | 650000 | 2200000 | 0.42 | 78.4 | anne | ola | | dk33332 |
| kari | 43 | true | 345000 | 370000 | 0.32 | 67.4 | | | arne | |
| arne | 22 | false | 290000 | 1530000 | 0.28 | 95.3 | kari | | | de88888 |
| anne | 67 | true | 160000 | 120000 | 0.15 | 57.6 | | | mari<br>karl | |
| olav | 55 | false | 215000 | 87000 | 0.28 | 67.0 | | | | de55555 |
| karl | 43 | true | 360000 | 560000 | 0.32 | 103.5 | anne | ola | linda | se12321 |

Biler:

| Bilnummer | Bilmodell | Farge | Eier |
|---|---|---|---|
| dk12345 | Toyota Yaris | hvit | linda |
| de55555 | Ford Fiesta | gul | olav |
| dk33332 | BMW X3 | gul | mari |
| se12321 | Nissan Micra | rod | karl |
| dk44323 | BMW X3 | bla | |
| de88888 | Citroen XM | solvgraa | arne |

Vi kan nå utføre spørringer mot disse dataene. Vi begynner med å hente ut navn og alder til alle personer som tjener over 300000 kroner i året. Illustrasjon 26 viser hvordan denne spørringen ser ut, samt resultatet av denne.

```
Skriv inn OQL-sporringen
select p.navn as navn, p.alder as alder from person p where p.lonn > 300000
Struct:
-------
navn = linda
alder = 25
Struct:
-------
navn = karl
alder = 43
Struct:
-------
navn = kari
alder = 43
Struct:
-------
navn = mari
alder = 45
Struct:
-------
navn = ola
alder = 65
Sporringen ble utfort
```

*Illustrasjon 26 : Navn og alder til personer som tjener over 300 000*

Vi kan også hente ut navnet til alle som har minst én gul bil. Illustrasjon 27 viser dette.

```
Skriv inn OQL-sporringen
select p.navn from person p where exists x in p.biler : x.farge = "gul"
mari
olav
Sporringen ble utfort
```

*Illustrasjon 27 : Select-uttrykk med exists-uttrykk i where-klausulen*

Det går fint an å utføre sub-select-uttrykk. Anta at vi vil hente ut navn og lønn til barna til personen `ola`. Illustrasjon 28 viser hvordan dette gjøres, samt resultatet av spørringen.

```
Skriv inn OQL-sporringen
select b.navn as navn, b.lonn as lonn from (element(select p.barn from person p
 where p.navn = "ola")) as b
Struct:
-------
navn = karl
lonn = 360000
Struct:
-------
navn = mari
lonn = 650000
Sporringen ble utfort
```

*Illustrasjon 28 : Uttrykk med sub-select*

Disse eksemplene viser hvordan OQL-implementasjonen kan brukes i praksis.

Det er én kjent feil i eksempelprogrammet. For å kunne utføre en spørring, må `Bil`-objektene og `Person`-objektene listes opp først. Denne feilen er det lett å programmere seg rundt, og burde være grei å rette.

# 4 Kritikk av ODMG 3.0-standarden

Under arbeidet med oppgaven fant jeg noen feil og unødvendig tunge formuleringer i ODMG 3.0. Disse blir diskutert i dette kapittelet.

Mange av feilene og manglene er i Java-bindingen. Det skal nevnes at ODMG er klar over at språkbindingene ikke er komplette.

## 4.1 Manglende skille mellom scanner og parser

Literalene i OQL er definert på EBNF form, hvilket er unødvendig rotete. Et eksempel er hvordan strengliteraler er definert (side 131 i [1]):

```
stringLiteral     ::= " { character } "
character         ::= letter
                    | digit
                    | special-character
letter            ::= A | B | ... | Z | a | b | ... | z
digit             ::= 0 | 1 | ... | 9
special-character ::= ? | _ | * | % | \
```

Disse syv linjene kan beskrives med et regulært uttrykk på én linje. De andre literalene blir definert på samme måte. Dette kan tolkes som at man selv kan bestemme hvor man vil trekke skillet mellom scanner og parser.

Grammatikken har også den feilen at det ikke angis hvor det er lov å ha mellomrom (`WhiteSpace`-literaler). Skal man tolke grammatikken strikt, må enten all OQL-kode være sammenhengende (uten mellomrom), ellers må det være lov å ha mellomrom overalt, f.eks. i variabelnavn. Dette må derfor tolkes som en feil i standarden.

## 4.2 Bruk av eksempler som ikke støttes av grammatikken

Slik OQLs EBNF-grammatikk er definert, støtter den ikke alle OQL-eksemplene i standarden. To slike eksempler fra side 92 i [1] følger:

```
vectint(select distinct age
          from Persons
          where name = "Pat")

stats(select stat (a: age, s:sex) from Persons where name =
"Pat")
```

Selve problemet er at man ikke kan bruke select-uttrykk i parentesene. Dette ble utdypet i kapittel 3.3.2.2.

Dette problemet kan løses ved å endre BNF-grammatikken i input-fila til CUP. Jeg har valgt å ikke gjøre dette, ettersom funksjonaliteten i disse eksemplene støtter seg på egendefinerte typer, som ikke er implementert i denne oppgaven.

### 4.3 Utilstrekkelig tegnsett i strengliteraler

Strengliteraler støtter kun tegnene a-z, A-Z, 0-9, ?, _, *, % og \. De støtter altså ikke mellomrom, selv om OQL-eksemplene i boka inneholder mellomrom. Følgende problem kan oppstå: Anta at man oppretter et objekt hvor ett av feltene er en streng `"dette går ikke"`. Dette objektet blir lagret i databasen. Når man skal man hente frem dette objektet igjen prøver man seg med følgende spørring:

```
select o
from objekter o
where o.felt = "dette går ikke"
```

Denne spørringen vil føre til en syntaksfeil, ettersom strengliteraler ikke kan inneholde mellomrom, og heller ikke bokstaven å.

Selv om dette er en stor begrensning, er det lett å endre dette i neste versjon av ODMG-standarden. For å legge til støtte for f.eks. æ, ø, å og mellomrom kan man ganske enkelt inkludere disse tegnene i definisjonen for strengliteraler.

### 4.4 Database-grensesnittet mangler status-metoder

Java-bindingen definerer ingen status metoder i `Database`-grensesnittet. Man kan dermed ikke finne ut om et `Database`-objekt er tilkoblet en database eller hvilket modus en database er åpnet i.

Det er enkelt å legge til to metoder for dette. Den ene kan f.eks. kalles `isOpen()`, og returnere den boolske verdien `true` hvis databasen er åpen, eller `false` i motsatt tilfelle.

Den andre metoden kan f.eks. kalles `openMode()` og returnere `int`-konstantene `NOT_OPEN`, `OPEN_READ_ONLY`, `OPEN_READ_WRITE` og `OPEN_EXCLUSIVE`, som allerede er definert i `Database`-grensesnittet.

### 4.5 Database-grensesnittet mangler metode for å tømme databaser

Man kan ikke tømme en database vha. et `Database`-objekt. En metode som

kan gjøre dette er nyttig for bl.a. test-formål under utvikling av applikasjoner, og burde være en del av `Database`-grensesnittet.

Hvis man skal tømme en database for data, må man manuelt finne alle rot-objekter, slette disse og kjøre garbage-collection på databasen.

I OSJI har `Database`-klassen en `destroy()`-metode som gjør alt dette. Denne blir brukt allerede i eksempelprogrammet i kapittel 2 i brukermanualen [10]. Dette tyder på at en slik metode er nyttig selv for enkle applikasjoner.

## 4.6 Java-bindingen støtter ikke listing av rot-objekter

Java-bindingen definerer ingen metoder for å hente ut en liste over navn på rot-objekter. Man kan heller ikke finne ut om et objekt er et rot-objekt eller hvilket navn som er knyttet til et objekt. Dermed kan man lett miste oversikten over hvilke objekter man har i databasen.

Man kan heller ikke finne ut hvilke klasser som er registrert i skjemaet til databasen. Dermed kan det bli liggende igjen objekter som brukerne av databasen ikke vet at er der.

Det finnes sikkert verktøy til de fleste OODBMS som gjør at man kan se hva en database inneholder. Dermed kan slike problemer stort sett løses, men dette kan da ikke gjøres på en standardisert måte.

I ObjectStore kan man hente ut en liste over rot-objekter ved hjelp av `Database`-klassen i OSJI. Dermed er dette problemet løst for ObjectStore. Likevel burde ODMG 3.0 omfattet bedre støtte for innsikt i databaser, ettersom dette er sentral funksjonalitet i OODBMS.

## 4.7 Java-bindingen bruker ikke generics i samlingsklasser

De forskjellige samlingsklassene i ODMG 3.0 skal ifølge standarden ha funksjonalitet som enkelt kan implementeres med generics, men ODMG 3.0 ble utgitt i 1999 og generics ble ifølge [11] en del av Java i 2004.

Anta at et `Person`-objekt skal referere til et sett av `Bil`-objekter. `Person`-klassen ser slik ut:

```
class Person{
    DSet biler;
}
```

Her kan man fint sette inn f.eks. `Sykkel`-objekter i samlingsobjektet `biler`. I ODMG 3.0 står det at implementasjonen skal sørge for at dette ikke skjer. Dermed må man programmere inn restriksjoner i de forskjellige samlingsklassene.

Hva slags objekter man kan ha i et samlingsobjekt skal beskrives i en `property`-fil som er definert i ODMG 3.0 (side 258 i [1]). Denne `property`-fila skal brukes sammen med ODL når klassene defineres.

Det ville være bedre om den samme klassen kunne defineres med generics på denne måten:

```
class Person{
    DSet<Bil> biler;
}
```

Her kan man ikke sette inn annet enn `Bil`-objekter i samlingsobjektet `biler`. Man er da heller ikke avhengig av `property`-fila for å få til dette (`property`-fila er nødvendig likevel, men for andre formål).

Hvis generics hadde vært en del av Java-språket da Java-bindingen i ODMG 3.0 ble utarbeidet, ville Java-bindingen antakeligvis brukt generics.

## 4.8 Java-bindingen definerer structer for overfladisk

Mange av uttrykkene i OQL resulterer i structer, men Java har ikke slike structer. På side 241 i [1] er det skrevet hvordan structer i OQL skal representeres i en Java-implementasjon av ODMG 3.0:

> *7.1.4.2 Structure*
>
> *The Object Model definition of a structure maps into a Java class.*

Man skal altså programmere en `Struct`-klasse hvis instanser representerer structer. Hva slags funksjonalitet en slik `Struct`-klasse skal inneholde blir ikke beskrevet.

Det er heller ikke definert noe Java-grensesnitt (interface) for en slik `Struct`-klasse. Dette medfører at `Struct`-objekter fra forskjellige implementasjoner ikke kan brukes om hverandre slik som f.eks. `DBag`-objekter.

## 4.9 Array-indekser kan ikke brukes i dot-notasjon

OQL støtter dot-notasjon for aksessering av medlemsvariable. Hvis man har et `Person`-objekt `p` med en medlemsvariabel `alder`, kan man aksessere denne medlemsvariabelen ved å skrive `p.alder` i OQL. Det samme fungerer også for metoder.

Man kan derimot ikke benytte dot-notasjon hvis array-indekser inngår som en del av uttrykket. Anta et `Person`-objekt `p` med et array `barn` av `Person`-objekter. For å finne alderen til det første av disse barna, ville man skrive `p.barn[0].alder`. Slike uttrykk støttes ikke av grammatikken til OQL, og må anses som en forglemmelse i standarden.

## 4.10 Nøkkelord kan ikke brukes i dot-notasjon

EBNF-grammatikken i OQL tar ikke høyde for at nøkkelord skal kunne brukes i uttrykk med dot-notasjon. Anta f.eks. at `Math.abs()`-metoden skal brukes mot argumentet `-14` i et uttrykk. Følgende uttrykk burde kunne gjøre dette:

```
Math.abs(-14)
```

Denne spørringen fører til en syntaksfeil. Dette er på grunn av en svakhet i EBNF-grammatikken. Vi ser nærmere på regelen for dot-notasjon:

```
primaryExpr { (.|->) identifier [arglist]}
```

Utdraget er hentet fra side 129 i [1]. Identifikatoren i dette uttrykket er `abs`, men `abs` er et nøkkelord i OQL, og blir derfor tolket som et nøkkelord istedenfor en identifikator.

Løsningen på dette problemet er å definere deler av dot-notasjonen som et regulært uttrykk. Man kan f.eks. lage følgende regel:

```
DotNotation = ("."|"->") [a-zA-Z0-9_]*
```

Med denne reglen vil f.eks. teksten `.abs` tolkes som et sammenhengende leksem istedenfor et punktum og et `abs`-leksem. På denne måten blir nøkkelord gjenkjent som identifikatorer når de blir brukt i uttrykk med dot-notasjon. I øvrige tilfeller blir de gjenkjent som nøkkelord.

# 5 Oppsummering og videre arbeid

Implementasjonen har kommet så langt at den gir riktige resultater på alle OQL-spørringer som bruker de funksjoner som er implementert og testet, hvilket er omtrent alle i hele standarden.

Det er med all sannsynlighet fremdeles noen feil i implementasjonen, og det er fortsatt noen få funksjoner i OQL som ikke er implementert i sin helhet. Mesteparten av denne funksjonaliteten burde gå greit å implementere, og blir diskutert i dette kapitlet.

## *5.1 Kjente feil og mangler i implementasjonen*

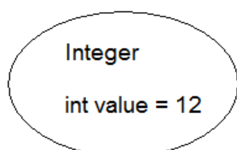
Transaksjonsmodellen i ODMG 3.0 er annerledes enn transaksjonsmodellen i ObjectStore. Å tilpasse transaksjonsmodellen til ObjectStore slik at den følger transaksjonsmodellen i ODMG 3.0, er mulig, men ettersom dette ikke er viktig for å få OQL til å fungere, har jeg ikke fokusert på dette.

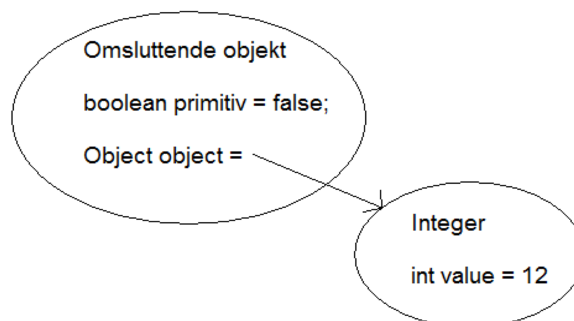
Litt spesielt for denne implementasjonen er at typesjekking av uttrykk i OQL-spørringer ikke utføres før spørringer eksekveres. Dette har ingen konsekvenser for resultatene av OQL-spørringer, men feil oppdages senere enn nødvendig. Det burde ikke være vanskelig å rette dette.

Ettersom svar på OQL-spørringer alltid skal være et objekt, og ikke et primitiv, har ODMG 3.0 definert hvilke objekter de forskjellige primitivene skal pakkes inn i. For eksempel skal primitiver av typen `int` pakkes inn i `Integer`-objekter.

Jeg har valgt å behandle objektene som pakker inn disse primitivene som primitiver i alle tilfeller. Dette betyr at i de tilfeller hvor disse objektene ikke skal behandles som primitiver, vil de bli behandlet som primitiver likevel.

Det går an å rette på denne forenklingen ved å lage en ny klasse som beskriver objekter som omslutter slike innpakningsobjekter og beskriver om de skal behandles som primitiver eller objekter. Denne ideen er vist i illustrasjon 29.




*Illustrasjon 29 : Forslag til løsning på problemet med å skille primitiver fra objekter*

I ODMG 3.0 står det at hvis klasse `A` har en referanse typet med klassen `B`, så må `B` ha en motsvarende referanse til klasse `A`. Dette kalles et *relationship.* Implementasjonen i denne oppgaven sørger ikke for at denne regelen blir overholdt ettersom det er ODLs ansvar å definere klasser og skjemaer.

Java-programmer kan aksessere C++ objekter i en ObjectStore-database ved å bruke såkalte *peer*-objekter. Dette støttes ikke i denne implementasjonen, og er heller ikke sentralt. Å implementere støtte for slike objekter vil gjøre det mulig å bruke en implementasjon mellom C++ og ObjectStore sammen med denne implementasjonen.

Det blir ikke sjekket at variabelnavn er unike. Det er derfor mulig å overskrive deklarasjoner av variable med nye deklarasjoner. Å rette på denne forenklingen kan stort sett gjøres rett frem ved å sjekke at nye variable ikke har samme navn som variable som allerede er definert.

Samlingsobjekter skal ha restriksjoner for hvilken type objekter de kan inneholde. Slike restriksjoner er ikke implementert, ettersom dette henger sammen med skjemaer og ODL.

Ikke alle uttrykkene i OQL er implementert i sin helhet. Det er forholdsvis enkelt å implementere de fleste av de som gjenstår, unntatt gruppering i select-uttrykk. Følgende tabell viser de uttrykkene som ikke er implementert:

| `select from where group by` | Gruppering i select-uttrykk er ikke implementert. |
|---|---|
| `select from where order by` | Sortering i select-uttrykk er ikke implementert. |
| `typedef ...` | Definisjon av egendefinerte typer er ikke implementert. |

| | |
|---|---|
| `list(1,2,3,4)[1 .. 3]`<br>`array(1,2,3,4)[1 .. 3]`<br>`list(1,2,3,4,5)[1,3,4]`<br>`array(1,2,3,4,5)[1,3,4]` | Å hente ut sub-lister eller sub-vektorer er ikke implementert. |
| `define query ...` | Definering og lagring av navngitte spørringer og variable er ikke implementert. Det burde gå greit å implementere dette, men det burde testes at alle typer uttrykk kan lagres, hvilket fører til en del arbeid. |
| `undefine query ...` | Sletting av navngitte spørringer og variable er ikke implementert. Dette burde være trivielt å implementere når definering og lagring av navngitte spørringer er gjort. |
| `objekt.metode(null)` | Metoder i OQL-spørringer kan ikke kalles med null-verdier. Implementasjonen må vite nøyaktig hvilke typer argumentene til en metode har. |
| `"tekst" like "t??s*"` | Matching av strenger mot regulære uttrykk er ikke implementert. |
| `enum` | Enumerations støttes ikke. |
| (set<type>) ... | Man kan ikke typekaste til en kompleks type. |
| `1 in list(1,2,3,4)` | Man kan ikke bruke frittstående `in`-uttrykk. Man kan derimot skrive om uttrykket i eksempelet til følgende uttrykk:<br>`exists x in list(1,2,3,4): x = 1` |
| `select p.navn, p.alder ...` | Select-uttrykk må navngi de projiserte attributtene eksplisitt. Spørringen i eksempelet støttes ikke, men kan skrives på følgende måte:<br>`select p.navn` **`as navn,`**<br>`p.alder` **`as alder ...`** |

## 5.2 Videre forskning og utvikling

Denne implementasjonen kan forbedres ved å implementere ODL. Dette kan gjøres ved å lage et program som leser ODL-definisjoner av klasser og lager Java-kildekode for disse klassene. Dette programmet må også lagre skjema-informasjon i databasen. Videre må OQL-implementasjonen tilpasses til å bruke denne skjema-informasjonen.

Effektivisering av kode og optimering av tiden brukt av spørringer har ikke vært i fokus, men det er heller ikke sikkert dette er så viktig i praksis.

Eksekvering av spørringer kan optimeres ved å bruke flere tråder til å regne ut resultatet av et select-uttrykk. Ved å dele opp svaret fra select-from-delen av uttrykket kan flere tråder jobbe parallelt med where-delen av uttrykket.

De ikke-implementerte delene av OQL angår hovedsaklig beregning av resultater av uttrykk. Parseren er komplett og kan bygge abstrakte syntakstrær for alle gyldige uttrykk i OQL. Det kan likevel hende at det lønner seg å gjøre små endringer i parseren.

Alle objekter som skal kunne nåes av OQL-spørringer må lagres eksplisitt i sine respektive ekstensjoner. Dette fører til problemer når ObjectStore bruker *Transitive persistence* til å lagre en samling sammenkoblede objekter i en database.

Hvis f.eks. et objekt `a` legges i en ekstensjon, så blir `a` lagret i databasen. Hvis `a` peker på objekt `b`, vil `b` automatisk lagres i databasen. Likevel blir ikke `b` lagt i en ekstensjon ettersom dette ikke gjøres eksplisitt. Det er mulig man kan løse dette problemet ved å skrive om `commit()`-metoden i `OSTransaction`-klassen til å legge endrede objekter i ekstensjoner.

Et liknende problem opptrer ved sletting av objekter. Hvis et objekt skal fjernes fra en database, så må det ikke bare kobles fra alle objekter som er lagret i databasen, men det må også slettes fra ekstensjoner. Det må altså ikke kunne nåes fra et rot-objekt.

# 6 Appendikser

## 6.1 Tabell med koblingen mellom ODMGs objektmodell og Java

Her er en oversikt over hva de forskjellige typene i ODMGs objektmodell skal representeres som i Java. Tabellen er hentet fra side 246 i [1].

| Object Model Type | Java Type | Literal? |
|---|---|---|
| `Long` | `int` (primitive), `Integer` (class) | yes |
| `Short` | `short` (primitive), `Short` (class) | yes |
| `Unsigned long` | `long` (primitive), `Long` (class) | yes |
| `Unsigned short` | `int` (primitive), `Integer` (class) | yes |
| `Float` | `float` (primitive), `Float` (class) | yes |
| `Double` | `double` (primitive), `Double` (class) | yes |
| `Boolean` | `boolean` (primitive), `Boolean` (class) | yes |
| `Octet` | `byte` (primitive), `Integer` (class) | yes |
| `Char` | `char` (primitive), `Character` (class) | yes |
| `String` | `String` | yes |
| `Date` | `java.sql.Date` | yes |
| `Time` | `java.sql.Time` | yes |
| `Timestamp` | `java.sql.Timestamp` | yes |
| `Set` | `interface DSet` | no |
| `Bag` | `interface DBag` | no |
| `List` | `interface DList` | no |
| `Array` | array type `[]` or `Vector` or interface `DArray` | no |
| `Dictionary` | interface `DMap` | no |
| `Iterator` | `java.util.Iterator` | no |

## 6.2 Beskrivelse av OQL-leksemer i JFlex

Her er innholdet i fila `oql.lex`, som blir brukt av JFlex til å generere en scanner. Fila beskriver også hvordan alle leksemer i OQL ser ut. Teksten har blitt formatert for å passe bedre inn i dette dokumentet.

```
package org.odmg.impl;
import java_cup.runtime.*;
import com.odi.odmg.*;

%%

%class Lexer
%public
%cup
%unicode
%line
%column

WhiteSpace          = [ |\t|\f|\r|\n|\r\n]
QueryParam          = "$" [1-9] [0-9]*
Identifier          = [a-zA-Z][a-zA-Z0-9_]*
Exponent            = [E|e][+|-]?[0-9]+
DoubleLiteral       = [0-9]+ "." [0-9]+ {Exponent}?
LongLiteral         = [0-9]+
CharLiteral         = "'" [a-zA-Z0-9?_*%\\] "'"
StringLiteral       = "\"" [a-zA-Z0-9?_*%\\ ]* "\""
DateLiteral         = "'" [0-9]+ "-" [0-9]+ "-" [0-9]+ "'"
TimeLiteral         = "'" [0-9]+ [:] [0-9]+ [:] ([0-9]+ |[0-9]+
"." [0-9]+)"'"
TimeStampLiteral    = "'" [0-9]+ "-" [0-9]+ "-" [0-9]+ [ ]
[0-9]+ [:] [0-9]+ [:] ([0-9]+ |[0-9]+ "." [0-9]+)"'"

%%

"abs"               { return new Symbol(sym._abs);}
"all"               { return new Symbol(sym._all);}
"and"               { return new Symbol(sym._and);}
"andthen"           { return new Symbol(sym._andthen);}
"any"               { return new Symbol(sym._any);}
"array"             { return new Symbol(sym._array);}
"as"                { return new Symbol(sym._as);}
"asc"               { return new Symbol(sym._asc);}
"avg"               { return new Symbol(sym._avg);}
"bag"               { return new Symbol(sym._bag);}
"boolean"           { return new Symbol(sym._boolean);}
"by"                { return new Symbol(sym._by);}
"char"              { return new Symbol(sym._char);}
"count"             { return new Symbol(sym._count);}
"date"              { return new Symbol(sym._date);}
"define"            { return new Symbol(sym._define);}
"desc"              { return new Symbol(sym._desc);}
```

```
"dictionary"        { return new Symbol(sym._dictionary);}
"distinct"          { return new Symbol(sym._distinct);}
"double"            { return new Symbol(sym._double);}
"element"           { return new Symbol(sym._element);}
"enum"              { return new Symbol(sym._enum);}
"except"            { return new Symbol(sym._except);}
"exists"            { return new Symbol(sym._exists);}
"false"             { return new Symbol(sym._false);}
"first"             { return new Symbol(sym._first);}
"flatten"           { return new Symbol(sym._flatten);}
"float"             { return new Symbol(sym._float);}
"for"               { return new Symbol(sym._for);}
"from"              { return new Symbol(sym._from);}
"group"             { return new Symbol(sym._group);}
"having"            { return new Symbol(sym._having);}
"import"            { return new Symbol(sym._import);}
"in"                { return new Symbol(sym._in);}
"intersect"         { return new Symbol(sym._intersect);}
"interval"          { return new Symbol(sym._interval);}
"is_defined"        { return new Symbol(sym._is_defined);}
"is_undefined"      { return new Symbol(sym._is_undefined);}
"last"              { return new Symbol(sym._last);}
"like"              { return new Symbol(sym._like);}
"list"              { return new Symbol(sym._list);}
"listtoset"         { return new Symbol(sym._listtoset);}
"long"              { return new Symbol(sym._long);}
"max"               { return new Symbol(sym._max);}
"min"               { return new Symbol(sym._min);}
"mod"               { return new Symbol(sym._mod);}
"nil"               { return new Symbol(sym._nil);}
"not"               { return new Symbol(sym._not);}
"octet"             { return new Symbol(sym._octet);}
"or"                { return new Symbol(sym._or);}
"order"             { return new Symbol(sym._order);}
"orelse"            { return new Symbol(sym._orelse);}
"query"             { return new Symbol(sym._query);}
"select"            { return new Symbol(sym._select);}
"set"               { return new Symbol(sym._set);}
"short"             { return new Symbol(sym._short);}
"some"              { return new Symbol(sym._some);}
"string"            { return new Symbol(sym._string);}
"struct"            { return new Symbol(sym._struct);}
"sum"               { return new Symbol(sym._sum);}
"time"              { return new Symbol(sym._time);}
"timestamp"         { return new Symbol(sym._timestamp);}
"true"              { return new Symbol(sym._true);}
"undefine"          { return new Symbol(sym._undefine);}
"undefined"         { return new Symbol(sym._undefined);}
"union"             { return new Symbol(sym._union);}
"unique"            { return new Symbol(sym._unique);}
"unsigned"          { return new Symbol(sym._unsigned);}
"where"             { return new Symbol(sym._where);}
```

```
"||"                 { return new Symbol(sym._concatenate);}
".."                 { return new Symbol(sym._dotdot);}
"->"                 { return new Symbol(sym._rightarrow);}
">="                 { return new Symbol(sym._ge);}
"<="                 { return new Symbol(sym._le);}
"!="                 { return new Symbol(sym._neq);}
"("                  { return new Symbol(sym._parenleft);}
")"                  { return new Symbol(sym._parenright);}
"{"                  { return new Symbol(sym._brackleft);}
"}"                  { return new Symbol(sym._brackright);}
"="                  { return new Symbol(sym._eq);}
"<"                  { return new Symbol(sym._lt);}
">"                  { return new Symbol(sym._gt);}
"+"                  { return new Symbol(sym._plus);}
"-"                  { return new Symbol(sym._minus);}
"*"                  { return new Symbol(sym._mult);}
"/"                  { return new Symbol(sym._div);}
":"                  { return new Symbol(sym._colon);}
";"                  { return new Symbol(sym._semi);}
"."                  { return new Symbol(sym._dot);}
","                  { return new Symbol(sym._comma);}
{QueryParam}
   { return new Symbol(sym._queryparam, yytext());}
{StringLiteral}
   { String s = yytext();
     return new Symbol(sym._stringliteral, s.substring(1,
     s.length()-1));}
{Identifier}
   { return new Symbol(sym._identifier, yytext());}
{CharLiteral}
   { return new Symbol(sym._charliteral, yytext().charAt(1) +
     "");}
{LongLiteral}
   { return new Symbol(sym._longliteral, yytext());}
{DateLiteral}
   { return new Symbol(sym._dateliteral, yytext());}
{DoubleLiteral}
   { return new Symbol(sym._doubleliteral, yytext());}
{TimeLiteral}
   { return new Symbol(sym._timeliteral, yytext());}
{TimeStampLiteral}
   { return new Symbol(sym._timestampliteral, yytext());}
{WhiteSpace}
   { }
.
   { throw new QueryInvalidException("Illegal character '" +
     yytext() + "' at line " + yyline + ", column " + yycolumn
     + "."); }
```

## 6.3 Beskrivelse av OQL-syntaks i CUP

Her er innholdet i fila `oql.cup`, som blir brukt av CUP til å generere en

parser. Fila beskriver også grammatikken i OQL. Teksten har blitt formatert for å passe bedre inn i dette dokumentet.

```
package org.odmg.impl;
import org.odmg.impl.syntaxtree.*;
import java_cup.runtime.*;

/* Terminals */
terminal _abs,_all,_and,_andthen,_any,_array,_as,_asc,_avg;
terminal _bag, _boolean,_brackleft,_brackright,_by,_char;
terminal _colon,_comma,_concatenate,_count,_date,_define;
terminal _desc,_dictionary,_distinct,_div,_dot,_dotdot;
terminal _double,_element,_enum,_eq,_except,_exists,_false;
terminal _first,_flatten,_float,_for,_from,_ge,_group,_gt;
terminal _having,_import,_in,_intersect,_interval,_is_defined;
terminal

_is_undefined,_last,_le,_like,_list,_listtoset,_long;
terminal _lt,_max,_min,_minus,_mod,_mult,_neq,_nil,_not;
terminal _octet,_or,_order,_orelse,_parenleft,_parenright;
terminal _plus,_query,_rightarrow,_select,_semi,_set,_short;
terminal _some,_string,_struct,_sum,_time,_timestamp,_true;
terminal _undefine,_undefined,_union,_unique,_unsigned,_where;

terminal  String   _charliteral;
terminal  String   _dateliteral;
terminal  String   _doubleliteral;
terminal  String   _identifier;
terminal  String   _longliteral;
terminal  String   _queryparam;
terminal  String   _stringliteral;
terminal  String   _timeliteral;
terminal  String   _timestampliteral;

/* Non-terminals */
non terminal Expr                    ADDITIVEEXPR;
non terminal Expr                    ANDEXPR;
non terminal Expr                    ANDTHENEXPR;
non terminal Expr                    ARGLIST;
non terminal Expr                    CASTEXPR;
non terminal Integer                 COMPOSITEPREDICATE;
non terminal Declaration             DECLARATION;
non terminal Declaration             DECLARATIONS;
non terminal Declaration             DEFINEQUERY;
non terminal Expr                    DIVS;
non terminal Expr                    DOTEXPR;
non terminal Expr                    EQUALITYEXPR;
non terminal Expr                    EQUALITYEXPRS;
non terminal Expr                    EXCEPTS;
non terminal Expr                    EXPR;
non terminal Expr                    EXPRLIST;
non terminal Expr                    FIELD;
```

```
non terminal Expr                       FIELDLIST;
non terminal Expr                       FROMCLAUSE;
non terminal Expr                       GROUPCLAUSE;
non terminal Expr                       HAVINGCLAUSE;
non terminal String                     IDENTIFIERS;
non terminal DeclarationImport          IMPORT;
non terminal Expr                       INDEX;
non terminal Expr                       INDEXES;
non terminal Expr                       INEXPR;
non terminal Expr                       INTERSECTS;
non terminal Expr                       ITERATORDEF;
non terminal Expr                       ITERATORDEFS;
non terminal Expr                       LIKEEXPRESSIONS;
non terminal Expr                       LITERAL;
non terminal Expr                       MINUSES;
non terminal Expr                       MODS;
non terminal Expr                       MULS;
non terminal Expr                       MULTIPLICATIVEEXPR;
non terminal Expr                       ORDERCLAUSE;
non terminal Expr                       ORELSEEXPR;
non terminal Expr                       OREXPR;
non terminal Expr                       ORS;
non terminal ParameterList              PARAMETERLIST;
non terminal Expr                       PLUSSES;
non terminal Expr                       POSTFIXEXPR;
non terminal Expr                       PRIMARYEXPR;
non terminal Expr                       PROJECTION;
non terminal Expr                       PROJECTIONATTRIBUTES;
non terminal Expr                       PROJECTIONLIST;
non terminal String                     QUALIFIEDNAME;
non terminal Expr                       QUANTIFIEREXPR;
non terminal Expr                       QUERY;
non terminal QueryProgram               QUERYPROGRAM;
non terminal Expr                       RELATIONALEXPR;
non terminal Expr                       SELECTEXPR;
non terminal Expr                       SORTCRITERIA;
non terminal Expr                       SORTCRITERION;
non terminal Typedef                    TYPE;
non terminal Expr                       UNARYEXPR;
non terminal DeclarationUndefineQuery UNDEFINEQUERY;
non terminal Expr                       UNIONS;
non terminal Expr                       VALUELIST;
non terminal Expr                       WHERECLAUSE;

/* BNF grammatikk */
QUERYPROGRAM
  ::= DECLARATIONS:d
    {: RESULT= new QueryProgram(d, null); :}
  | DECLARATIONS:d _semi QUERY:q
    {: RESULT= new QueryProgram(d, q); :}
  | QUERY:q {: RESULT= new QueryProgram(null, q); :};
DECLARATIONS
```

```
  ::= DECLARATION:d
    {: RESULT= d; :}
  | DECLARATIONS:ds _semi DECLARATION:d
    {: ds.insertLast(d); RESULT = ds; :};
DECLARATION
  ::= IMPORT:i {: RESULT=i; :}
  | DEFINEQUERY:d {: RESULT=d; :}
  | UNDEFINEQUERY:u {: RESULT=u; :};
IMPORT
  ::= _import QUALIFIEDNAME:string
    {: RESULT = new DeclarationImport(string, null); :}
  |  _import QUALIFIEDNAME:string _as _identifier:id
    {: RESULT = new DeclarationImport(string, id); :};
DEFINEQUERY
  ::= _define _query _identifier:i _parenleft PARAMETERLIST:p
  _parenright _as QUERY:q
    {:RESULT=new DeclarationDefineQuery(i, q, p);:}
  | _define _query _identifier:i _parenleft _parenright _as
  QUERY:q
    {:RESULT=new DeclarationDefineQuery(i, q, null);:}
  | _define _query _identifier:i _as QUERY:q
    {:RESULT=new DeclarationDefineQuery(i, q, null);:}
  | _define _identifier:i _parenleft PARAMETERLIST:p
  _parenright _as QUERY:q
    {:RESULT=new DeclarationDefineQuery(i, q, p);:}
  | _define _identifier:i _parenleft _parenright _as
  QUERY:q
    {:RESULT=new DeclarationDefineQuery(i, q, null);:}
  | _define _identifier:i _as QUERY:q
    {:RESULT=new DeclarationDefineQuery(i, q, null);:};
PARAMETERLIST
  ::= TYPE:t _identifier:i
    {:RESULT = new ParameterList(i, t, null);:}
  | _identifier:t _identifier:i
    {:RESULT = new ParameterList(i, new Typedef(null, t, null,
    null), null);:}
  | TYPE:t _identifier:i _comma PARAMETERLIST:p
    {:RESULT = new ParameterList(i, t, p);:}
  | _identifier:t _identifier:i _comma PARAMETERLIST:p
    {:RESULT = new ParameterList(i, new Typedef(null, t, null,
    null), p);:};
UNDEFINEQUERY
  ::= _undefine _identifier:i
    {: RESULT=new DeclarationUndefineQuery(i);:}
  | _undefine _query _identifier:i
    {: RESULT=new DeclarationUndefineQuery(i);:};
QUALIFIEDNAME
  ::= _identifier:i {: RESULT=i; :}
  | _identifier:i _dot QUALIFIEDNAME:qn
    {: RESULT=i + "." + qn;:};
QUERY
  ::= SELECTEXPR:se {:RESULT=se;:}
```

```
  | EXPR:e {:RESULT=e;:};
SELECTEXPR
  ::= _select _distinct PROJECTIONATTRIBUTES:p FROMCLAUSE:f
  WHERECLAUSE:w GROUPCLAUSE:g ORDERCLAUSE:o
    {:RESULT=new ExprSelect(true, p, f, w, g, o);:}
  | _select _distinct PROJECTIONATTRIBUTES:p FROMCLAUSE:f
  WHERECLAUSE:w GROUPCLAUSE:g
    {:RESULT=new ExprSelect(true, p, f, w, g, null);:}
  | _select _distinct PROJECTIONATTRIBUTES:p FROMCLAUSE:f
  WHERECLAUSE:w ORDERCLAUSE:o
    {:RESULT=new ExprSelect(true, p, f, w, null, o);:}
  | _select _distinct PROJECTIONATTRIBUTES:p FROMCLAUSE:f
  WHERECLAUSE:w
    {:RESULT=new ExprSelect(true, p, f, w, null, null);:}
  | _select _distinct PROJECTIONATTRIBUTES:p FROMCLAUSE:f
  GROUPCLAUSE:g ORDERCLAUSE:o
    {:RESULT=new ExprSelect(true, p, f, null, g, o);:}
  | _select _distinct PROJECTIONATTRIBUTES:p FROMCLAUSE:f
  GROUPCLAUSE:g
    {:RESULT=new ExprSelect(true, p, f, null, g, null);:}
  | _select _distinct PROJECTIONATTRIBUTES:p FROMCLAUSE:f
  ORDERCLAUSE:o
    {:RESULT=new ExprSelect(true, p, f, null, null, o);:}
  | _select _distinct PROJECTIONATTRIBUTES:p FROMCLAUSE:f
    {:RESULT=new ExprSelect(true, p, f, null, null, null);:}
  | _select PROJECTIONATTRIBUTES:p FROMCLAUSE:f WHERECLAUSE:w
  GROUPCLAUSE:g ORDERCLAUSE:o
    {:RESULT=new ExprSelect(false, p, f, w, g, o);:}
  | _select PROJECTIONATTRIBUTES:p FROMCLAUSE:f WHERECLAUSE:w
  GROUPCLAUSE:g
    {:RESULT=new ExprSelect(false, p, f, w, g, null);:}
  | _select PROJECTIONATTRIBUTES:p FROMCLAUSE:f WHERECLAUSE:w
  ORDERCLAUSE:o
    {:RESULT=new ExprSelect(false, p, f, w, null, o);:}
  | _select PROJECTIONATTRIBUTES:p FROMCLAUSE:f WHERECLAUSE:w
    {:RESULT=new ExprSelect(false, p, f, w, null, null);:}
  | _select PROJECTIONATTRIBUTES:p FROMCLAUSE:f GROUPCLAUSE:g
  ORDERCLAUSE:o
    {:RESULT=new ExprSelect(false, p, f, null, g, o);:}
  | _select PROJECTIONATTRIBUTES:p FROMCLAUSE:f GROUPCLAUSE:g
    {:RESULT=new ExprSelect(false, p, f, null, g, null);:}
  | _select PROJECTIONATTRIBUTES:p FROMCLAUSE:f
  ORDERCLAUSE:o
    {:RESULT=new ExprSelect(false, p, f, null, null, o);:}
  | _select PROJECTIONATTRIBUTES:p FROMCLAUSE:f
    {:RESULT=new ExprSelect(false, p, f, null, null, null);:};
PROJECTIONATTRIBUTES
  ::= PROJECTIONLIST:p {:RESULT=p;:}
  | _mult {:RESULT=null;:};
PROJECTIONLIST
  ::= PROJECTION:p {:RESULT=p;:}
  | PROJECTION:p _comma PROJECTIONLIST:pl
```

```
    {:((ExprCommon)p).right = pl; RESULT=p;:};
PROJECTION
  ::= FIELD:f
    {:RESULT=new ExprCommon(f, ExprCommon.PROJECTIONLIST);:}
  | EXPR:e
    {:RESULT=new ExprCommon(e, ExprCommon.PROJECTIONLIST);:}
  | EXPR:e _as _identifier:i
    {:RESULT=new ExprCommon(e, null, i,
      ExprCommon.PROJECTIONLIST);:};
FROMCLAUSE
  ::= _from ITERATORDEFS:i {:RESULT=i;:};
ITERATORDEFS
  ::= ITERATORDEF:i {:RESULT=i;:}
  | ITERATORDEF:i _comma ITERATORDEFS:is
    {:((ExprCommon) i).right=is;RESULT=i;:};
ITERATORDEF
  ::= EXPR:e _as _identifier:i
    {:RESULT = new ExprCommon(e,null, i,
    ExprCommon.ITERATORDEF);:}
  | EXPR:e _identifier:i
    {:RESULT = new ExprCommon(e,null, i,
    ExprCommon.ITERATORDEF);:}
  | EXPR:e
    {:RESULT = new ExprCommon(e,null, null,
    ExprCommon.ITERATORDEF);:}
  | _identifier:i _in EXPR:e
    {:RESULT = new ExprCommon(e,null,  i,
    ExprCommon.ITERATORDEF);:};
WHERECLAUSE
  ::= _where EXPR:e {:RESULT=e;:};
GROUPCLAUSE
  ::= _group _by FIELDLIST:f
    {:RESULT=new ExprCommon(f, ExprCommon.GROUPCLAUSE);:}
  | _group _by FIELDLIST:f HAVINGCLAUSE:h
    {:RESULT=new ExprCommon(f, h, ExprCommon.GROUPCLAUSE);:}
  ;
HAVINGCLAUSE
  ::= _having EXPR:e {:RESULT=e;:};
ORDERCLAUSE
  ::= _order _by SORTCRITERIA:s {:RESULT=s;:};
SORTCRITERIA
  ::= SORTCRITERION:sn {:RESULT=sn;:}
  | SORTCRITERION:sn _comma SORTCRITERIA:sa
    {:((ExprCommon)sn).right= sa;RESULT=sn;:};
SORTCRITERION
  ::= EXPR:e
    {:RESULT=new ExprCommon(e, null, "desc",
    ExprCommon.SORTCRITERION);:}
  | EXPR:e _asc
    {:RESULT=new ExprCommon(e, null, "asc",
    ExprCommon.SORTCRITERION);:}
  | EXPR:e _desc
```

```
    {:RESULT=new ExprCommon(e, null, "desc",
    ExprCommon.SORTCRITERION);:};
EXPR
  ::= CASTEXPR:e {:RESULT=e;:};
CASTEXPR
  ::= OREXPR:o {:RESULT=o;:}
  | _parenleft TYPE:t _parenright CASTEXPR:c
    {:RESULT=new ExprCast(t, c);:}
  | _parenleft _identifier:i _parenright CASTEXPR:c
    {:RESULT=new ExprCast(new Typedef(null, i, null, null),
    c);:};
OREXPR
  ::= ORELSEEXPR:oe {:RESULT=oe;:}
  | ORELSEEXPR:oe _or OREXPR:o
    {:RESULT=new ExprCommon(oe, o, ExprCommon.OR);:};
ORELSEEXPR
  ::= ANDEXPR:ae {:RESULT=ae;:}
  | ANDEXPR:ae _orelse ORELSEEXPR:oe
     {:RESULT=new ExprCommon(ae, oe, ExprCommon.ORELSE);:};
ANDEXPR
  ::= QUANTIFIEREXPR:qe {:RESULT=qe;:}
  | QUANTIFIEREXPR:qe _and ANDEXPR:ae
    {:RESULT=new ExprCommon(qe, ae, ExprCommon.AND);:};
QUANTIFIEREXPR
  ::= ANDTHENEXPR:ate {:RESULT=ate;:}
  | _for _all _identifier:i _in EXPR:e _colon ANDTHENEXPR:ate
    {:RESULT=new ExprCommon(ate, e, i,
    ExprCommon.QUANTIFIERFOR_ALL);;:}
  | _exists _identifier:i _in EXPR:e _colon ANDTHENEXPR:ate
    {:RESULT=new ExprCommon(ate, e, i,
    ExprCommon.QUANTIFIEREXISTS);;:};
ANDTHENEXPR
  ::= EQUALITYEXPR:ee {:RESULT=ee;:}
  | EQUALITYEXPR:ee _andthen ANDTHENEXPR:ate
    {:RESULT=new ExprCommon(ee, ate, ExprCommon.ANDTHEN);:};
EQUALITYEXPR
  ::= EQUALITYEXPRS:ee {:RESULT=ee;:}
  | LIKEEXPRESSIONS:le {:RESULT=le;:}
  | RELATIONALEXPR:re {:RESULT=re;:};
EQUALITYEXPRS
  ::= RELATIONALEXPR:ee _eq  RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ee, re, ExprCommon.EQ);:}
  | RELATIONALEXPR:ee _neq RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ee, re, ExprCommon.NEQ);:}
  | RELATIONALEXPR:ee _eq  COMPOSITEPREDICATE:cp
  RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ee, re,ExprCommon.EQ
    +cp.intValue());:}
  | RELATIONALEXPR:ee _neq COMPOSITEPREDICATE:cp
  RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ee, re,ExprCommon.NEQ
    +cp.intValue());:}
```

```
  | EQUALITYEXPRS:ee _eq  RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ee, re, ExprCommon.EQ);:}
  | EQUALITYEXPRS:ee _neq RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ee, re, ExprCommon.NEQ);:}
  | EQUALITYEXPRS:ee _eq  COMPOSITEPREDICATE:cp
  RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ee, re,ExprCommon.EQ
    +cp.intValue());:}
  | EQUALITYEXPRS:ee _neq COMPOSITEPREDICATE:cp
  RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ee, re,ExprCommon.NEQ
    +cp.intValue());:};
LIKEEXPRESSIONS
  ::= RELATIONALEXPR:re _like RELATIONALEXPR:rre
    {:RESULT=new ExprCommon(re, rre, ExprCommon.LIKE);:}
  | LIKEEXPRESSIONS:le _like RELATIONALEXPR:re
    {:RESULT=new ExprCommon(le, re, ExprCommon.LIKE);:};
RELATIONALEXPR
  ::= ADDITIVEEXPR:ae {:RESULT=ae;:}
  | ADDITIVEEXPR:ae _lt RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ae, re, ExprCommon.LT);:}
  | ADDITIVEEXPR:ae _le RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ae, re, ExprCommon.LTE);:}
  | ADDITIVEEXPR:ae _gt RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ae, re, ExprCommon.GT);:}
  | ADDITIVEEXPR:ae _ge RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ae, re, ExprCommon.GTE);:}
  | ADDITIVEEXPR:ae _lt COMPOSITEPREDICATE:cp
  RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ae, re, ExprCommon.LT
    +cp.intValue());:}
  | ADDITIVEEXPR:ae _le COMPOSITEPREDICATE:cp
  RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ae, re, ExprCommon.LTE
    +cp.intValue());:}
  | ADDITIVEEXPR:ae _gt COMPOSITEPREDICATE:cp
  RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ae, re, ExprCommon.GT
    +cp.intValue());:}
  | ADDITIVEEXPR:ae _ge COMPOSITEPREDICATE:cp
  RELATIONALEXPR:re
    {:RESULT=new ExprCommon(ae, re, ExprCommon.GTE
    +cp.intValue());:};
COMPOSITEPREDICATE
  ::= _some {:RESULT=new Integer(ExprCommon.SOME_OFFSET);:}
  | _all {:RESULT=new Integer(ExprCommon.ALL_OFFSET);:}
  | _any {:RESULT=new Integer(ExprCommon.ANY_OFFSET);:};
ADDITIVEEXPR
  ::= MULTIPLICATIVEEXPR:m {:RESULT=m;:}
  | PLUSSES:p {:RESULT=p;:}
  | MINUSES:mi {:RESULT=mi;:}
  | UNIONS:u {:RESULT=u;:}
```

```
  | EXCEPTS:e {:RESULT=e;:}
  | ORS:o {:RESULT=o;:};
PLUSSES
  ::= MULTIPLICATIVEEXPR:m _plus MULTIPLICATIVEEXPR:mm
    {:RESULT=new ExprCommon(m, mm, ExprCommon.PLUS);:}
  | PLUSSES:p _plus MULTIPLICATIVEEXPR:m
    {:RESULT=new ExprCommon(p, m, ExprCommon.PLUS);:};
MINUSES
  ::= MULTIPLICATIVEEXPR:m _minus MULTIPLICATIVEEXPR:mm
    {:RESULT=new ExprCommon(m, mm, ExprCommon.MINUS);:}
  | MINUSES:mi _minus MULTIPLICATIVEEXPR:m
    {:RESULT=new ExprCommon(mi,m,ExprCommon.MINUS);:};
UNIONS
  ::= MULTIPLICATIVEEXPR:m _union MULTIPLICATIVEEXPR:mm
    {:RESULT=new ExprCommon(m, mm, ExprCommon.UNION);:}
  | UNIONS:un _union MULTIPLICATIVEEXPR:m
    {:RESULT=new ExprCommon(un, m, ExprCommon.UNION);:};
EXCEPTS
  ::= MULTIPLICATIVEEXPR:m _except MULTIPLICATIVEEXPR:mm
    {:RESULT=new ExprCommon(m, mm, ExprCommon.EXCEPT);:}
  | EXCEPTS:e _except MULTIPLICATIVEEXPR:m
    {:RESULT=new ExprCommon(e, m, ExprCommon.EXCEPT);:};
ORS
  ::= MULTIPLICATIVEEXPR:m _concatenate MULTIPLICATIVEEXPR:mm
    {:RESULT=new ExprCommon(m, mm, ExprCommon.CONCATENATE);:}
  | ORS:o _concatenate MULTIPLICATIVEEXPR:m
    {:RESULT=new ExprCommon(o, m, ExprCommon.CONCATENATE);:};
MULTIPLICATIVEEXPR
  ::= INEXPR:i {:RESULT=i;:}
  | MULS:m {:RESULT=m;:}
  | DIVS:d {:RESULT=d;:}
  | MODS:m {:RESULT=m;:}
  | INTERSECTS:ins {:RESULT=ins;:};
MULS
  ::= INEXPR:i _mult INEXPR:ii
    {:RESULT=new ExprCommon(i,ii, ExprCommon.MULT);:}
  | MULS:m _mult INEXPR:i
    {:RESULT=new ExprCommon(m,i, ExprCommon.MULT);:};
DIVS
  ::= INEXPR:i _div INEXPR:ii
    {:RESULT=new ExprCommon(i, ii, ExprCommon.DIV);:}
  | DIVS:d _div INEXPR:i
    {:RESULT=new ExprCommon(d, i, ExprCommon.DIV);:};
MODS
  ::= INEXPR:i _mod INEXPR:ii
    {:RESULT=new ExprCommon(i, ii, ExprCommon.MOD);:}
  | MODS:m _mod INEXPR:i
    {:RESULT=new ExprCommon(m, i, ExprCommon.MOD);:};
INTERSECTS
  ::= INEXPR:i _intersect INEXPR:ii
    {:RESULT=new ExprCommon(i, ii, ExprCommon.INTERSECT);:}
  | INTERSECTS:ins _intersect INEXPR:i
```

```
    {:RESULT=new ExprCommon(ins, i, ExprCommon.INTERSECT);:};
INEXPR
  ::= UNARYEXPR:e {:RESULT=e;:}
  | INEXPR:i _in UNARYEXPR:e
    {:RESULT=new ExprCommon(i, e, ExprCommon.IN);:};
UNARYEXPR
  ::= _plus UNARYEXPR:u
    {:RESULT=new ExprCommon(u, ExprCommon.UNARYPLUS);:}
  | _minus UNARYEXPR:u
    {:RESULT=new ExprCommon(u, ExprCommon.UNARYMINUS);:}
  | _abs UNARYEXPR:u
    {:RESULT=new ExprCommon(u, ExprCommon.ABS);:}
  | _not UNARYEXPR:u
    {:RESULT=new ExprCommon(u, ExprCommon.NOT);:}
  | POSTFIXEXPR:p {:RESULT=p;:};
POSTFIXEXPR
  ::= PRIMARYEXPR:p {:RESULT=p;:}
  | PRIMARYEXPR:p INDEXES:i
    {:RESULT=new ExprCommon(p,i, ExprCommon.INDEXED);:}
  | PRIMARYEXPR:p DOTEXPR:d
    {:RESULT=new ExprCommon(p,d, ExprCommon.DOTEXPR);:};
INDEXES
  ::= _brackleft INDEX:i _brackright
    {:RESULT=new ExprCommon(i, null, ExprCommon.VALUELIST);:}
  | _brackleft INDEX:i _brackright INDEXES:is
    {:RESULT=new ExprCommon(i, is, ExprCommon.VALUELIST);:};
DOTEXPR
  ::= _dot _identifier:i ARGLIST:a DOTEXPR:d
    {:RESULT=new ExprCommon(a, d,i, ExprCommon.DOTEXPR);:}
  | _dot _identifier:i ARGLIST:a
    {:RESULT=new ExprCommon(a, null,i, ExprCommon.DOTEXPR);:}
  | _dot _identifier:i DOTEXPR:d
    {:RESULT=new ExprCommon(null, d,i, ExprCommon.DOTEXPR);:}
  | _dot _identifier:i
    {:RESULT=new ExprCommon(null, null,i,
    ExprCommon.DOTEXPR);:}
  | _rightarrow _identifier:i ARGLIST:a DOTEXPR:d
    {:RESULT=new ExprCommon(a, d,i, ExprCommon.DOTEXPR);:}
  | _rightarrow _identifier:i ARGLIST:a
    {:RESULT=new ExprCommon(a, null,i, ExprCommon.DOTEXPR);:}
  | _rightarrow _identifier:i DOTEXPR:d
    {:RESULT=new ExprCommon(null, d,i, ExprCommon.DOTEXPR);:}
  | _rightarrow _identifier:i
    {:RESULT=new ExprCommon(null, null,i,
    ExprCommon.DOTEXPR);:};
INDEX
  ::= EXPRLIST:c {:RESULT=c;:}
  | EXPR:e _colon EXPR:ee
    {:RESULT=new ExprCommon(e, ee, ExprCommon.COLON);:};
EXPRLIST
  ::= EXPR:e
    {:RESULT=new ExprCommon(e, ExprCommon.VALUELIST);:}
```

```
  | EXPR:e _comma EXPRLIST:c
    {:RESULT=new ExprCommon(e, c, ExprCommon.VALUELIST);:};
ARGLIST
  ::= _parenleft _parenright
    {:RESULT=new ExprCommon(null, ExprCommon.VALUELIST);:}
  | _parenleft VALUELIST:v _parenright {:RESULT=v;:};
PRIMARYEXPR
  ::= _parenleft QUERY:q _parenright {:RESULT=q;:}
  | _listtoset _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.LISTTOSET);:}
  | _element _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.ELEMENT);:}
  | _distinct _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.DISTINCT);:}
  | _flatten _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.FLATTEN);:}
  | _first _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.FIRST);:}
  | _last _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.LAST);:}
  | _unique _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.UNIQUE);:}
  | _exists _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.EXISTS);:}
  | _sum _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.SUM);:}
  | _min _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.MIN);:}
  | _max _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.MAX);:}
  | _avg _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.AVG);:}
  | _count _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.COUNT);:}
  | _count _parenleft _mult _parenright
    {:RESULT=new ExprCommon(new ExprCommon(null, null, "*",
    ExprCommon.FIELDLIST), ExprCommon.COUNT);:}
  | _is_undefined _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.IS_UNDEFINED);:}
  | _is_defined _parenleft QUERY:q _parenright
    {:RESULT=new ExprCommon(q, ExprCommon.IS_DEFINED);:}
  | _identifier:i _parenleft FIELDLIST:f _parenright
    {:RESULT=new ExprObjectConstruction(i,f);:}
  | _identifier:i _parenleft SELECTEXPR:s _parenright
    {:RESULT=new ExprObjectConstruction(i, s);:}
  | _struct _parenleft FIELDLIST:f _parenright
    {:RESULT=new ExprStructConstruction(f);:}
  | _array _parenleft _parenright
    {:RESULT=new ExprCollectionConstruction("array");:}
  | _array _parenleft VALUELIST:v _parenright
    {:RESULT=new ExprCollectionConstruction("array", v);:}
  | _set _parenleft _parenright
```

```
    {:RESULT=new ExprCollectionConstruction("set");:}
  | _set _parenleft VALUELIST:v _parenright
    {:RESULT=new ExprCollectionConstruction("set", v);:}
  | _bag _parenleft _parenright
    {:RESULT=new ExprCollectionConstruction("bag");:}
  | _bag _parenleft VALUELIST:v _parenright
    {:RESULT=new ExprCollectionConstruction("bag", v);:}
  | _list _parenleft _parenright
    {:RESULT=new ExprCollectionConstruction("list");:}
  | _list _parenleft VALUELIST:v _parenright
    {:RESULT=new ExprCollectionConstruction("list", v);:}
  | _list _parenleft EXPR:e _dotdot EXPR:ee _parenright
    {:RESULT=new ExprCollectionConstruction("list", new
    ExprCommon(e, ee, ExprCommon.LISTRANGE));:}
  | _identifier:i {:RESULT=new ExprCommon(null,null,i,
    ExprCommon.IDENTIFIER);:}
  | _identifier:i ARGLIST:a
    {:RESULT=new ExprCommon(a, null, i,
    ExprCommon.IDENTIFIER);:}
  | _queryparam:q {:RESULT=new ExprQueryParam(q);:}
  | LITERAL:l {:RESULT=l;:};
FIELDLIST
  ::= FIELD:f {:RESULT=f;:}
  | FIELD:f _comma FIELDLIST:fl
    {:((ExprCommon)f).right=fl; RESULT=f;:};
FIELD
  ::= _identifier:i _colon EXPR:e
    {:RESULT=new ExprCommon(e, null, i,
    ExprCommon.FIELDLIST);:};
VALUELIST
  ::= EXPR:e
    {:RESULT=new ExprCommon(e, ExprCommon.VALUELIST);:}
  | EXPR:e _comma VALUELIST:v
    {:RESULT=new ExprCommon(e, v, ExprCommon.VALUELIST);:};
TYPE
  ::= _short {:RESULT=new Typedef("short");:}
  | _unsigned _short {:RESULT=new Typedef("unsigned short");:}
  | _long {:RESULT=new Typedef("long");:}
  | _unsigned _long {:RESULT=new Typedef("unsigned long");:}
  | _long _long {:RESULT=new Typedef("long long");:}
  | _float {:RESULT=new Typedef("float");:}
  | _double {:RESULT=new Typedef("double");:}
  | _char {:RESULT=new Typedef("char");:}
  | _string {:RESULT=new Typedef("string");:}
  | _boolean {:RESULT=new Typedef("boolean");:}
  | _octet {:RESULT=new Typedef("octet");:}
  | _enum IDENTIFIERS:i
    {:RESULT=new Typedef("enum",i, null, null);:}
  | _date {:RESULT=new Typedef("date");:}
  | _time {:RESULT=new Typedef("time");:}
  | _interval {:RESULT=new Typedef("interval");:}
  | _timestamp {:RESULT=new Typedef("timestamp");:}
```

```
  | _set _lt TYPE:t _gt
    {:RESULT=new Typedef("set", null, t, null);:}
  | _bag _lt TYPE:t _gt
    {:RESULT=new Typedef("bag", null, t, null);:}
  | _list _lt TYPE:t _gt
    {:RESULT=new Typedef("list", null, t, null);:}
  | _array _lt TYPE:t _gt
    {:RESULT=new Typedef("array", null, t, null);:}
  | _dictionary _lt TYPE:t _gt
    {:RESULT=new Typedef("dictionary", null, t, null);:};
IDENTIFIERS
  ::= _identifier:i {:RESULT = i;:}
  | IDENTIFIERS:ii _dot _identifier:i
    {: RESULT = ii + "." + i;:};
LITERAL
  ::= _true {:RESULT=new ExprLiteral(new Boolean("TRUE"));:}
  | _false {:RESULT=new ExprLiteral(new Boolean("FALSE"));:}
  | _longliteral:l {:RESULT=new ExprLiteral(new Integer(l));:}
  | _doubleliteral:d
    {:RESULT=new ExprLiteral(new Double(d));:}
  | _charliteral:c
    {:RESULT=new ExprLiteral(new Character(c.charAt(0)));:}
  | _stringliteral:s {:RESULT = new ExprLiteral(s);:}
  | _date _dateliteral:d
    {: RESULT=new ExprLiteral
    ( java.sql.Date.valueOf( d.replace('\'',' ') .trim() ));:}
  | _time _timeliteral:t
    {: String[] s = t.replace( '\'', ' ' ).trim().split(":");
    long i = (((Long.parseLong(s[0])-1) * 60) +
    Long.parseLong(s[1]))* 60 * 1000;
    if (s[2].indexOf(".")!=-1){
      String[] ssplit = s[2].split("\\.");
      i += Long.parseLong(ssplit[0]) * 1000;
      i += Long.parseLong((ssplit[1] + "00").substring(0, 3));
    }else i += Long.parseLong(s[2]) * 1000;
    RESULT=new ExprLiteral( new java.sql.Time(i));:}
  | _timestamp _timestampliteral:t
    {: RESULT=new ExprLiteral(
    java.sql.Timestamp.valueOf(t.replace('\'', ' ').trim()
    ));:}
  | _nil {:RESULT=new ExprLiteral(null);:}
  | _undefined {:RESULT=new ExprLiteral(new Undefined());:};
```

## 6.4 Kildekode

Kildekoden til hele implementasjonen er tilgjengelig i følgende zip-fil:

http://folk.uio.no/~oivindha/OQL.zip

## 6.5 Unntakshierarkiet i ODMG 3.0

Slik er ODMGs unntak-hierarki:

- `ODMGException`
    - `DatabaseNotFoundException`
    - `DatabaseOpenException`
    - `ObjectNameNotFoundException`
    - `ObjectNameNotUniqueException`
    - `QueryException`
        - `QueryInvalidException`
        - `QueryParameterCountInvalidException`
        - `QueryParameterTypeInvalidException`
- `ODMGRuntimeException`
    - `ClassNotPersistenceCapableException`
    - `DatabaseClosedException`
    - `DatabaseIsReadOnlyException`
    - `LockNotGrantedException`
    - `NotImplementedException`
    - `ObjectDeletedException`
    - `ObjectNotPersistentException`
    - `TransactionAbortedException`
    - `TransactionInProgressException`
    - `TransactionNotInProgressException`

De fleste av disse er allerede implementert i OSJI. De resterende unntaksklassene blir programmert rett frem. Et eksempel følger:

```
package com.odi.odmg;

public class QueryException
    extends ODMGException
{
```

```
    public QueryException( String arg0 )
    {
        super( arg0 );
    }
    public QueryException( )
    {
        super( "" );
    }
}
```

# 7 Bibliografi

1. **Catell, Roderic Geoffrey Galton; Barry, Douglas K.** (2000)
   *The Object Data Standard: ODMG 3.0*
   Morgan Kaufmann Publishers
2. **Claypool, Kajal T.; Jin, Jing; Rundensteiner, Elke Angelica** (1998)
   *OQL SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework*
   http://delivery.acm.org/10.1145/790000/783169/p9-claypool.pdf
3. **Codd, Edgar Frank** (August 1969)
   *Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks*
4. **Codd, Edgar Frank** (Juni 1970)
   *A Relational Model of Data for Large Shared Data Banks*
   CACM 13 utgivelse 6
5. **Klein, Gerwin** (2005)
   *JFlex User's Manual*
   http://www.jflex.de/manual.htm
6. **Louden, Kenneth C.** (1997)
   *Compiler Construction, Principles and practice*
   PWS Publishing Company
7. **Mikhalenko, Peter V.** (November 2006)
   *Look inside the Java Reflection class*
   http://articles.techrepublic.com.com/5100-3513-6132941.html
   (lest 3 mars 2008)
8. **Olsen, Russ** (Mars 2007)
   *Reflections on Java Reflection*
   O'Reilly
   http://www.onjava.com/pub/a/onjava/2007/03/15/reflections-onjava-reflection.html
   (lest 25 februar 2008)
9. **Panda, Debu** (2006)
   *Standardizing Java Persistence with the EJB3 Java Persistence API*
   http://www.onjava.com/pub/a/onjava/2006/05/17/standardizingwith-ejb3-java-persistence-api.html
   (lest 18 februar 2008)
10. **Progress** (2003)
    *Java API Userguide (for ObjectStore 6.0)*
11. **Wikipedia**
    *Generics in Java*
    http://en.wikipedia.org/wiki/Generics_in_Java
    (lest 4 mars 2008)

12. **Wikipedia**
    *Object Query Language*
    http://en.wikipedia.org/wiki/Object_Query_Language
    (lest 4 februar 2008)
13. **Wikipedia**
    *ObjectStore*
    http://en.wikipedia.org/wiki/ObjectStore
    (lest 2 januar 2008)
14. **Wikipedia**
    *Test-driven development*
    http://en.wikipedia.org/wiki/Test-driven_development
    (lest 28 mars 2008)