Datalogi, modul 1
Roskilde Universitetcenter
Vejleder: Mads Rosendahl

Compilerteknik, fra Java til Assembler

Efterår 1999

version 1.01

Forfattet af:
Kasper B. Graversen

vha LaTeX2e

Resumé

Projektet udgangspunkt var følgende tese:

Ved at anvende en compiler, der kan oversætte et Java program til native kode, kan programmer afvikles hurtigere end JDK's virtual machine. Dette gælder både med hensyn til opstart, og den faktiske udførsel også selvom den producerede kode ikke optimeres.

Compileren, dette projekt præsenterer, kan derfor anvendes til compilering af små nyttige programmer, der herved opnår at være mindre ressourcekrævende og hurtigere i deres udførsel.

Udfra dette skabes et programmeringssprog, Qjava, der er en lille delmængde af Java. Endvidere realiseres en Qjava compiler i programmeringsproget Java, der kunne oversætte Qjava kode til assembler, der efterfølgende kunne assembleres og afvikles på en DOS maskine.

Igennem stærk afgrænsning af sprog, fejlkontrol mv. lykkedes det at skabe en velfungerende compiler. Idéen bag compileren er, at brugerens programmer skal skabes og kunne fungere i JDK. Qjava compileren skal først anvendes, når brugeren ønsker en hurtigere afvikling af sit færdige program. Således skal javaprogrammer, der er indeholdt i Qjava, og som kan compileres i JDK også kunne compileres i Qjava compileren.

Efter realiseringen af compileren blev et regnetungt primtalsprogram testet i henholdsvis JDK og i Qjava. Resultatet var, at JDK afviklede koden hurtigere, end den uoptimerede assembler Qjava compileren kunne generere! Tesen, der var igangsætter af projektet, var falsificeret.

Rapporten konkluderer herefter:

Anvendeligheden af Qjava compileren, set i et tidsbesparelses-perspektiv, ikke er specielt stor. Der skal implementeres meget effektive optimeringsalgoritmer, hvis målsætningen med Qjava compileren skal opfyldes.

Forord

Med udgivelsen af denne nye version af projektrapporten, håber jeg at de fleste fejl der havde sneget sig ind i afleveringsversionen er elimineret. Endvidere har erfaringer fra den mundlige eksamination medført to nye appendix. Det har dog været mit mål med denne nye version, at lade rapporten fremstå som jeg gerne ville have haft afleveret den, fremfor at omrevidere den.

I forbindelse med den mundlige eksamen (i januar 2000), oversatte jeg primtalsprogrammet til C og compilerede programmet i Microsofts Visual C++ '97 som gav en exe-fil med en køretid på 8 sekunder. Herefter programmerede jeg en assemblerversion "som en assembler programmør ville have skrevet programmet" (koden findes i appendix L side 155). Programmet havde en køretid på 5 sekunder. Set i dette lys er Qjava compilerens performance (medtaget de to små foreslåede optimeringer), ikke så ringe som antydet i rapporten.

Versionshistorie:

Version 1.0, 23 dec. 1999 Den oprindelige version afleveret til eksamination.

Version 1.01, 4 feb. 2000 Der er rettet stave og kommafejl, samt en mindre redigering af fordækte sætningskonstruktioner. Endvidere er henvisninger korrigeret og figurer justeret. Endelig er appendix K, L tilføjet, der viser udsnit af Qjava compilerens kodegenerering (af primtalstest programmet) og en håndskrevet assembler version af samme program.

Rapporten er trods sin længde og omfang blevet udfærdiget på normeret tid, dvs. 1/2 semester. For dem der kunne have interesse, kan det til slut nævnes, at rapporten blev bedømt til 13.

Jeg håber du vil få glæde og inspiration ved læsning af rapporten
–Kasper B. Graversen, feb. 2000

Indhold

— Kapitel

1 Indledning

En compiler læser et program skrevet i et sprog, og oversætter det til et ækvivalent program i et andet sprog. Tilbage i 1960'erne, var dét at skabe en compiler betragtet som svært. Aho beretter i bogen "Compilers — Principles, techniques, and tools", at den første Algol-compiler tog 18 mandeår at udfærdige [Aho et al;1986]. Med tiden har man dog dels fået udviklet en struktureret måde at nedfælde grammatikker, dels fået beskrevet de generelle faser en compiler gennemgår i et oversættelsesforløb. Således er det i dag muligt for en studerende, at skrive en compiler af rimeligt omfang på et semester.

Til dette projekt knytter der sig følgende tese, som var udgangspunkt for projektet:

> *Ved at anvende en compiler, der kan oversætte et Java program til native kode, kan programmer afvikles hurtigere end JDK's virtual machine. Dette gælder både med hensyn til opstart, og den faktiske udførsel også selvom den producerede kode ikke optimeres.*
>
> *Compileren dette projekt præsenterer, kan derfor anvendes til compilering af små nyttige programmer, der herved opnår at være mindre ressourcekrævende og hurtigere i deres udførsel.*

At skabe en compiler på et semester, der tilnærmelsesvis ligner Java, er fuldstændigt urealistisk. Den eneste måde dette kan lade sig gøre på, er ved hårdhændet at anvende følgende to begræsninger, trods de mange fristelser for at fordybe sig i små specifikke områder:

- Compilerens kontrol for fejl nedtones kraftig.
- Der implementeres kun en lille delmængde af Javasproget.

1.1 Kravsspecifikation

Målet med projektet er at skabe en compiler i Java, der kan oversætte en delmængde af Java til Intel 80286 kompatibel assembler. Assemblerkoden skal kunne afvikles efter assemblering og linkning med en ekstern assembler (i projektet anvendes TASM fra Borland). Delmængden af Java bliver i rapporten refereret til som "Qjava", mens compileren, der konstrueres, går under navnet "Qjava compileren".

Projektets fokus er at skabe en compiler, der kan producere assemblerkode som kan afvikles. Der ses bort fra effektivitetsspørgsmål, både med hensyn til compillerens effektivitet og effektiviteten af den genererede kode. Der ses bort fra fejl ved oversættelsen (typefejl) og dynamiske fejl (overflow, division med 0 osv). De eneste fejl, der detekteres, er syntaks fejl, og et mindre antal scopefejl (dvs. om en variabel kan anvendes i det rigtige scope). Således skal javaprogrammer, der er indeholdt i Qjava og som kan kan compileres i JDK, også kunne compileres i Qjava compileren, der her præsenteres. Ydermere lægges der ikke vægt på en eksakt formel præcisering af Qjava. I kapitel 4 side 24 uddybes afgrænsningen af QJava.

1.1.1 Fremgangsmåde

For at kunne skabe Qjava compileren, argumenteres og opstilles en grammatik for Qjava, som compileren herefter følger. Slutteligt findes samt opstilles ækvivalenter mellem sprogene Java og assembler, som kodegeneratordelen slutteligt anvender.

Overordnet for rapporten gælder at den sigter på at vurdere og diskutere forskellige implementationsmuligheder, fremfor udelukkende at fremlægge "den optimale løsningsmodel".

1.1.2 Implementering

Kriterierne for implementationen af compileren, udover at den skal realiseres i programmeringssproget Java, er, at det søges udført udfra dels objekt orienterede principper, dels på baggrund af, hvad der er muligt at gøre elegant i Java. Disse mål konkretiseres igennem følgende 5 punkter:

Indkapsling Indkapsling af funktionalitet ved brug af klasser, der igennem restriktioner (access modifiers) kun giver brugeren adgang til klassens overordnede funktionalitet.

Polymorfisme Polymorfisme, der igennem nedarvning lader programmeringssproget finde og udføre de passende funktioner.

Begrænset nedarvning Brugen af nedarvning begrænses så compileren ikke udformes i "stive" klasser.

Designmønstre Til løsning af kravsspecifikationen, indgår anvendelsen af design mønstret Facade.

Nuanceret klassedefinition Der anvendes nuancerede klassedefinitioner, det vil sige almindelige klasser, abstrakte klasser, samt interfaces.

1.1.3 Målgruppe

Læseren forudsættes at være velbefærdiget i Java, da beskrivelser og forklaringer af Javas funktionalitet og udforming er udeladt. Forklaringer af de anvendte, samt foreslåede datastrukturer, er ligeledes udeladt. Yderligere vil kendskab til assembler være en fordel. Der findes et utal af lærebøger om algoritmer og compilerteknik, Java og assembler — denne rapport er ikke et forsøg på at efterligne nogle af dem.

1.2 Begrænsning

Afgrænsningen af Qjava er sket udfra to principper

- Qjava skal fremstå som "eksemplarisk" (jfr. "det eksemplariske princip", altså at belyse et givent område udfra et eksempel) hvorfor det skal være enkelt, så principperne, hvorpå det hviler, skinner igennem.
- Det skal være muligt at skrive små programmer i Qjava uden for mange krumspring.

Eller kortere: der søges en balance mellem "need to have" og "nice to have".

1.2.1 Need to have

Hvis principperne for sammenligning er vist igennem implementeringen af operatorene `<, <=`, giver implementeringen af operatorene `>, >=` ikke anledning til nyt stof. Det eneste der introduceres er mere sourcekode for Qjava compileren, der gør det sværere at danne sig et overblik, samtidig med, at der opstår flere potentielle fejlkilder. Følgende dele af Java er derfor udeladt, da de let kan substitueres af andre dele af Qjava.

`char, int:` substituerer typerne `boolean, byte, short, long, float, double`.

`while:` substituerer `for`, `do-while` løkker.

`if-else:` substituerer `if, switch-case` betingelsessætninger.

`<, <=:` substiturer operatorene `>, >=`.

`+, -:` substiturerer operatorene `++`, `--`.

Da Qjava compileren er underlagt klare udeladelser er følgende dele udeladt:

Semantik Da compileren ikke har nedarvning, udelades `instanceof` operatoren.

1-fil Da compileren udelukkende opererer på en sourcekodefil, udelukker dette: `import, package` med flere.

Sjælden anvendelse `continue` og `break` (med label) udelades, trods de let kan implementeres, da de repræsenterer dårlig programmeringsskik og sjældent anvendes.

Casting Casting er ikke medtaget, hverken til konvertering for simple typer (f.eks. fra `int` til `char`), eller til dynamisk typekontrol, da sidstnævnte kun kan forekomme hvis nedarvning eksisterer.

this `this` referencen er ikke medtaget.

Access modifiers Access modifiers (`public, private, final`) kan alle let implementeres, men for små programmer, er disse elementer ubetydelige, hvorfor de er udeladt.

Slutteligt er `abstract, try-catch-throws, synchronized` ikke implementeret.

Semantisk analyse, nedarvning, garbage collection, kode optimering er interessante facetter af en compiler, men er bevidst udladt jfr. 1.2 begrænsninger.

Fejl

Fejlmeldinger er i compileren holdt på et minimum, oftest med programstop til følge. Qjava compileren kan være mere tolerent overfor fejl end JDK (SUN's implementation af en javacompiler), men aldrig mere stringent — bortset fra det faktum, at Qjava i sig selv er en begrænsning i forhold til Java.

Idéen bag compileren er, at brugerens programmer skal skabes og kunne fungere i JDK. Qjava compileren skal først anvendes, når brugeren ønsker en hurtigere afvikling af sit færdige program.

Optimering

Der er i projektet ikke søgt at optimere hverken hukommelsesforbrug eller afviklingshastighed af compileren, da optimeringer oftest aldrig gør implementationen klarere. Står valget mellem to implementationsmuligheder, der har samme kompleksitet med hensyn til implementering, vælges dog i reglen den hurtigste.

1.2.2 Nice to have

Nedenstående viser hvilke operatorer, der kunne være udeladt, men som alligevel er inkluderet, da de dels gør programmeringen langt mindre besværlig, dels var lette at implementere.

`||, &&` kunne skrives med flere `if, while`.

`|, &` anvendes sjællent og kunne derfor udgå.

`<=, ==, !=` kunne skrives ved brug af `<`.

`-, *, /, %` kunne implementeres med `+`.

1.3 Beskrivelse af Qjava

Overordnet består Qjava af: Klasser, funktioner, felter og variable af typerne {char, int, String}, referencer til objekter, samt sætninger.

Bemærk at hverken nedarvning eller konstruktorer er implementeret for klasser. Som gældende i Java, må variabelnavne, funktionsnavne og klassenavne ikke indeholde "`.`", da "`.`" anvendes som separator af disse. Klassenavne forlanges i unikke, og skal være forskellige fra "String" og "Integer". Endvidere skal der forefindes én og kun en funktion ved navn "`main()`".

Sætninger

Af sætningstyper eksisterer: Tildelinginger, `if-else, while, break, return`, funktionskald i eget objekt og funktionskald i andet objekt (dog ikke flere niveauer som f.eks. `s.next.p()`), tilgang til variable i eget og i andre objekter.

Operatorer

Operatorerne er begrænset til:
`+, -, *, /, %, !, !=, <=, <, =, ==, ||, &&, |, &`

Variable og deres typer

Typerne er `char, int, String`, samt boolske udtryk. `char, int` er implementeret ved 16 bit, med et talområde på $\pm 2^{15}$ = ca. ± 32768, da 1 bit anvendes til repræsentation af fortegn. Dog er resultater fra `%` (modulo operationen) kun 8 bit stor pga. assemblers begrænsninger.

Da `this` referencen ikke er implementeret, må variables navne ikke være identiske i scopes der ligger inde i hinanden. Det vil sige, at felter ikke må være identiske med variable erklæret i funktioner eller som argumenter til funktioner. Variable og funktioners navne må være identiske.

Alle referencer til objekter kan antage værdien `null`

Kommentarer

Kommentarerne er standard Javakommentarer: `/* */` samt `// [EOL]`. Som i Java, understøttes ikke nestedede kommentarer.

`String` klassen

Javas `String` klasse er delvist implementeres, hvilket vil sige følgende funktioner (hvor `s1, s2` er af typen `String`, og `pos` er af typen `int`):

Tildeling	`s1 = "text string "`
Aflæsning	`s1.charAt(pos)`
Sammenkædning	s1 = `s1.contcat(s2);`

Bemærk at `StringBuffer` klassen ikke implementeres.

Andre standard Javapakker

Ud over `String` er en delmængde af `System.in` og `System.out` implementeret.

`int System.in.read()`	returnerer ASCII værdi af et tastetryk
`void System.out.print(String s)`	Udskriver `s` til skærm.

`System.out.print()` kan **ikke** udskrive `char` eller `int`, hvorfor metoden `static String Integer.toString(int)` skal anvendes. `n` af typen `int`, kan således udskrives ved følgende kald: `System.out.print( Integer.toString(n) );`

Afrunding

I forhold til modul 1 på datalogi/RUC's målsætning om at lære Java og forstå sprogets sammenhænge, passer dette projekt godt, da det netop er sproget og ikke kodegenereringen, der er fokus. Projektet er endvidere et godt oplæg til et modul 2 projekt, der kunne udbygge compileren med en semantik kontrol, advanceret kodegenerering, lager-administration og kodeoptimering.

1.4 Praktiske detaljer for rapporten

1.4.1 Typografi

For overblikkets skyld har typografien en semantisk betydning. `Mono-space font som dette` betegner programmeringskode, eller in- og out-put, der anvendes i forbindelse med kode eller test.

For tal gælder det, at ender de på `h` eller `b` betyder det, at tallet er skrevet i henholdsvis det hexadecimale eller binære talsystem. Tal uden bogstavendelse er skrevet i det decimale talsystem.

1.4.2 Sproganvendelse

Rapporten anvender i flæng danske og engelske gloser. Dette kan for nogen være et udtryk for dårlig dansk eller måske mangel på smag. Imidlertidigt er det et forsøg på at afværge brugen af alt for klodsede danske ord såsom "spildopsamling" og "indlejrede". Af samme grund er grammatikken for Qjava, samt al programmeringskoden holdt i engelsk.

Samtidig er håbet, at sproganvendelsen reducerer nogle af de tvetydigheder, der findes i det skrevne sprog. Når der anvendes "compiler" om oversættelsesprocessen som ét hele, kan "oversætte" anvendes i snævre situationer, hvor f.eks. et tegn bliver til et tal. Altså hvor der slet ikke er tale om samme 'oversættelses-apparat', som det "compiler" dækker over i denne rapport.

Felter og variable

Der skelnes i rapporten mellem variable erklæret i klasse scope og varible erklæret i funktioner/variable der er argumenter til funktioner. Førstnævnte benævnes "felter", mens de resterende kaldes "variable". Benævnes alle variable i en klasse kaldes disse "klassens variable".

1.4.3 Diagrammer

I de diagrammer, hvor det er meningsfuldt at tale om start- og slut-tilstande, er disse udformet som:

1.5 Rapportens opbygning

Kapitel 2 beskriver alle compilerens faser i korte træk, og gennemgår hvorledes Qjava koden er opbygget.

Kapitel 4 specificerer Qjava og opstiller en EBNF grammatik.

Kapitlerne 3–8 uddyber kapitel 2.

Kapitel 10 fastsætter grænserne for compilerens pålidelighed igennem afprøvning af compilerens faser.

Kapitel 11 prøver meget kort at sammenligne Qjava compileren med JDK.

Kapitel 12–13 afrunder projektet med perspektivering og konklusion, som vil præsentere erfaringer med kørsel af compileren.

I appendix forelægger hele sourcekoden til Qjava.

1.6 LaTeX

For de interesserede, er rapporten udfærdiget i LaTeX2e med hjælp fra følgende pakker:
Xy-pic til de fleste diagrammer.
De resterende illustrationer blev tegnet i JavaFig, og konverteret til ps med fig2dev
graphicx til inkludering af ps-billeder.
fancyhdr til definering af headings.
showkeys til debug af henvisninger.
Kapitel-layout blev manuelt fremstillet af undertegnede i en cls fil, der er at finde på internettet.

— Kapitel

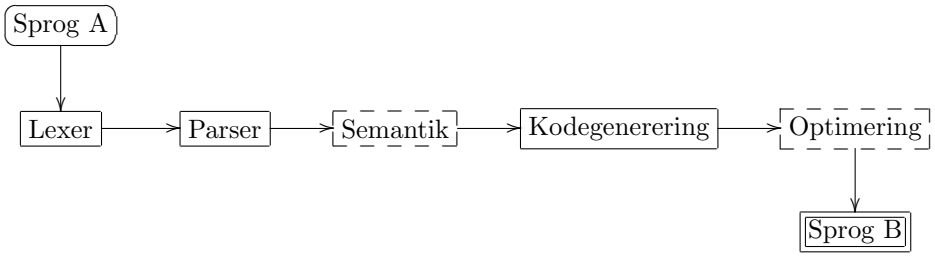
2 Compilerens struktur

Dette kapitel vil omhandle en compilers struktur, og i korte træk opridse hvorledes Qjava compileren er konstrueret.

- Leksikals analyse, sektion 2.1 side 10
- Parser, sektion 2.2 side 10
- Oversættelse, sektion 2.3 side 12
- Oversigt over implementationen, sektion 2.5 side 15

Oversigt

En compiler "programmeret efter bogen" gennemgår følgende stadier:



Figur 2.1: *Oversættelse fra kode A til kode B (en compilers indre funktionalitet).*

Figuren skal læses som programkode skrevet i sproget A, der bliver oversat via en compiler til programkode i sproget B. Grunden til "semantik" er stiplet er, at semantik kontrollen i Qjava compileren er meget begrænset, hvorfor der ikke er tale om en reel kontrolfase. "optimering" er ligeledes stiplet, men er ikke implementeret i Qjava compileren.

Generelt søger man at opdele oversættelsen i flere faser med hver sin simple funktionalitet, da dette drastisk mindsker kompleksiteten af opgaverne der skal løses, og dermed også sandsynligheden for fejl. Compileren gøres også mere portabel, da der ved en portering til en anden platform, let kan lokaliseres og udskiftes moduler, der ikke er kompatible med den pågældende platform [Aho et al;1986, side 84–85].

2.1 Leksikalsk analyse

Leksikalsk analyse (ofte forkortet til "en lexer") er compilerens første stadie, hvor indlæsningen finder sted. Under indlæsningen oversættes inddata til intern repræsentation, mens det søges at finde leksikaler, som kaldes tokens. Hver token er en selvstændig enhed, som f.eks. tal, ord og andre tegn. Formalisering er en klar fordel, da man på dette stadie én gang for alle eliminerer følgende elementer i oversættelsen:

Blanktegn Blanktegn (space/return/TAB) samt kommentarer er fjernet. Læses det næste token, skal der derfor ikke først "spoles forbi" eventuelle tegn.

Entydighed Nogle tokens er sammensatte af to eller flere tegn, der i sig selv kunne være tokens, eksempelvis `<=` eller `x2`. Disse regler, og undtagelser, skal der kun tages højde for et sted i programmet.

Kort sagt er alt overflødigt skåret fra og input er ikke tvetydigt.

På et implementationsplan vil et token oftest indeholde flere informationer, såsom ("tal", 12) altså, tokenet er et tal, og tallets værdi er 12. Eller ("streng", "hej hans"), altså tokenet er en streng, hvis værdi er "hej hans". Andre gange indeholder tokenet kun én information, såsom ("+") eller ("while"), der blot angiver at tokenet er et plus-tegn eller keywordet "while".

Lexeren er videre uddybet i kapitel 3 side 18.

2.2 Parser

Parserens funktion er at kontrollere, om værdierne i strømmen af tokens følger rækkefølgen dikteret af en grammatik. Produktet af denne kontrol er et parsertræ — en datastruktur, senere faser af compileren kan traversere. Parsertræet kan dog være implicit i den forstand, at der udelukkende returneres hvilke produktioner der er genkendt, altså "iterationsmodellen" beskrevet i sektion 2.4 side 13.

En måde at repræsentere et sådan parsertræ på, er ved et træ der har et blad for hver token, og en knude for hver grammatisk regel gennemgået ved parsningen. Et sådan parsertræ kaldes et konkret parsertræ, da det er en afbildning af den konkrete grammatik. Parsertræet vil dog indeholde en del overflødig information såsom `{`, `}`, `(`, `)`, `=` og `;`. Information, der er særdeles nyttig ved programmering og til parsning, men som er unødvendig på grund af parsertræets træstruktur.

Desuden er det konkrete parsertræ for afhængigt af grammatikken, da det vil indeholde overflødige non-terminaler og grammatik-produktioner der stammer fra fjernelse af venstrerekursion, dobbelttydighed mv. Sådanne størrelser skal holdes på grammatikplan og ikke forplumre de senere faser. Der vælges derfor et abstrakt parsertræ som repræsentationsmodel. Det abstrakte parsertræ er en væsenlig forsimplet udgave af det konkrete parsertræ, hvor en række struktureringer og omskrivningsregler er fjernet, dog på en sådan måde, at meningen er bevaret [Appel;1998, side 98].

2.2.1 Metasproget EBNF

Ved at beskrive sit programmeringssprog på et metaplan, i form af en grammatik, opnås en kort og formel beskrivelse af sproget. Til definering af Qjava anvedes EBNF notationsformen, der er en udbyggelse af BNF ved følgende metasymboler `{}, [], ()`.

Fordelen ved EBNF fremfor BNF er, at den eliminerer de fleste tomme produktioner (ϵ) samt rekursionskald, hvilket skaber overblik. Grammatikken ligger derved tættere på implementationen af parseren — f.eks. kan { `foobar` } implementeres direkte i Java med en while-løkke.

Fordelen ved EBNF ses tydeligt i nedenstående eksempel, hvor et udpluk af en grammatik er skrevet på begge former.

Eksempel #2.1:

vardef $\rightarrow$ ⟨id⟩ ⟨id⟩ ⟨vardef$_2$⟩ ⟨vardef$_3$⟩ “;”
vardef$_2$ $\rightarrow$ = ⟨E⟩ | ϵ
vardef$_3$ $\rightarrow$, ⟨id⟩ ⟨vardef$_2$⟩ ⟨vardef$_3$⟩ | ϵ
På EBNF form reduceres dette til:
vardef $\rightarrow$ ⟨id⟩ ⟨id⟩ [“=” ⟨E⟩] { “,” ⟨id⟩ [“=” ⟨E⟩] } “;”

Grammatikken kan være drilsk at opskrive på EBNF, til gengæld har parseren “skrevet sig selv”. Grammatikken har simpelthen specificeret programmeringssproget så akkurat, at det kan skrives direkte i Java, hvis der anvendes rekursiv nedstigning eller afarter heraf. Da BNF og EBNF notation varierer en smule i litteraturen, gennemgås kort den anvendte notationsform.

Symbol	Beskrivelse
⟨foo⟩	Angiver `foo` er en syntaks-enhed, altså en non-terminal.
{foo}	Noterer $0 \ldots n$ antal `foo`.
[foo]	Noterer 0 eller 1 antal `foo`.
(foo|bar)	Noterer enten `foo` eller `bar`.
$\rightarrow$	Betyder at venstresiden “defineres som” højresiden. Især i ældre litteratur anvendes `::=` istedet.
|	Skilletegn mellem forskellige produktioner eller definitioner af non-terminalen.
“bar”	Angiver, at `bar` er en terminal (en tegnsekvens), som den fremgår i input.

2.2.2 Rekursiv nedstigning

Parserens implementering sker ved teknikken rekursiv nedstigning. For hver nonterminal på venstresiden, svarer en metode i parseren med ækvivalent navn og funktionalitet. Produktioner på højresiden svarer til (muligvis rekursive) metodekald. Terminaler svarer til scanning af input. I kapitel 4 side 24 udformes grammatikken for Qjava på EBNF, mens kapitel 5 side 33 beskæftiger sig med implementering af parser og parsertræ.

2.3 Oversættelse

I sidste fase foretages en oversættelse af parsertræet til sprog B. Ofte vil sprog A og sprog B ikke indeholde samme funktionaliteter, hvorfor strukturene i sprog A må repræsenteres på bedst mulig vis i sprog B. For eksempel har assembler slet ikke det samme variabel eller klasse begreb som Java. Et andet men mere trivielt problem er, at Assembler kun accepterer højst to parametre pr. kald, så udtrykket `t = 1+2+3+4` skal sættes på formen

```
mov t, 1
add t, 2
add t, 3
add t, 4
```

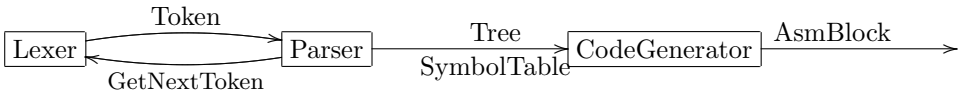
Oversættelsen beror på to grundpprincipper nemlig

- Sætninger: Kodegenerering af en række sætninger foregår kronologisk, hvor hver sætning kan indeholde kodegenerering, som igen genereres kronologisk. Dette gælder for alle sætninger med undtagelse af while-operatorens "krop", nemlig break og generelt for return, da disse skal kende til kode (en label) som findes senere i programmet.
- Udtryk: (variable, udregning, mv) er nettoresultatet altid en og kun en ekstra værdi på stakken.

2.3.1 Assembler

"Sprog B" er valgt til at være Intel 80286 kompatibel assembler, dog har hensigten været at generere så generel assembler som muligt. Det vil sige, der anvendes ikke specielle 80x86 kommandoer, samtidig er anvendelsen af multible hukommelsessegmenter og andre DOS specifikke ting søgt elimineret. tabel 7.1 side 44 forklarer de mest anvendte operationer.

For yderligere at forsimple oversættelsesprocessen, implementeres alle variable i 16 bit. Dette skaber nogle begrænsninger for f.eks. `int`, mens omvendt `char` og `String` repræsenteres ved for mange bits, da kun de første 8 bit anvendes, og specielt for boolskeværdier er ligeledes repræsenteres ved 16 bit.



Figur 2.2: *Oversigtsbeskrivelse af klasserne i Qjava compileren.*

Kapitlerne 7–8 uddyber yderligere.

2.4 Overordnet implementationsovervejelser

De beskrevne faser Lexer, Parser og Oversættelse svarer til klasserne `Lexer`, `Parser` og `CodeGenerator` i implementationen. Det overordnede forløb af faserne kan foregå ligefra en total integration af faserne til en total opsplittelse af dem.

Iterationsmodellen: Faserne kommunikerer med hinanden. Kodegeneratoren beder parseren om et parsertræ. Parseren bliver nu aktiveret og beder lexeren om tokens, et af gangen, til der er tokens nok til at kunne opbygge parsertræet. For hver gang lexeren kaldes, læses fra input indtil et token er genkedt der kan returneres. Det hele foregår iterativt mellem de enkelte faser.

Vandfaldsmodellen: Faserne forløber sekventielt. Lexeren indlæser al input som f.eks. gemmes som en liste af tokens. Parseren parseren en liste af tokens og gemmer et parsertræ. Kodegeneratoren traverserer et parsertræ og genererer kode udfra dette. Den ene faser stopper helt før den næste igangsættes. I et ekstremt tilfælde ville lexeren, parseren og kodegeneratoren være hver sit selvstændige program der hver kunne læse input og skrive deres produkt f.eks. på harddisken.

I implementation blev begge modeller anvendt, hvor iterationsmodellen anvendes mellem lexer og parser faserne, mens vandfaldsmodellen anvendes mellem parser og oversættelsesfaserne, som skitseret på figur 2.2 side 13. Iterationsmodellen blev valgt for implementationen af lexeren, der konkret betød den skulle returnere ét token af gangen fremfor en liste af tokens. Dette valg blev taget udfra følgende

- Tokens anvendes udelukkende til opbygning af parsertræet, hvor den videre manipulation finder sted. Hvert token bliver derfor kun tilgået en gang.
- Hvis der er syntax eller semantikfejl i programmet, stoppes indlæsningen præcis hvor fejlen opstod, og ikke først efter hele programmet er blevet analyseret.

Valget på vandfaldsmodellen passer for de sidste faser langt bedre, da der hér skal foretages flere kontroller og manipulationer. Kan parseren opbygge et korrekt

parsertræ, er programkoden syntaktisk korrekt. Herefter skal der foretages kontrol af typer (semantikkontrol) og optimering. For kodegenereringen vil det oftest også være praktisk eksempelvis at kunne indlæse en hel metode af gangen, for at kunne producere mindre og hurtigere kode. Symboltabellen kunne indbygges i parsertræet, men ville dels forplumre fokus af den stringente træstruktur, dels ville den besværliggøre gennemsøgning og opdateringen. Igen anvendes princippet om opsplitning og modularitet.

Kodegeneratoren anvender hjælpeklassen `AsmBlock` der vedligeholder en blok assemblerkode. Kodegeneratoren konstruerede fem instanser af klassen, to til henholdsvis data- og code- segmenterne, en til implementationen af standardfunktioner, og en AsmBlock indeholdende main-metoden og slutteligt en blok til generering af opstartskoden. Blokkene blev efter endt generering gemt en af gangen.

2.4.1 Designmønstre

Ved implementeringen af compileren, har brugen af designmønstre været overvejet og delvist anvendt. Beskrivelserne og overvejlserne med hensyn til designmønstrene er meget kortfattede, der henvises til letlæselig forklaring af designmønstrene i [Gamma et al;1995].

Singleton

Singleton designmønstret kunne man implementere for alle faserne, således der kun kunne eksistere en instans af hver fase. Designmønstret blev ikke implementeret, selvom man kunne argumentere, at compileren ikke ville give mening, at have to parser-faser på en gang. Omvendt vil implementeringen af singleton betyde begrænsninger af brugbarheden af compileren. Compileren ville f.eks. ikke være anvendelig som back-end for en grafisk brugergrænseflade, hvor man ønskede at compilere flere klasser på en gang (især på dual processor computere ville dette være hensigtsmæssigt).

Visitor

Visitorfacaden kan fjerne funktionalitet fra parsertræet og placere det i selvstændige klasser. Parsertræet forsimples og bliver overskueligt, men vigtigere, nye operationer kan implementeres og udføres på instanser af parsertræet uden at ændre på parsertræets klasser.

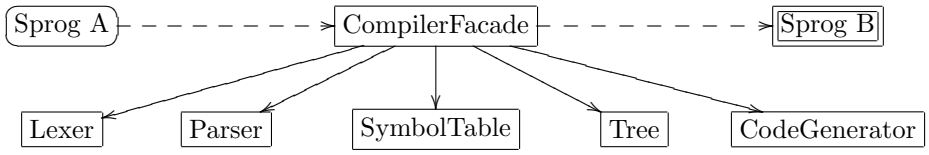
Dette vil være særligt anvendeligt hvis compileren skulle udføre type kontrol, kode optimering, flow analyse, eller skulle kontrollere om variable har fået tildelt værdi før de anvendes. Normalt skulle alle disse funktioner nedarves fra superklassen og implementeres. Med visitor kan ny funktionalitet udføres igennem en visitormetode.

Visitor ville være perfekt til videre udvikling af compileren, men implementeres ikke jævnfør kravsspecifikationen.

Facade

Facade anvendes til styring af faserne, og kommunikationen mellem klasserne. På denne måde skabes et interface eller en facade, der gør brugen af undersystemer lettere og mere overskuelig for programmøren. Facaden skabes ved at instantiering af faserne styres fra en klasse. Skal f.eks. parseren anvende en lexer, kaldes parseren med et lexer objekt som argument. Typisk vil output fra faserne returneres til facaden, der sørger for at sende det videre til næste fase.

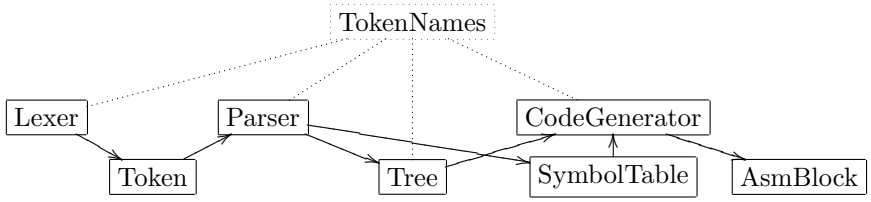
Ved anvendelse af facade designmønstret skabes der, på implementationsniveau, orden og orverblik over hvorledes de enkelte klasser kommunikerer med hinanden og i hvilken rækkefølge de udføres. Facade designmønstret er beskrevet i [Gamma et al;1995, side 185–193]. Mens realiseringen findes i appendix B side 97.



Figur 2.3: *Oversigtsbeskrivelse af klasserne med Facade designmønstret implementeret*

2.5 Kode oversigt

Figur 2.4 side 15, illustrerer hvorledes kommunukationen mellem Qjava compilerens klasser. `TokenNames` er et interface, som alle klasser forbundet med stiplede linier nedarver fra. Specielt for `Tree`-klassen (samt underklasser) gælder det, at kun deres `toString()` metoder anvender `TokenNames` interfacet.



Figur 2.4: *Kommunikation mellem faserne, hvor hver kasse samtidig repræsenterer en klasse i implementationen med samme navn. De stiplede linier angiver hvilke klasser der nedarver den* `TokenNames`*, mens pilene angiver in- og out- put mellem klasserne.*

Med udgangspunkt i figur 2.5 side 17 beskrives hvorledes Qjava compileren fungerer på sourcekode niveau. Indlæsning af data forgår med funktionen `getNextToken()` i `Lexer`, som returnerer det næste genkendte token. `Token`-klassen blev skabt som en protokol mellem `Lexer` og `Parser`, da kommunikationen indeholder mere end

én information. Klassen indeholder kun plads til data, og er således ækvivalent med en struct i C/C++.

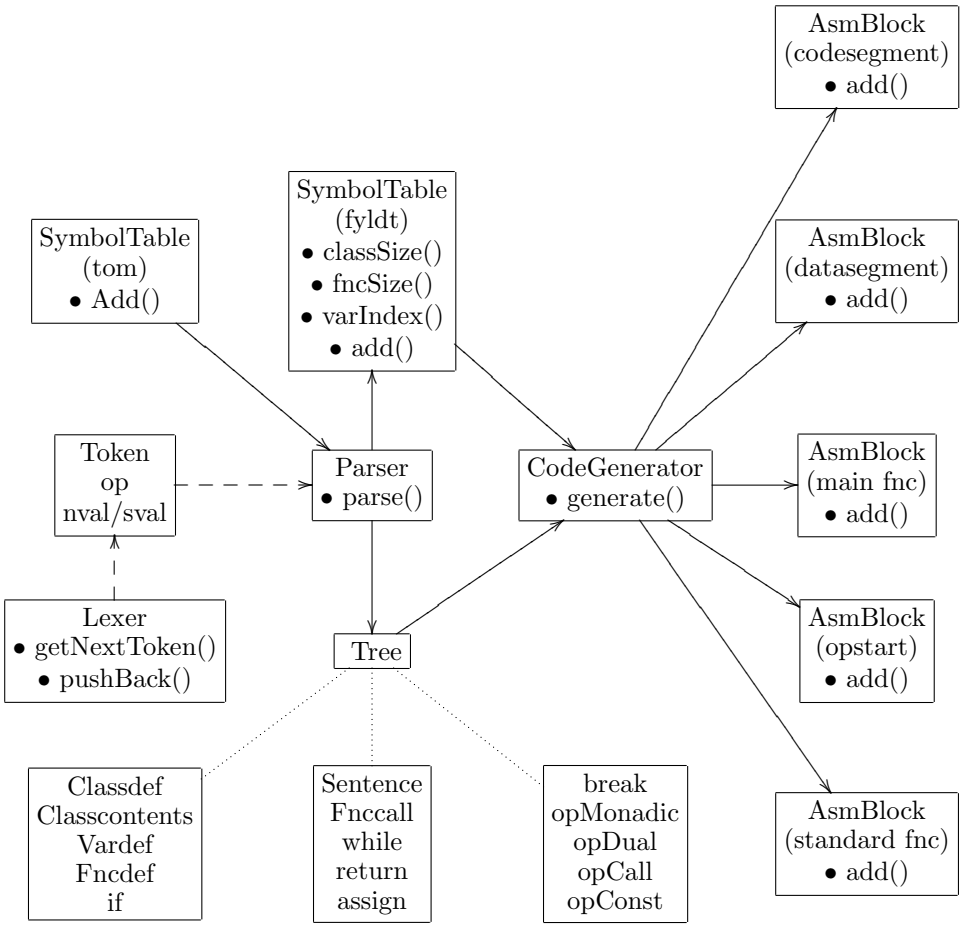
`id` feltet i `Token` kan eventuelt kombineret med `nval` eller `sval` felterne, og angiver dermed præcist det konkrete token. `id` indeholder en værdi defineret i interface `TokenNames`. `TokenNames` er et simpelt interface, der knytter en unik værdi til hver type token i QJava.

Parseren opbygger, udfra tokens, dels et parsertræ, dels en symboltabel. Instantieringen af klassen foregår med kaldet `new Parser(lexer, symboltable)`, mens opbygningen af parsertræet, sker med funktionen `parse()`, der returnerer træet efter endt parsning. Parsertræet er af typen `Tree`, og indeholder alle venstresiderne i QJavas grammatik i form af nedarvede klasser. Klassernes konstruktorer opbygger automatisk parsertræet. De stiplede linier viser klasserne der alle nedarver `Tree`. Hver klasse repræsenterer en knudetype i parsertræet.

Symboltabellen opbygges med `SymbolTable` klassens `add()`. Når parsertræet er opbygget, indeholder `SymbolTable` data, som kan anvendes med metoderne `classSize(), fncSize(), ...`.

Kodegeneratoren opbygger fem `AsmBlock`, ved hjælp af `AsmBlock`'s `add()`. Dette foregår under traversering af parsertræet (via dennes accesormetoder), samt opslag i symboltabellen (via dennes accesormetoder `classSize(), fncSize(), varIndex()` `og varType()`).

Koden der starter compileren findes i appendix A side 96



Figur 2.5: *Oversigt over klasserne hvor de hyppigst anvendte metoder er inkluderet. De stiplede pile angiver, at* `Token` *er en protokol-klassen mellem* `Lexer` *og* `Parser`*. De prikkede linier angiver, at 15 klasser nedarver den så* `Tree` *i virkeligheden består af ialt 16 klasser.*

— Kapitel

3 Leksikalsk analyse

Dette kapitel beskæftiger sig med implementationsovervejelser for klassen `Lexer`, samt de dele af programmet, der naturligt hører herunder.

3.1 Implementationsovervejelser

3.1.1 Lexer klassen

Et krav til lexeren er, at slutproduktet er identisk med det JDK producerer, ellers vil Qjava compileren kunne generere kode, der fungerede utilsigtet. Her menes ikke hvorvidt eksempelvis negative tal læses `- tal` eller `-tal`, men om f.eks. sekvensen `+++` læses `++ +`, `+ ++` eller `+++`.

Til indlæsning af input stod valget imellem enten en statemachine opbygget ved hjælp af en DFA eller ved anvendelse af Javas `StreamTokenizer`. Javas `StreamTokenizer` var utrolig letanvendelig. Den kunne bl.a. automatisk genkende kommentarer, strenge, tal og operatorer. Valget faldt derfor på `StreamTokenizer` klassen. `StreamTokenizer` havde dog en alvorlig mangel; den læste tegnsekvensen "`<=`" som to tokens, nemlig "`<`" , "`=`" og tegnsekvensen "`A.f`" som "`A`" , "`.`" , "`f`". Samtidig skulle der wrappes funktionalitet omkring, som kunne genkende tokens, der ikke blev understøttet af Qjava compileren (f.eks. "`switch`"), og melde fejl, hvis den mødte ulovlige tegn såsom "@". Løsningen blev klassen `Lexer`, der arvede `StreamTokenizer`'s funktionalitet, hvor tokengenkendelsen blev udbygget, og hvor advarsler og fejlmeldinger blev implementeret.

3.1.2 Token klassen

Kommunikationen mellem klasserne `Lexer` og `Parser` sker igennem klassen `Token`, hvor hver instans kan indeholde information om præcis et token. Klassen indeholder flere felter, da tokens kan være tekststrenge, tal og tegn. Af hensyn til

brugeren, lagres også linien, hvorpå det enkelte token befinder sig i sourcekoden, således hvis senere faser detekterer uregelmæssigheder eller fejl, kan angive hvor problemet er opstået. Klassen er implementeres som følgende

```
class Token
{
    int id;
    int nval;
    String sval;
    int lineno;
}
```

`id` feltet indeholder, en for hver token, unik værdi, og angiver dermed hvilken type, det pågældende token er. Er token'et en `VAL_INT`, indeholder `nval`, tal-værdien. På samme vis for strenge og `sval`.

Da der hverken er konstruktør- eller accesor- metoder i klassen, er det op til klasserne `Lexer` og `Parser` at læse og skrive de rigtige felter. Man kunne argumentere, at denne implementation ikke følger kriterierne for implementering kapitel 1.1.2 side 2, omvendt skal man passe på med blindt at anvende "det store objektorienterede apparat" i alle tilfælde. `Token` klassens design blev valgt udfra, at den skulle være enkel og fleksibel, og dens størrelse og funktion, retfærdiggør ovenstående implementation.

3.1.3 TokenNames interface

`TokenNames` er en konstruktion der gør det muligt at skelne et token fra et andet. Til implementeringen stod valget mellem tre løsninger, der alle diskuteres i det følgende

Løsningsmodel 1

Udfra en abstrakt klasse `Token` nedarves én klasse for type token, der hver især indeholder de relevante informationer. På den måde sikres det, at alle klasserne er af typen `Token` og at hver token ikke indeholder mere eller mindre information end nødvendigt. Løsningen knytter identitet til hver token, i kraft af den type det pågældende token har. Et implementationsskitse er:

```
abstract class Token{ }
class VAL_INT extends Token{int val; }
class VAL_String extends Token{String val;}
class PLUS extends Token{}
```

Modellen er meget tung og skal udbygges med en ny klasse for hver ny token Qjava skal indeholde, som for det lille sprog Qjava allerede er, inkluderer hele 40 klasser. Javakræver endvidere eksplicit casting der gør programmeringen besværlig, som følgende eksempel, der ikke virker, viser:

Eksempel #3.1:

```
        .
        .
        Token t = lexer.getNextToken();
        f(t);
        .
        .
    public String f(PLUS t){...}
    public String f(VAL_INT t){...}
    public String f(VAL_STRING t){...}
    .
    .
```

Skulle ovenstående eksempel virke, kunne det se ud som følgende:

Eksempel #3.2:

```
        .
        .
        Token t = lexer.getNextToken();
        if(t instanceof PLUS)    f((PLUS) t);
        else
        if(t instanceof VAL_INT) f((VAL_INT) t);
        .
        .
```

Der kræves altså en masse besværlige `instanceof` kald. Ovenstående viser en situation, hvor anvendelsen af nedarvning ikke er en fordel. Kunne metoden `f` flyttes ud i de enkelte token klasser, ville løsningen være langt mere elegant, men ville ske på bekostning af at compilerens faseopdeling nedbrydes.

Løsningsmodel 2

Man kunne argumentere, at sammen med tokens hører deres navne i samme klasse. For at undgå unødig memory forbrug, og for at tilgangen til tokennavnene bliver uniform placeres navnene i et `static` scope. Tokenklassen vil tage form som:

Eksempel #3.3:

```
class Token
{
    int id;
    int nval;
    String sval;
    int lineno;

    static int PLUS       = 1;
    static int VAL_INT    = 2;
    static int VAL_STRING = 3;
    .
    .
}
```

Tokennavnene kan dermed tilgås via eksempelvis "`Token.PLUS`". Det bemærkes, at der refereres direkte ind i klassen uden brug af pointere. Metoden kunne altså naturligt anvendes i f.eks. parser delen, hvor det ikke ville være naturligt at allokere tokens.

Løsningsmodel 3

Den sidste løsningsmodel tager udgangspunkt i Javas interface egenskaber. Fordelen ved interfaces er, at de muliggør multibel arv. Ved anvendelse af arv før det er muligt at tilgå tokennavnene, gøres det eksplicit hvilke klasser der gør hvad. Dette er modsat løsningsmodel 2, hvor tokennavnene kan tilgås uden videre. Modellen ser således ud:

Eksempel #3.4:

```
public interface TokenNames
{
    int
    .
    .
    PLUS         = 50,
    .
    .
    VAL_INT      = 172,
    VAL_STRING   = 173,
}
```

Ved at anvende (konstante) variable, knyttes der et tokennavn til en unik værdi. Der henvises iøvrigt til den fulde sourcekode i appendix E side 106.

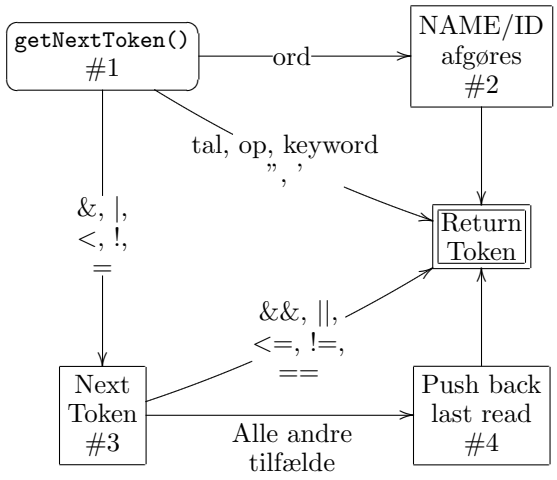
3.1.4 Opsummering

Klassen `Lexer` anvendes til læsning af input, der sendes videre igennem klassen `Token`. Til identifikation af tokens anvendes løsningsmodel 3, hvor hver token defineres udfra værdien i `id` feltet i `Token` klassen, der er fastsat i interfacet `TokenNames`.

3.2 Implementation

3.2.1 metode `getNextToken`

Da den udvidede funktionalitet inkluderede ændring i retur-værdien fra `int` til `Token`, kunne `StreamTokenizer`'s `nextToken()` metode ikke overskrives. Metodens navn blev derfor til `getNextToken()`. Klassens opførsel er skitseret nedenfor samt på figur 3.1 side 22.



Figur 3.1: *Figur der viser* `getNextToken`*'s funktionalitet.*

#1 Næste token læses (t), er t et ord hoppes til (#2).
Er t en operator af typen `&, |, <, !` eller `=` hoppes til (#3).
Er t et ikke negativt tal, en operator, et keyword (alle Javas reserverede keywords, med undtagelse af `char, int, String`), en tekststreng (omgivet af gåseøjne) eller et bogstav (omgivet af appostroffer), returneres t som et `Token`.

#2 I denne fase afgøres, hvorvidt et token er af typen `name` eller `id`, hvor name er defineret som: ⟨name⟩ → ⟨id⟩ "." ⟨id⟩ { "." ⟨id⟩ } skrevet på EBNF. ⟨id⟩ er defineret som en tegnsekvens der er forskellig fra Javas reserverede keywords med undtagelse af `char, int, String`. Dette gøres ved at læse et og eventuelt flere tokens. Efter undersøgelsen returneres resultatet som en `Token`.

#3 Næste token indlæses: er tokenet en operator af typen `&, |, <, !` eller `=` returneres et `Token` ellers hoppes til #4.

#4 Det sidstlæste token skubbes tilbage i bufferen hvorfra der læses, hvorefter det førstlæste token returneres.

> "*The tokenizer for Java is a "greedy tokenizer." It grabs as many characters as it can to build up the next token, not caring if this creates an invalid sequence of tokens. So because* `++` *is longer than* `+`*, the expresion* `j = i+++++i;` *(...) is interpreted as (...)* `j = i++ ++ +i`" [Arnold et al;1997, side 91].

Som det kan læses udfra figuren og/eller ovenstående beskrivelse, er slutresultatet af Qjava compilerens lexer identisk med JDK.

3.2.2 metode `pushBack`

Den anden public metode der findes i `Lexer` er `void pushBack()` metoden der sørger for, at sidste genkendte token "skubbes tilbage", således næste kald af `getNextToken()` vil returnere det sidst genkendte token. Metoden anvendes i situationer, hvor det er nødvendigt for parseren at kende næste token i rækken, for at kunne afgøre i hvilken kontekst det nuværende token står i.

Sourcekoden for `Lexer` findes i appendix C side 98, mens henholdsvis `Token` og `TokenNames` findes i appendix D side 105 og i appendix E side 106.

— Kapitel

4 Qjava specifikation

Oversigt

Sproget compileren skal oversætte defineres i første omgang syntaktisk ved hjælp af en grammatik. Grammatikken udformes udfra hvorledes compileren tilgår syntakskontrollen f.eks. top-down, eller predictive-top-down. Da compileren anvender predictive-top-down, udformes QJavas grammatik på EBNF.

Dette kapitel vil derfor først forklare sprogets opbygning med ord, hvorefter den skrives formaliseret på EBNF. Herefter tilrettes grammatikken til predictive-top-down metodens krav for grammatik. Trods læseren forventes at kende Javas opbygning og syntaks, beskrives dette alligevel med ord, da det reflekterer forfatterens indgangsvinkel og fungerer dermed som argumentation for den færdige grammatik.

4.1 Udformning af QJavas grammatik

Generelt for klassenavne, metodenavne, variabelnavne, pointernavne, samt variabeltyper gælder det, at de hverken må starte med et tal, eller indeholde "`.`" . Førstnævnte af syntaktiske årsager, og sidstnævnte, fordi "`.`" anvendes som separator mellem metoder, klasser og variable og pointere; f.eks. variablen `i` i klassen `A`, skrives `A.i`.

Inden beskrivelsen af grammatikken finder sted, gentages for læseren definitionerne på ⟨id⟩ og ⟨name⟩.

id	→	En tegnsekvens, der er forskellig fra Javas reserverede keywords, med undtagelse af `char, int, String`.
name	→	⟨id⟩ "." ⟨id⟩ { "." ⟨id⟩ }

Herfra forklares og formaliseres den del af Java Qjava dækker "oppe fra og ned", dvs. der startes med hvad et Javaprogram er, og afsluttes med operatorer og variable. Grammatikkens start noteres med S.

S

Et Javaprogram består af en eller flere klassedefinitioner (⟨classdef⟩). Det giver ikke mening at angive noget udenfor dette scope, bortset fra `import, package` — som ikke er indeholdt i Qjava.

S → { ⟨classdef⟩ }

classdef, classcontents

En klassedefinition består af et navn, samt klassens "krop" (⟨classcontents⟩) indkapslet i tuborgparenteser ({...}). For Qjava gælder det, at klassens krop udelukkende består af variabel- og metode- definitioner. Dette er en restriktion i forhold til Java der også tillader klassedefinitioner, `static` blokke og konstruktører. Ekstra ";" før og efter erklæringer tillades men ignoreres under parsningen.

classdef → "class" ⟨id⟩ "{" ⟨classcontents⟩ "}"

classcontents → { (⟨vardef⟩ | ⟨fncdef⟩ | ";") }

vardef

Qjava understøtter udelukkende variabeldefinitioner der opretter en og kun en variabel pr. variabeldefinition, uden værditildeling. Variablens type kan i grammatikken ikke fastlægges på forhånd, da Qjava tillader definering af nye klasser. De eneste faste typer i Qjava er `int`, `char` og `String` som alle i lexeren bliver til ⟨id⟩ tokens. Da variabelnavnet ikke må indeholde ".", er variabelnavne altid ⟨id⟩.

vardef → ⟨id⟩ ⟨id⟩ ";"

fncdef

Funktioner i Qjava kan ikke returnere værdi, hvilket defineres eksplicit med "void". Argumenter til funktioner indkapsles af parenteser, og angives ved først deres type derefter deres navn. Funktionernes "krop" (⟨sentences⟩) indkapsles af tuborgparenteser. Som før angives typerne på argumentvariable samt deres navne med ⟨id⟩.

fncdef → "void" ⟨id⟩ "(" [⟨id⟩ ⟨id⟩ { "," ⟨id⟩ ⟨id⟩ }] ")" "{" ⟨sentences⟩ "}"

sentences

Indholdet af funktioner består af en eller flere sætninger. Sætningerne spænder lige fra "flow control" (`if`, `while`, `break` og `return`), variabel definitioner (⟨vardef⟩), funktionskald (⟨fnccall⟩), til tildelingssætninger (⟨assign⟩). Endvidere kan "`;`" optræde mellem hver sætning.

sentences → { ⟨vardef⟩|⟨fnccall⟩|⟨if⟩|⟨while⟩|⟨break⟩|⟨return⟩|⟨assign⟩|"`;`" }

fnccall

Et funktionskald kan indeholde komma-separerede argumenter i form af udtryk eller variable (da disse genkendes i ⟨E_{10}⟩) indkapslet i parenteser. Da der skelnes mellem tegnsekvenser med og uden "`.`", kan funktionskald både være typen ⟨name⟩ og ⟨id⟩, f.eks. "`f()`" og "`A.f()`".

fnccall → (⟨name⟩ | ⟨id⟩) "`(`" [⟨E⟩ { "," ⟨E⟩ }] "`)`" "`;`"

if

if-sætninger i Qjava kræver en else-blok, der dog kan være tom.

if → "if" "`(`" ⟨E⟩ "`)`" "{" ⟨sentences⟩ "}" "else" "{" [⟨sentences⟩] "}"

while

Den eneste løkkestruktur Qjava understøtter er `while`. while-løkken udføres sålænge udtrykket (⟨E⟩) mellem parenteserne er sandt. Følgeligt skal udtrykket være et udtryk, hvor det er muligt at tale om at det er sandt eller falsk, f.eks. `i < 100` men ikke `i = 2`. while's krop, der kan være tom, indkapsles af tuborgparenteser.

while → "while" "`(`" ⟨E⟩ "`)`" "{" [⟨sentences⟩] "}"

break

"break" anvendes til at afbryde while-løkker.

break → "break" "`;`"

return

"return" kan ikke returnere værdier, men kan på grund af kompabilitet efterfølges af et tomt parentessæt.

return → "return" ["(" ")"] "`;`"

assign

Tildelinger kan enten være variable der tildeles en værdi, eller pointere der tildeles en adresse til et objekt. Variabelnavnet kan enten være lokalt defineret, eller være

defineret i en anden klasse, hvorfor både ⟨name⟩ og ⟨id⟩ tillades som variabelnavn.

assign → (⟨name⟩ | ⟨id⟩) = ⟨E⟩ “;”

E

Det eneste der nu står os uklart er ⟨E⟩. ⟨E⟩ står for et expression, der defineres således:

E → ⟨number⟩ | ⟨id⟩ | ⟨name⟩ | ⟨E⟩ + ⟨E⟩ | ⟨E⟩ - ⟨E⟩ | ⟨E⟩ * ⟨E⟩ | ⟨E⟩ / ⟨E⟩ | ⟨E⟩ % ⟨E⟩ | (⟨E⟩) | - ⟨E⟩ | ⟨E⟩ “|” ⟨E⟩ | ⟨E⟩ & ⟨E⟩ | ⟨E⟩ && ⟨E⟩ | ⟨E⟩ ”||”⟨E⟩ | ⟨E⟩ `==` ⟨E⟩ | ⟨E⟩ `!=` ⟨E⟩ | ⟨E⟩ `<=` ⟨E⟩ | ⟨E⟩ `<` ⟨E⟩ | `!` ⟨E⟩ | “new” ⟨id⟩ “`()`”

Man kunne undres over at finde `new` i ⟨E⟩ istedet for at være defineret i ⟨sentences⟩ som en “allokerings-sætning”. Dette ville blot komplicere genkendelsen af ⟨assign⟩ og ⟨vardef⟩ , og ville umuliggøre anvendelsen af eksempelvis `new A()` som argument i et funktionskald.

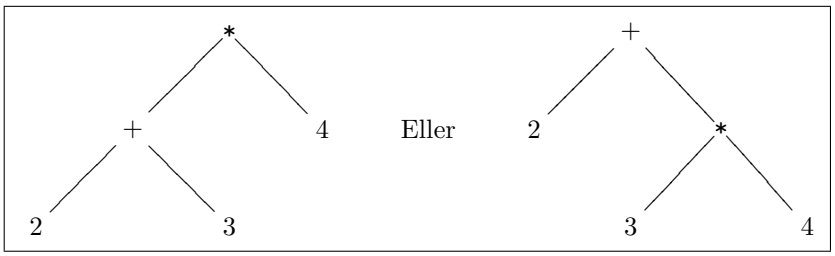
For overblikket skyld ses her den samlede grammatik.

S	→	{ ⟨classdef⟩ }
classdef	→	“class” ⟨id⟩ “{” ⟨classcontents⟩ “}”
classcontents	→	{ (⟨vardef⟩ | ⟨fncdef⟩ | “;”) }
vardef	→	⟨id⟩ ⟨id⟩ “;”
fncdef	→	“void” ⟨id⟩ “(” [⟨id⟩ ⟨id⟩ { “,” ⟨id⟩ ⟨id⟩ }] “)” “{” ⟨sentences⟩ “}”
sentences	→	{ ⟨vardef⟩ | ⟨fnccall⟩|⟨if⟩|⟨while⟩|⟨break⟩ | ⟨return⟩ | ⟨assign⟩ | “;” }
fnccall	→	(⟨name⟩ | ⟨id⟩) “(” [⟨E⟩ { “,” ⟨E⟩ }] “)” “;”
if	→	“if” “(” ⟨E⟩ “)” “{” ⟨sentences⟩ “}” “else” “{” [⟨sentences⟩] “}”
while	→	“while” “(” ⟨E⟩ “)” “{” [⟨sentences⟩] “}”
break	→	“break” “;”
return	→	“return” [“(” “)”] “;”
assign	→	(⟨name⟩ | ⟨id⟩) = ⟨E⟩ “;”
E	→	⟨number⟩ | ⟨id⟩ | ⟨name⟩ | ⟨E⟩ + ⟨E⟩ | ⟨E⟩ - ⟨E⟩ | ⟨E⟩ * ⟨E⟩ | ⟨E⟩ / ⟨E⟩ | ⟨E⟩ % ⟨E⟩ | (⟨E⟩) | - ⟨E⟩ | ⟨E⟩ “|” ⟨E⟩ | ⟨E⟩ & ⟨E⟩ | ⟨E⟩ && ⟨E⟩ | ⟨E⟩ ”||”⟨E⟩ | ⟨E⟩ `==` ⟨E⟩ | ⟨E⟩ `!=` ⟨E⟩ | ⟨E⟩ `<=` ⟨E⟩ | ⟨E⟩ `<` ⟨E⟩ | `!` ⟨E⟩ | “new” ⟨id⟩ “`()`”

4.2 Flertydighed

Opskrives udtrykket `2+3*4` i et parsertræ kan det ifølge gramatikken antage form som skitseret på figur 4.1 side 28, der betyder henholdsvis `(2+3)*4` (figur 4.1 a) og `2+(3*4)` (figur 4.1 b).

I den virkelige verden er dette ikke noget problem, da udtrykket altid læses som `2+(3*4)`, man siger, at “`*` binder stærkere end `+`”. Dette skyldes udelukkende, at



Figur 4.1: *(a) og (b) — To forskellige parsertræer, for udtrykket* `2+3*4`.

operatorene ved konvention er blevet tildelt forskellige prioriteter eller bindinger. Samtidig er det vedtaget, at binære operatore er associerer til venstre, dvs `a=b=c` er ækvivalent med `a=(b=c)`.

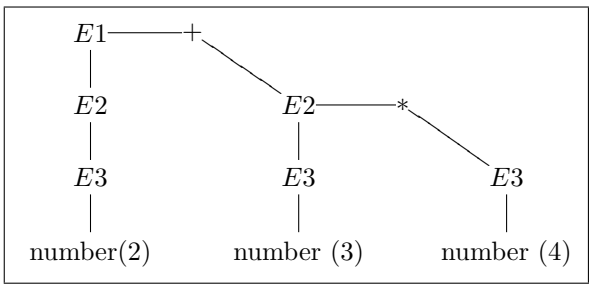
Grammatikken skal altså modificeres, da den er flertydig, og dermed ikke kan repræsentere operatorenes prioritet. For at vise principperne bag en sådan modificering, tages der udgangspunkt i en delmængden "`+ - * /` ⟨`number`⟩" af ⟨E⟩. Prioriteten disse operatorer er givet ved:

1. prioritet: number 2. prioritet: `* /`
3. prioritet: `+ -`

Dette fører til en opsplitning af ⟨E⟩:

$$
\begin{array}{lcl}
\mathrm{E}_1 & \rightarrow & \langle \mathrm{E}_2 \rangle \ \{ \ (+|-) \ \langle \mathrm{E}_2 \rangle \ \} \\
\mathrm{E}_2 & \rightarrow & \langle \mathrm{E}_3 \rangle \ \{ \ (*|/) \ \langle \mathrm{E}_3 \rangle \ \} \\
\mathrm{E}_3 & \rightarrow & \text{number}
\end{array}
$$

Man kan overbevise sig selv om korrektheden af grammatikken ved at opskrive et parsertræ (man kunne jo mene, at $\langle \mathrm{E}_1 \rangle$ og $\langle \mathrm{E}_2 \rangle$ skulle ombyttes). På figur 4.2 side 28 vises udtrykket `2+(3*4)`. Kombinationen af figur samt den nye grammatik giver forhåbenligt læseren det indtryk, at hvis udtrykket `2+(3*4)` skal udregnes, skal multiplikationen udføres før additionen.



Figur 4.2: *Parsertræet for* `2+(3*4)`.

For Qjava kan være kompatibels med Java, må sprogene have identiske prioriter for operatorene. tabel 4.1 side 29 lånt fra [Arnold et al;1997, side 102–103], viser

operatorenes indbyrdes prioriteter, hvor kun de operatorer der eksisterer i Qjava er inkluderet. Prioriteten er givet ved operatorens placering i tabellen, hvor den øverste række har højest prioritet, og den nederste række lavest prioritet.

Beskrivelse	Operator
Postfix operatorer	`. ( )`
Postfix op.	`! - (monadisk)`
Creation	`new`
Multiplicative	`* / %`
Additive	`+ -`
Relational	`< <=`
Equality	`== !=`
Bitwise AND	`&`
Bitwise OR	`|`
Logical AND	`&&`
Logical OR	`||`
Assignment	`=`

Tabel 4.1: *Operatorer og deres indbyrdes prioritet i Java.*

Parentessættet er ikke operatorrer, men medtages alligevel. Prioriteten de tildeles er afgørende for, hvor deres placering i udtryk er tilladt. Var parenteserne placeret på linie med `+, -` i tabellen, ville udtrykket `3*(4/5)` være syntaktisk ulovligt, mens `2+(3*4)` ville være tilladt. Da parenteser anvendes til enten at omdefinere rækkefølgen hvormed udtryk udregnes, eller til at vise rækkefølgen eksplicit, skal de have højst mulig prioritet, så de kan placeres overalt i udtryk. Af samme grund skal ⟨id⟩ , ⟨name⟩, ⟨number⟩ indføres med høj prioritet.

Modificeres grammatikken således operatorenes relative prioritet er identisk med Javas, bliver resultatet:

$$
\begin{array}{lcl}
\mathrm{E} & \rightarrow & \langle \mathrm{E}_1 \rangle \ \{ \ \text{“||”} \ \langle \mathrm{E}_1 \rangle \ \} \\
\mathrm{E}_1 & \rightarrow & \langle \mathrm{E}_2 \rangle \ \{ \ \text{“\&\&”} \ \langle \mathrm{E}_2 \rangle \ \} \\
\mathrm{E}_2 & \rightarrow & \langle \mathrm{E}_3 \rangle \ \{ \ \text{“|”} \ \langle \mathrm{E}_3 \rangle \ \} \\
\mathrm{E}_3 & \rightarrow & \langle \mathrm{E}_4 \rangle \ \{ \ \text{“\&”} \ \langle \mathrm{E}_4 \rangle \ \} \\
\mathrm{E}_4 & \rightarrow & \langle \mathrm{E}_5 \rangle \ \{ \ (\ \text{“==”} \mid \text{“!=”} \ ) \ \langle \mathrm{E}_5 \rangle \ \} \\
\mathrm{E}_5 & \rightarrow & \langle \mathrm{E}_6 \rangle \ \{ \ (\ \text{“<”} \mid \text{“<=”} \ ) \ \langle \mathrm{E}_6 \rangle \ \} \\
\mathrm{E}_6 & \rightarrow & \langle \mathrm{E}_7 \rangle \ \{ \ (\ \text{“+”} \mid \text{“-”} \ ) \ \langle \mathrm{E}_7 \rangle \ \} \\
\mathrm{E}_7 & \rightarrow & \langle \mathrm{E}_8 \rangle \ \{ \ (\ \text{“*”} \mid \text{“/”} \mid \text{“\%”} \ ) \ \langle \mathrm{E}_8 \rangle \ \} \\
\mathrm{E}_8 & \rightarrow & \langle \mathrm{E}_9 \rangle \mid \text{“new”} \ \langle \text{id} \rangle \ \text{“()”} \\
\mathrm{E}_9 & \rightarrow & [\ \text{“!”} \mid \text{“-”} \ ] \ \langle \mathrm{E}_{10} \rangle \\
\mathrm{E}_{10} & \rightarrow & \langle \mathrm{E}_{11} \rangle \mid (\ \langle \text{name} \rangle \mid \langle \text{id} \rangle \ ) \ [\ \text{“(”} \ \langle \mathrm{E} \rangle \ \{ \ \text{“,”} \ \langle \mathrm{E} \rangle \ \} \ \text{“)”} \ ] \\
\mathrm{E}_{11} & \rightarrow & \langle \text{number} \rangle \mid \text{“(”} \ \langle \mathrm{E} \rangle \ \text{“)”}
\end{array}
$$

Tabel 4.2: ⟨E⟩ opsplittet således operatorenes indbyrdes prioriteter kan repræsenteres i parsertræet.

Det ses i tabellen, er operatoren `=` ikke er medtaget, da der ikke ønskes udtryk som `while(i = 2)`, og fordi de steder hvor `=` bør kunne anvendes, kan ⟨assign⟩ anvendes.

Opsplitning i $\langle E_{10} \rangle$ og $\langle E_{11} \rangle$ sker udfra filosofien, at funktionskald og variable er noget "eksternt" der skal hentes "udefra", hvorimod tal er "interne" størrelser. Opsplitningen har ingen syntaktiske konsekvenser.

4.3 Venstrerekursion

Da teknikken predictive-top-down parsing anvendes til at opbygge et parse-træ, er det vigtigt at grammatikken ikke er venstre-rekursiv. Venstre-rekursiv vil sige, at det første i en produktion på højresiden af pilen er en non-terminal identisk med non-terminalen på venstresiden af pilen. Eksisterer en sådan konstruktion vil en anvendelse af grammtikken føre til en uendelig regres ved parsning af input der indeholder den pågældende non-terminal.

Grammatikken var for ⟨E⟩ venstre-rekursiv, før introduktionen af operatorenes prioritet. Men som det ses af tabel 4.2 side 29, er den det ikke længere.

4.4 Første sæt

Den sidste undersøgelse grammatikken underlægges før den skal anvendes, er hvorvidt den muliggør en let genkendelse af non-terminalerne. Det første der står på højre side af pilen må ikke være identisk for tokens der syntaktisk kan stå på samme sted i en tokensekvens — med andre ord, må grammatikkens første sæt (eng: first set) ikke være overlappende. Sker dette, kan rekursiv nedstigningteknikken ikke anvendes, da det ikke er muligt at afgøre hvilken nedstigningen, der skal foretages før flere tokens er indlæst.

Projektet vil ikke præsentere en formel første sæt analyse, men nøjes med grundigt at se på grammatikken. Det ses, at følgende dele af grammatikken er problematiske:

EKSEMPEL #4.1:

sentences	→	{ ⟨vardef⟩ | ⟨fnccall⟩ | ... | ⟨assign⟩ }
vardef	→	⟨id⟩ ⟨id⟩ ";"
fnccall	→	⟨id⟩ "(" [⟨id⟩ { "," ⟨id⟩ }] ")" ";"
assign	→	⟨id⟩ = ⟨E⟩ ";"

Hvis parseren beginder sig i en `sentences` og det følgende token er ⟨id⟩ hvilken nedstigning skal da finde sted? Dette er umuligt at afgøre, før endnu et token er indlæst. `Lexer` klassen faciliterer en `pushBack()` metode til netop dette formål. Ønsker man ikke at anvende "se næste tegn og bestem dig" teknikken kan grammatikken omskrives til:

EKSEMPEL #4.2:

sentence	→	{ ⟨sentence$_2$⟩ | ⟨if⟩ | ⟨while⟩ }
sentence$_2$	→	⟨id⟩ (⟨vardef⟩ | ⟨fnccall⟩ | ⟨assign⟩)
vardef	→	⟨id⟩ ";"
fnccall	→	"(" [⟨id⟩ { "," ⟨id⟩ }] ")" ";"
assign	→	= ⟨E⟩ ";"

Den færdige grammatik for Qjava, som klasserne `Parser`, `Tree` (samt alle dens underklasser) og `CodeGenerator` bygger på ses i tabel tabel 4.3 side 32.

Klassen `Tree` med underklasser findes i appendix G side 120.

4.5 Afrunding

Det har været interessant at opbygge en grammatik for et programmeringsprog fra bunden, da man på måde forstår mekanismerne i sproget. Endvidere får man en forståelse af, at forskellige konstruktioner af grammatikken giver forskellige muligheder, men også besværligheder ved implementationen. Grammatikkens gennemgang kan virke banal, men man skal være varsom med at lade sig snyde af den kompakte form hvormed den er opskrevet, da selvom det ser let ud, har været et tidskrævende arbejde. Under udarbejdelsen af grammatikken har der ikke været hentet "inspiration" fra andre kilder.

Opsplitningen i ⟨id⟩ og ⟨name⟩ har været foretaget for dels at kunne begrænse ⟨vardef⟩ samt argumenterne i ⟨fncdef⟩, da det vil være meningsløst at anvende ⟨name⟩, da Qjava ikke understøtter `static` variable. Dels gøres kodegenereringen lettere, da ⟨id⟩ repræsenterer lokale variable, mens ⟨name⟩ repræsenterer variable/metoder i andre objekter.

Dette har dog også være grobund for diskusioner om hvorvidt et ⟨id⟩ burde udskiftes med ⟨name⟩, eller omvendt, eller om begge typer tokens måtte tillades på det pågældende sted.

I forhold til Java afviger Qjava helt sikkert med ⟨id⟩/⟨name⟩ differentieringen, og en sådan burde nok også fjernes, hvis den semantiske kontrol blev implementeret, da denne dels kunne tage sig af det semantiske (at der ikke ønskes en ⟨name⟩ i ⟨vardef⟩ f.eks.) men også kunne markere om navnet indeholdt ".", således kodegeneratoren lettere kunne generere kode.

Den anden umiddelbare forskel i forhold til Java er, at operatoren = i Qjava er en sætning, hvor den i Java er et udtryk. Dette ses udfra tabel 4.1 side 29, hvor = har en prioritet modsat grammatikken i tabel 4.3.

S	→	{ ⟨classdef⟩ }
classdef	→	"class" ⟨id⟩ "{" ⟨classcontents⟩ "}"
classcontents	→	{ (⟨vardef⟩ | ⟨fncdef⟩ | ";") }
vardef	→	⟨id⟩ ⟨id⟩ ";"
fncdef	→	"void" ⟨id⟩ "(" [⟨id⟩ ⟨id⟩ { "," ⟨id⟩ ⟨id⟩ }] ")" "{" ⟨sentences⟩ "}"
sentences	→	{ ⟨vardef⟩ | ⟨fnccall⟩|⟨if⟩|⟨while⟩|⟨break⟩ | ⟨return⟩ | ⟨assign⟩ | ";" }
fnccall	→	(⟨name⟩ | ⟨id⟩) "(" [⟨E⟩ { "," ⟨E⟩ }] ")" ";"
if	→	"if" "(" ⟨E⟩ ")" "{" ⟨sentences⟩ "}" "else" "{" [⟨sentences⟩] "}"
while	→	"while" "(" ⟨E⟩ ")" "{" [⟨sentences⟩] "}"
break	→	"break" ";"
return	→	"return" ["(" ")"] ";"
assign	→	(⟨name⟩ | ⟨id⟩) = ⟨E⟩ ";"
E	→	$\langle E_1 \rangle$ { "||" $\langle E_1 \rangle$ }
E_1	→	$\langle E_2 \rangle$ { "&&" $\langle E_2 \rangle$ }
E_2	→	$\langle E_3 \rangle$ { "|" $\langle E_3 \rangle$ }
E_3	→	$\langle E_4 \rangle$ { "&" $\langle E_4 \rangle$ }
E_4	→	$\langle E_5 \rangle$ { ("==" | "!=") $\langle E_5 \rangle$ }
E_5	→	$\langle E_6 \rangle$ { ("<" | "<=") $\langle E_6 \rangle$ }
E_6	→	$\langle E_7 \rangle$ { ("+" | "-") $\langle E_7 \rangle$ }
E_7	→	$\langle E_8 \rangle$ { ("*" | "/" | "%") $\langle E_8 \rangle$ }
E_8	→	$\langle E_9 \rangle$ | "new" ⟨id⟩ "()"
E_9	→	["!" | "-"] $\langle E_{10} \rangle$
E_{10}	→	$\langle E_{11} \rangle$ | (⟨name⟩ | ⟨id⟩) ["(" ⟨E⟩ { "," ⟨E⟩ } ")"]
E_{11}	→	⟨number⟩ | "(" ⟨E⟩ ")"

Tabel 4.3: *Den komplette grammatik for QJava.*

— Kapitel

5 Parser

- Opbygning af parsertræet, sektion 5.1 side 34
- Repræsentation af parsertræet, sektion 5.2 side 37

Oversigt

Under parsningen kontrolleres syntaksen af sourcekoden, mens en repræsentation af grammatikken skabes i form af et parsertræ. Implementationen af parseren er hvad man kunne kalde "EBNF grammatikken skrevet på Java'sk". For hver non-terminal definition (altså for hver højreside) der eksisterer i grammatikken, findes en ækvalent metode med et ækvavilent navn i klassen `Parser`.

Foruden de "grammatiske klasser" findes der to hjælpemetoder `eat(int)`, `errorAndExit(String)`. Den første kontrollerer om det sidstlæste token har samme id, som metodens argument. Er id identisk læses næste token, ellers meldes fejl med `errorAndExit()` metoden.

Princippet bag alle metoderne illustreres flot i metoden, der parser et `if` udtryk.

Eksempel #5.1:

```
private Tree if_()
{
    eat(IF);
    eat(LPAR);
    Tree condCode = E();
    eat(RPAR);

    eat(LBRACE);
    Tree thenCode = sentences();
    eat(RBRACE);

    eat(ELSE);
    eat(LBRACE);
    Tree elseCode = sentences();
    eat(RBRACE);

    return( new If(condCode, thenCode, elseCode, T.lineno) );
}
```

Syntakskontrollen for et if-udtryk, er på grund af sin rekursive form simpel og

overskuelig. Som det ses, er syntakskontrollen af if's kroppe overladt til metoden `sentences()`, der igen forgrener sig ud i `assign(), while(), break(), ...` Det ses endvidere, at metoden `if_()` samt metoderne `E(), sentences()` alle returnerer et træ. Simultant med syntakskontrollen opbygges parsertræet altså.

Parsningen for udtryk følger på samme vis grammatikken. Metoden for parsning af $\langle E_6 \rangle$ ses her

Eksempel #5.2:

```
private Tree E6() throws IOException
{
    Tree t = E7();

    while(T.id == PLUS || T.id == MINUS)
    {
        int op;

        if(T.id == PLUS)
        {
            eat(PLUS); op = PLUS;
        }
        else // -
        {
            eat(MINUS); op = MINUS;
        }
        OpDual tree = new OpDual(op, t, E7(), T.lineno);
        t = tree;
    }
    return(t);
}
```

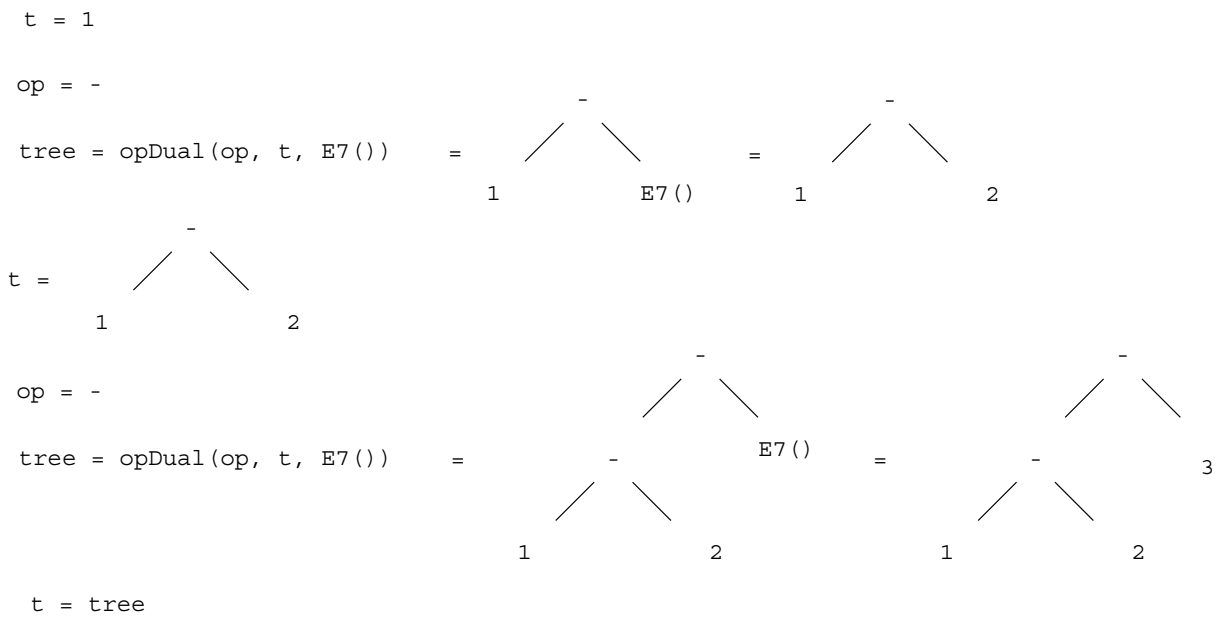
Man skal dog være særlig omhyggelig i konstruktionen af parsertræet, da operatorer med ens prioritet ikke nødvendigvis garanterer, at evalueringsrækkefølgen er uden betydning. Således er f.eks. $2 - (3 - 4) \neq (2 - 3) - 4$. Udtrykket `1-2-3` er ifølge reglen om venstreassociering `(1-2)-3` vil i parsertræet blive dannet som et "minustræ" med 1 og 2 som børn. Dette træ er igen barn i et "minustræ" med 3 som det andet barn. Dette sker ved 1 læses ved `Tree t = E7();`. Dernæst noteres operatoren `-`, og slutteligt oprettes et `opDual`-træ med "`t`" som venstretræ, `-` som operator, og næste del af af udtrykket (som er "2") som højretræ. Dette træ bliver herefter venstre barn i et nyt "minustræ". Oprettelsen af træet for udtrykket `1-2-3` er skitseret på figur 5.1 side 35.

5.1 Implementationsovervejelser

Der vil i denne sektion blive diskuteret tre overordnede implementationer af parsertræet

5.1.1 Løsningsmodel 1

Parsertræet er et stort sammenhængende træ, hvor hver non-terminal implementeres i en separat klasse indeholdende nok felter til en repræsentation af produk-



Figur 5.1: *Opbygningen af parsertræet for udtrykket `1-2-3`.*

tionen af non-terminalen. I ovenstående eksempel, vil `If` klassen indeholde tre referencer til henholdsvis et E-træ og to sentences-træer.

Gentagelser der i grammatikken angives med {} implementeres med en next-pointer. Klassen `classcontents()` vil implementeres således:

Eksempel #5.3:

```
private Tree classcontents()
{
    .
    .
    Tree contents;
    Tree next;
}
```

5.1.2 Løsningsmodel 2

Den anden løsningsmodel implementerer ligesom den første en klasse for hver non-terminal. Men håndteringen med gentageler administreres med Javas `Vector` klasse istedet. Dette gør koden mere overskuelig og lettere at skrive, da håndteringen med `elementAt()` er simplere end at jonglere med next-pointere.

5.1.3 Løsningsmodel 3

Den sidste løsningsmodel implementerer som de to foregående klasser for ⟨S⟩, ⟨classdef⟩, ⟨classcontents⟩ og ⟨fncdef⟩ fra grammatikken som selvstændige klasser.

De sidste non-terminaler (⟨fnccall⟩, ⟨while⟩, ⟨if⟩, ... og ⟨E⟩'erne) repræsenteres ved en generisk klasse, hvilket kraftigt reducerer klassernes antal. Den generiske klasse implementeres som:

Eksempel #5.4:

```
private generic()
{
    String op;
    Tree one;
    Tree two;
    Tree three;
}
```

De forskellige dele af grammatikken kan dermed repræsenteres således:

Op	one	two	threæ
"if"	E	s_1	s_2
"break"	`null`	`null`	`null`
"vardef"	E	E	`null`
"+"	E	E	`null`
"!"	E	`null`	`null`

Løsningsmodellen gør administrationen let, da der dels ikke er specielt mange klasser, dels vil ny funktionalitet oftest kunne tilpasses løsningsmodellen, som i værste tilfælde blot skulle udvides med flere felter. Omvendt mangler løsningsmodellen struktur, og kan nemt fremstå som et "hack". Et lidt overdrevet eksempel er, at når ⟨while⟩ skal defineres, defineres det som `(''while'', s1, E, null)` — altså stik modsat af definitionen af ⟨if⟩. Eksemplet ville fungere, men der vil med stor sandsynlighed opstå fejl i senere faser, fordi felternes betydning let forveksles af programmøren.

5.1.4 Valg af repræsentation

Valget faldt på en kombination af løsningsmodel 2 og 3. Løsningsmodel 2 administrerede {..} fra grammatikken ved brug af `Vector`-klassen, og satte parsertræet i faste rammer igennem de mange klassedefinitioner. Løsningsmodel 3 angav hvorledes antallet af klasser og dermed antallet af forskellige knuder i parsertræet, kunne reduceres. Non-terminalerne E–E_{11} blev derfor implementeret ved hjælp af følgende fire generiske klasser.

Klassenavn	Dækningsområde	non-terminaler
`opMonadic`	Monadiske operatorer	E_9
`opDual`	Duale operatorer	E – E_8
`opCall`	Variable og funktionskald	E_{10}
`opConst`	Konstante udtryk	E_{11}

5.2 Implementering af parsertræet

Parsertræet er realiseret udfra en abstrakt klasse `Tree`, hvis eneste indhold er en variabel indeholdende et linienummer og metoder til fejludskrift og debug. Alle klasserne arver denne klasse, og er således alle af samme type. Fordelen ved denne konstruktion er, at reference variablen eksempelvis: `Tree p`, kan referere til en hvilken som helst klasse (knudetype) i parsertræet.

Linienummeret der lagres, er stedet i sourcekoden den enkelte knude repræsenterer.

Resultatet er 1 superklasse (`Tree`) og 15 nedarvende klasser, der knytter sig tæt til grammatikken, uden at være en konkret afbildning af den. Klasserne er alle realiseret med private felter og public accessor metoder, således, implementationen med `Vector` kan udskiftes med en mere effektiv klasse uden vanskeligheder. Dette sker på bekostning af overskueligheden, da klassedefinitionerne alle er blevet væsenligt større. Dog skal de fleste linier kode tilskrives `toString()` metoderne der anvendes til testformål.

5.2.1 Koden

```
public abstract class Tree
{
    public abstract String toString();

    protected void errorAndExit(String s)
    {System.out.print(s); System.exit(0);}

    protected int lineno; // line # of element in source
    public int lineno(){return(lineno);}
}
.
.
.

class Sentences extends Tree
{
    private int getCounter = 0;
    private Vector sList = new Vector(); // liste of <sentences>

    // constructor
    Sentences(int lineno){super.lineno = lineno;}

    public void add(Object o){sList.addElement(o);}

    public Tree getNext()
    {
        if(getcounter > -1 && getCounter < sList.size())
            return((Tree) sList.elementAt(getCounter++));
        return(null);
    }

    public void pushBack(){getCounter--;}

    public String toString()
```

```
    {
        StringBuffer buf = new StringBuffer();

        for(int i = 0; i < sList.size(); i++)
            buf.append( sList.elementAt(i).toString() + "\n");

        return(buf.toString());
    }
}
```

5.3 Parsertræet

Den interne repræsentation er skitseret på figur 5.2 side 39 for det følgende lille kodeeksempel. På figuren ses øverst klassen `classdef` istedet for `S` som grammatikken foreskrev. `S` er blevet fjernet, og er blevet implementeret i `classdef`. Navnene i parentes evt. efterfulgt af en slash ("/"), angiver det tilladte indhold af en knude, det vil sige at en `defList` knude indeholder enten `varDef` og/eller `fncDef` knuder. Nederst ved opsplitningen af `opDual` ses effekten af $\langle E_{10} \rangle$-$\langle E_{11} \rangle$ opsplitningen, da "`i`" er en `opCall` knude ($\langle E_{10} \rangle$), mens "`3`" er en `opConst` knude ($\langle E_{11} \rangle$). Figuren illustrerer endvidere, at træstrukturen gør tegnene `(, ), {, }, ;` osv overflødige.

Eksempel #5.5:

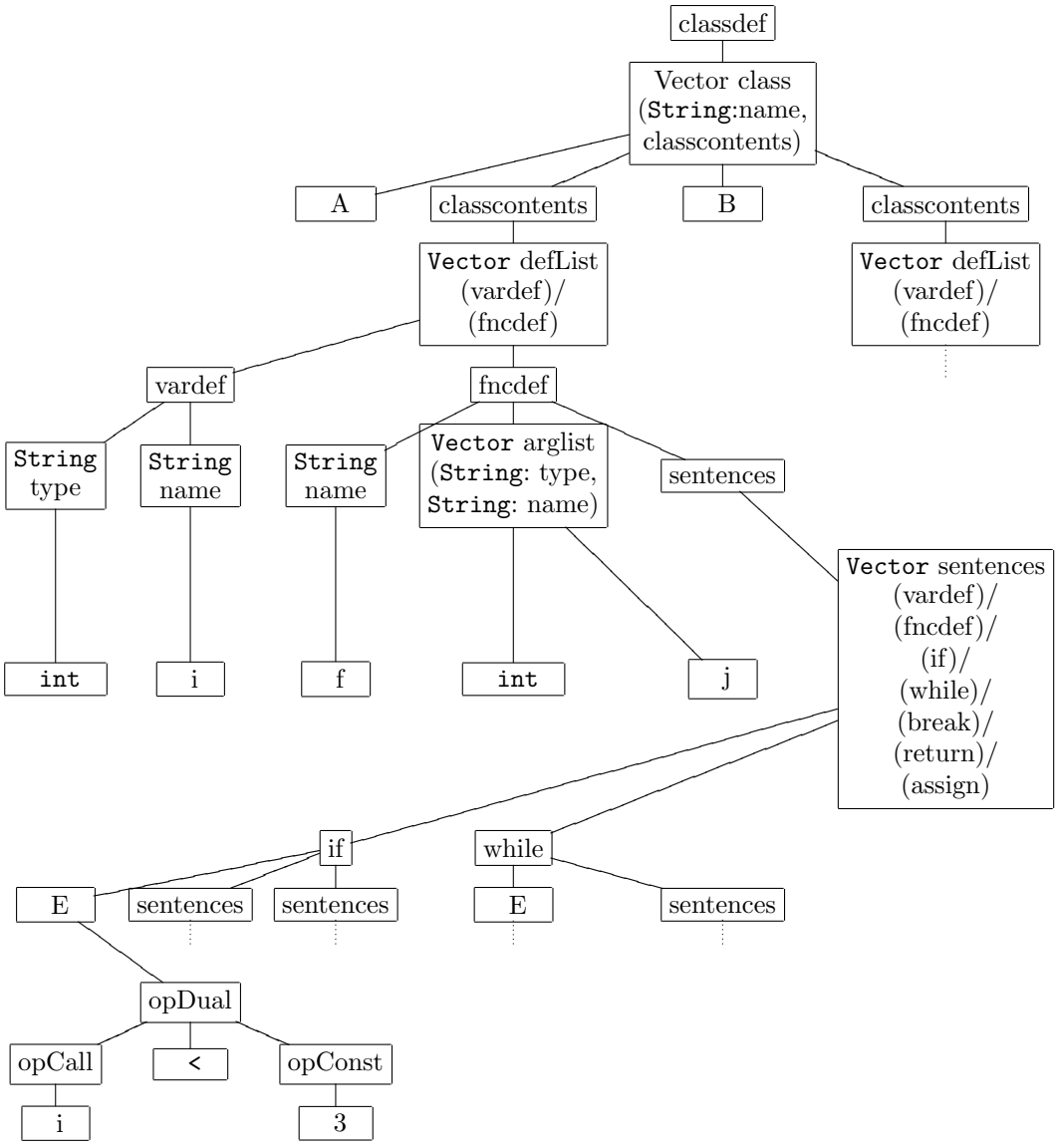
```
class A
{
    int i;

    void f(int j)
    {
        if(i < 3)
        {
        }
        else
        {
        }

        while()
        {
        }
    }
}

class B
{
}
```

Den fulde implementation findes i appendix F side 108.



Figur 5.2: *Grafisk præsentation af et parsertræ.*

— Kapitel

6 Semantisk analyse

6.1 Semantik

Udfra grammtikken i tabel 4.3 side 32, kan følgende program konstrueres.

Eksempel #6.1:

```
class A
{
    void f()
    {
        int i;
        break;
    }

    void g()
    {
        i = 2;
    }
}
```

Programmet er syntaktisk korrekt, men giver ingen mening, hvis man skulle forklare hvad det udførte. Den syntaktiske analyse er altså i sig selv ikke en garanti for et korrekt program. Mange af sådanne problemer kunne sikkert håndteres i grammatikken, som f.eks. at `break` kun må eksisterer indenfor et while-scope. Næste eksempel viser hvorledes dette gøres ved at substituere ⟨sentences⟩ og ⟨while⟩ fra tabel 4.3 side 32.

Eksempel #6.2:

sentences	→	$\{$⟨vardef⟩|⟨fnccall⟩|⟨if⟩|⟨while⟩|⟨return⟩|⟨assign⟩|";"$\}$
whilesentncs	→	$\{$⟨vardef⟩|⟨fnccall⟩|⟨if⟩|⟨while⟩|⟨break⟩|⟨return⟩|⟨assign⟩|";"$\}$
while	→	"while" "(" ⟨E⟩ ")" "{" [⟨whilesentncs⟩] "}"

Grammatikken vil dog på denne måde hurtigt vokse sig uoverskuelig stor og kompleks. Det andet problem i første eksempel er, at variablen `i` sættes til værdien 2 udenfor scopet den er defineret i. Et sådan problem vil ikke kunne løses lige så let,

om overhovedet muligt. Sådanne problematikker bør overlades til den semantiske fase, altså fasen der kontrollerer meningen med sætningerne, fremfor at klare det i den syntaktiske kontrol.

Den semantiske kontrol er i Qjava compileren begrænset til de sideeffekter, der opstår ved anvendelse af `SymbolTable` klassen.

6.2 Klassen SymbolTable

`SymbolTable` klassen opbygges under parsningen. Hvergang en ⟨vardef⟩ mødes, kaldes `SymbolTable`'s følgende metode `add(Variabeltype, Klassenavn, Metodenavn, Variabelnavn)`. `SymbolTable` indeholder information om hvor hvilke variable allokeres i hvilke scope, samt rækkefølgen herfor. Foruden `remove()` indeholder klassen følgende metoder

`typeCheck()` Kontrollerer om en variabel i et givent scope er af en given type.

`varType()` Returnerer typen på en bestemt variabel.

`fncSize()` Returnerer antallet af variable `*` 2 defineret i den angivne metode.

`classSize()` Returnerer antallet af variable `*` 2 defineret i den angivne klasse.

`varIndex()` Returnerer hvornår den angivne variabel blev defineret i det angivne scope.

Heraf anvendes de sidste fire metoder i `CodeGenerator` klassen, hvor den første metode er tiltænkt fremtidig brug. Grunden til Funktionerne returnerer værdier dobbelt så store som de reelt er, skyldes, at alle variable fylder 2 bytes i assembler oversættelsen.

6.3 Semantikkontrol i Qjava compileren

Semantikkontrollen i Qjava, eller rettere de mekanismer der finder sted, hvis sideeffekter ligner semantikkontrol, er af følgende to typer: 1) manglende hoplabel, 2) manglende index nummer.

6.3.1 Manglende hoplabel

Når koden for `while` genereres, skal den label der skal hoppes til hvis `break` mødes, specificeres. Dette er den kun hvis der genereres kode i en while-løkke. Mødet `break` udenfor en while-løkke, er den label der skal hoppes til lig `null`, og der meldes fejl.

6.3.2 Manglende index

Når kodegeneratoren møder `i = 2`, kaldes
`varIndex(Klassenavn, metodensnavn, variabelnavn)`, det kunne f.eks. være `varIndex("A", "g", "i"`, altså hvornår i klassen `A` i metoden `g()` blev variablen `i` erklæret. Er returværdien $== -1$, altså ikke fundet, kan det sluttes, at `i` forsøges anvendt uden for det scope det blev defineret i. Dog skal man forinden denne konklusion forvisse sig, at variablen ikke er et felt (og således anvendes i et korrekt scope). Dette gøres let ved metodekaldet `varIndex("A", null, "i")`. Er resultatet stadig -1, anvendes `i` i et scope hvor `i` ikke er defineret.

6.4 Implementation `SymbolTable`

Symboltabellen er en vektor indeholdende `Symbol`. Hver `Symbol` indeholder de nødvendige felter til at kunne beskrive en variabel fuldt ud. Felterne er variabel type, variabel navn, klassen hvori den er defineret, funktionen hvor den er defineret, og slutteligt hvornår variablen blev erklæret.

Metoderne `classSize(), fncSize()` leder hele vektoren igennem og summerer hhv. alle klassefelter og funktionsvariable. `varIndex()` leder vektoren igennem og returnerer det fundne elements `index` felt eller -1 ved mislykkedes søgning (lignende sker i `varType(), typeCheck()` metoderne, bare med andre felter end `index` feltet). Klassen er konstrueret på en sådan vis, at brute force søgningen let kan erstattes af f.eks. `HashTable` for at opnå hastighedsforøgelse ved `varIndex(), varType() og typeCheck()` metoderne. Da størrelsen af funktioner og klasser ikke kan ændres under eller efter compileringstidspunktet, kunne `classSize(), fncSize()` implementeres ved at gemme de konstante værdier i en `HashTable`. Dette ville dog kræve ekstra kode, der kunne fortælle `SymbolTable` af parsningen var tilendebragt. I det følgende vises et udsnit af `SymbolTable`.

```
\begin{footnotesize}
private Vector list = new Vector(); // contains elements

protected class Symbol
{
    int index;
    String fromClass, fromFnc, name, type;

    // konstruktør
    Symbol(String t, String fC, String fF, String n, int i)
    {type = t; fromClass = fC; fromFnc = fF; name = n; index = i;}
}

\end{footnotesize}
```

Den fulde kode findes i appendix H side 128.

— Kapitel

7 Assembler

Beskrivelse af kodegenerering er opdelt i to kapitler, hvor dette kapitel kort forklarer den anvendte assembler, samt generelt om hvorledes lageradministrationen foregår.

Kapititel 8 side 53 tager sig af forklaringen af kodegenereringen og på hvorledes implementeringen blev foretaget.

Oversigt

7.1 Assemblersproget

Det anvendte assemblersprog er Intel 80286'er kompatibelt, beregnet på at blive udført i Microsoft DOS. Assembler faciliterer en række kommandoer til aritmetik, hop osv., hvor hovedparten Qjava anvender ses i tabel 7.1 side 44. Alle kommandoerne er meget generelle og findes i næsten enhver assembler, hvilket gør oversættelsesfasen let porterbar. Foruden disse kommandoer har 80286 arkitekturen adskillelige registre, hvor kun de der anvendes her fremstilles i tabel 7.1 side 45.

Alle registre er 16 bit, pånær ax–dx, der hver kan opdeles i to 8 bit registre. Registrene tilgås med `al` og `ah`, `bl`, `bh` osv. hvor `l` ("`Low`") tilgår de nederste 8 bit og `h` ("`High`") tilgår de øverste 8 bit. Registrenes navne er forskellige, ligeså er deres funktionalitet. Der skal her ikke forsøges at argumentere for brugen af de enkelte registre.

Flytning af data	
`mov` a, b	Kopierer indholdet af register b til a.
`mov` a, [b]	Kopierer adressen på b til a.
`mov` a, b[x]	Kopierer indholdet af adressen b+x til a.
`push` b	Kopierer b til toppen af stakken.
`pop` a	Kopierer og fjerner toppen af stakken til a.
Aritmetiske kommandoer	
`add` a, b	Adderer b til a og gemmer resultatet i a.
`sub` a, b	Subtraherer b fra a og gemmer resultatet i a.
`cmp` a, b	Sammenligner a med b.
`mul` a, b	Multiplicerer b med a og gemmer resultatet i a.
`div` a	Dividerer a med et register, rest gemmes i a, modulo-værdi gemmes et andet register.
`and` a, b	Udfører en logisk AND på a og b og gemmer resultatet i a.
`or` a, b	Udfører en logisk OR på a og b og gemmer resultatet i a.
`not` a	Udfører en logisk NOT på a og gemmer resultatet i a.
`cmp` a, b	Sammenligner a med b ved subtraktion.
Hop	
`call` c	(ubetinget) Hop til procedure c, og returnerer når c slutter.
`jmp` c	(ubetinget) Hop til c
`je` c	Hop til c hvis lig
`jne` c	Hop til c hvis ikke lig
`jl` c	Hop til c hvis mindre
`jle` c	Hop til c hvis mindre eller lig
`jg` c	Hop til c hvis større

Tabel 7.1: *Oversigt over de mest anvendte assemblerkommandoer i oversættelsen til assembler.*

Registre		Segmenter	
ax	Accumulator	cs	Code Segment
bx	Base Index	ds	Data Segment
cx	Count	ss	Stack
dx	Data		
sp	Stack Pointer		
bp	Base Pointer		
di	Destination Index		
si	Source Index		
(ip)	(Instruction Pointer)		

Figur 7.1: *Oversigt over de anvendte registre og segmenter i Intel 80286 arkitekturen.*

Ethvert program indeholder et .DATA segment og et .CODE segment, samt en stak.

data .DATA segmentet indeholder globale "variable" og konstanter, som f.eks. tekststrenge.

code I .CODE segmentet placeres al koden. Gennemløbet af koden under udførelse sker vha. PC registret (Program counter).

stak Stakken styres af SP registret (Stack Pointer). SP peger altid på det øverste element på stakken. SP registret tæller negativt for hvert element der lægges på stakken, således SP går mod 0.

7.1.1 Syntaks

Det anvendte assemblersprogs syntaktiske grundtræk, kan let specifficeres udfra følgende EBNF grammatik:

S	→	{ ⟨line⟩ }
line	→	(⟨label⟩ | ⟨sentence⟩ | ⟨procedure⟩)
label	→	⟨name⟩ "`:`"
sentence	→	⟨command⟩ ⟨unit⟩ ["`,`" ⟨unit⟩]
procedure	→	⟨name⟩ "`PROC`" { ⟨line⟩ } "`ret`" ⟨name⟩ "`ENDP`"
command	→	("`add`" | "`push`" | "`pop`" | "`jmp`" | "`call`" | ...)
unit	→	⟨string⟩ | ⟨number⟩

Hvor der for ⟨name⟩ gælder samme restriktioner som variabelnavne i Java. I en ⟨procedure⟩, må en ny procedure ikke defineres mens de to ⟨name⟩ skal være identiske. Ved ⟨sentence⟩ med to units, angiver den første ⟨unit⟩ **destination**, mens anden ⟨unit⟩ angiver **source**. "læg 2 til register AX" skrives som `add ax, 2`. Kommentarer angives med "`;`", hvis funktion er identisk med Javas "`//`".

⟨command⟩ var med vilje ikke angivet fuldstændigt i grammatikken, da kommandoerne og deres funktion findes i tabel 7.1 side 44.

7.1.2 Hop i assembler

En vigtig del af assembler, er dens mulighed for at *branche* eller bryde et ellers lineært afviklingsforløb af koden. I assembler findes der to måder at hoppe på, hop til en procedure, og hop til en label. Ved hop til procedurer, anvendes kommandoen `call`, mens kommandoerne angivet i tabel 7.1 side 44 anvendes til hop til labels.

Ved hop til procedurer push'es adressen på PC registret, altså hvorfra hoppet blev udført. Når assembleren møder kommandoen `ret`, pop'es det øverste element fra stakken i PC registret, hvilket resulterer i et automatisk hop til denne adresse. Udfører man i en procedure ligeså mange `pop` som `push`, returneres der således tilbage til stedet i programmet hvorfra `call` blev kaldt. Dette minder om metodekald i Qjava, procedurer fasciliterer dog ikke argumenter til `call` som Qjava gør til metodekald.

Den anden type hop er til en label et andet sted i sourcekoden. Denne type hop kan sammenlignes med `goto` i sprog som C/C++ og minder om "`break labelnavn:`" i Java. Denne type kald anvendes hovedsageligt til løkke konstruktioner.

7.2 Overordnet lageradministration

I det følgende beskrives lager administrationen ultra kort, for så at blive uddybet i de respektive undersektioner.

I CODE segmentet placeres funktioner og standardfunktioner.

I DATA segmentet placeres hoben. Da Qjava ikke implementerer `static` variable placeres yderligere kun simple variable til intern administration. Hoben er en stor blok hukommelse, som anvendes til repræsentation af objekter.

Stakken anvendes til lokale variable samt argumenter til funktioner.

Overordnet set skal følgende fire typer ting administreres: felter i objekter, lokale variable, parametre til funktioner og mellemresultater. For alle tingene gælder, at de på samme tidspunkt kan eksistere i mange instanser. Adresseringen af disse sker derfor relativt via registrene `bx, bp` og `sp`, da assemblers globale variable ikke kan anvendes.

For instanserne gælder altid at de eksisterer i form af instanser i objekter, eller i funktioner. Objekter placeres i hoben istedet for på stakken, da de skal eksistere efter funktionskald, som normalt tømmer stakken når funktionen forlades. Placeredes objekterne på stakken ville rekursive funktionkald (og dynamisk garbage collection hvis dette en dag skulle implementeres) være meget besværlige at implementere.

Bemærk, at mens stakken tæller nedaf, tæller hoben opad. Det be-

tyder, at det øverste element på stakken tegnes nederst, og omvendt øverst når hoben afbilledes.

7.2.1 Klasser

Klasser i Qjava består af en række felter, samt en række funktionaliteter (metoder). De to størrelser hænger dog ikke så tæt sammen, som man umiddelbart skulle tro. Oprettes 1000 instanser af klassen `X`, oprettes kun felterne i `X`. Da metoderne ikke kan ændres fra instans til instans, er der ingen grund til at repræsentere dem 1000 gange. Metoders lokale variable kan ikke tilgås fra referencer til klasser, hvorfor disse ikke oprettes ved oprettelse af klasseinstanser. Klassens metoder genereres kun én gang, og placeres et sted i den genererede kode i form af procedurer.

En klasses størrelse i hoben, er derfor antallet af felter i objektet multipliceret med felternes størrelse. Nedenstående klasse `A` fylder altså $2 \cdot 16$ bit $= 4$ bytes.

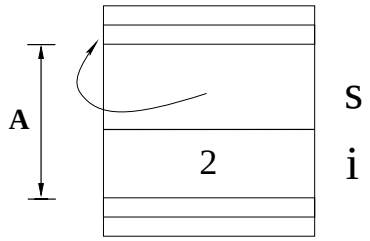
Eksempel #7.1:

```
class A
{
    int i;
    String s;
}
.
.
    A klasse;
    klasse = new A();
```

I de 4 bytes, placerede `i` først i hoben, da `i` defineres først og således har et lavere index nummer. Da alle variable er 16 bit optager `i` derfor de første 2 bytes, mens `s` optager de to sidste. Udføres følgende kode:

```
klasse.i = 2;
klasse.s = "hej ib";
```

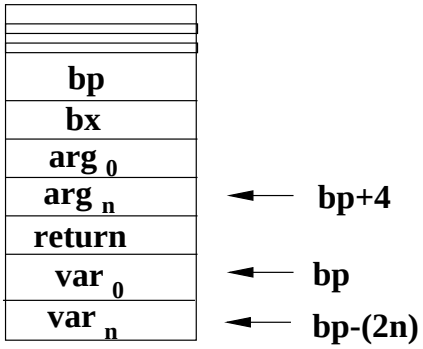
tildeles `i` og `s` nye værdier, som er henholdsvis 2 og adressen på et `String`-objekt med indholdet `hej ib`. Situationen efter tildelingen, er skitseret på figur 7.2 side 47, som forestiller et udsnit af hoben.



Figur 7.2: *Skitse af hoben efter tildeling af* `A`*'s felter.*

7.2.2 Variable

Variable er enten variable defineret i metoder, eller er argumenter til metoder. Variablerne skabes hvergang metoden kaldes, og dør, når metoden forlades. For begge typer variable gælder det, at en sådan opførsel elegant kan implementeres ved anvendelse af assemblers stak. Da et objekts felter gemmes i hoben istedet for på stakken, er der ingen problemer med rekursive kald. For hvert rekursive kald, ligesom for alle andre metodekald, lagres argumenterne samt lokale variable først på stakken inden metodens indhold udføres. Der er derfor ingen forskel på metoder hvad entent de er rekursive i deres natur eller ej.



Figur 7.3: *Stakken som den tager sig ud lige efter et funktionskald.*

Eksempel #7.2:

```
class B
{
    int b;

    void f(int x, int y)
    {
        int len;
        f(x, y);
    }
}
```

Stakkens udformning efter første kald af `B.f()` ses på figur 7.3 side 48. Registret `bp` peger altid på den første lokale variabel i metoden (på stakken). Og da hver variabel fylder 16 bit, kan variablene tilgås med `[bp-index]`, hvor `index` er variablens indexnummer. Indexnummeret tildeles variablene efter hvornår de defineres, således er den første varibels index nummer = 0, men den næste er $2, 4, ...$ Argumenterne til metoden kan ligeledes tilgås via `bp` registret, nemlig ved `[bp+4+index]`. De ekstra 4 adressepladser skyldes, at der skal hoppes forbi retur adressen.

Ved brug af `bp` pointeren, er tilgangen til variable uafhængigt af hvor lidt eller hvor meget stakken anvendes.

7.2.3 Felter

Variable i klasse scope blev læst og skrevet fra hoben. Styringen af hvor i hoben der læses og skrives styres af `bx` registret, der hele tiden indeholder 'this' adressen for det aktuelle objekt. `bx` virker altså som `this` referencen gør i Java. Felter kan tilgås på to måder, i det aktuelle objekt, eller fra andre objekter. For begge tilfælde tilgås variable med kaldet `hob[bx+index]`, hvor `index` er variablens index nummer. For felter i andre objekter skal man dog først sætte `bx` til det andet objekts 'this' adresse. Dette gøres let, da det blot er referencevariablens værdi. Altså for kaldet `a.i` udføres:

Eksempel #7.3:

gem `bx`;
bx = a;
`i` kan læses/skrives med `hob[bx+index]`
hent `bx`;

7.2.4 Mellemresultater

Da mellemresultater er temporære, og kun er anvendelige på de konkrete steder, anbringes de på stakken, hvor de automatisk fjernes (med `pop`), når de anvendes.

7.2.5 Oprettelse af hoben

For simpelthedens skyld, er hobens størrelse statisk og givet ved programmets start. Alle programmer der genereres indeholder således følgende blok kode, der allokerer hoben ved programstart. Koden er placeres i DATA-segmentet.

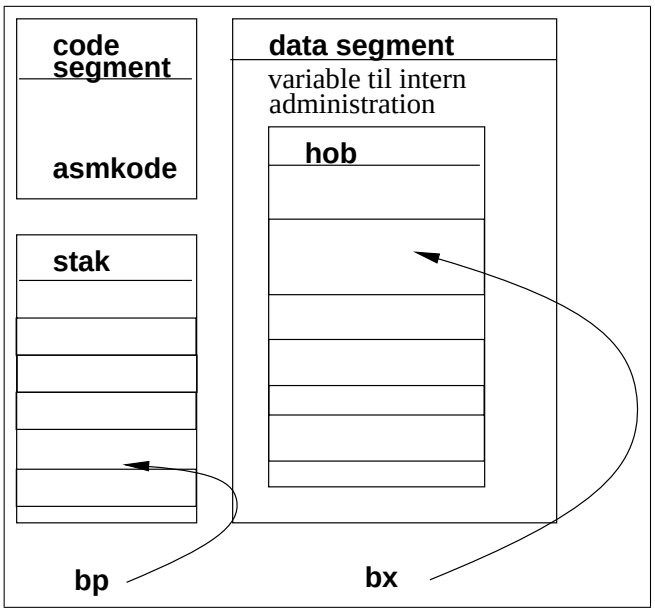
```
hobptr  dw 0
hob     dw <HOBSIZE> dup(0)
```

Hvor `<HOBSIZE>` er en konstant indsat af compileren, der angiver hobens størrelse i antal word. Foruden hoben, allokeres også en `hobptr`, en pointer, der angiver hvor meget af hoben der er i anvendelse, og som modificeres ved allokeringer. Da compileren ikke indeholder garbage collection, opbruges hoben med tiden, da "døde" objekter, altså objekter som ingen referencer har, ikke fjernes.

7.2.6 Opsummering

Opsumeret på figur 7.4 side 50 er følgende gennemgået

- Code segmentet indeholder den genererede assemblerkoden.
- Data segmentet indeholder hoben, der rummer instanser af objekter samt interne variable til intern administration f.eks. hvor meget af hoben der er blevet anvendt.



Figur 7.4: *Oversigt over lageradministrationen i Qjava.*

- Stakken anvendes til opbevaring af lokale variable samt argumenter til funktioner, som tilgås `[bp-index]` og `[bp+4+index]`.
- Felter tilgås med `hob[bx+index]`.

Der er altså blevet forklaret, *hvordan* hob, stak osv. tager sig ud. I de følgende sektioner, forklares *hvorledes* henholdsvis stak, hob osv. får dette indhold.

7.3 Referencer & Objekter

Objekter "allokeres" med ved hjælp af proceduren `new`. Når `new` kaldes, reserveres et angivet antal bytes af hoben i en blok. Startadressen på blokken returneres, mens `hobptr` tælles op med størrelsen af den reserverede blok.

Referencer til objekter har som alle andre variable en størrelse på 16 bit. Men istedet for at indeholde en værdi, indeholder de adressen på blokken der blev allokeret. Følgende eksempel viser hvad der sker når koden `A a; (...) a = new A();` udføres, og hvor klassen `A`'s størrelse er 4. Værdien af `hobptr` ved start (i dette tilfælde 30), er blot et udtryk for, at hoben ikke er tom på tidspunktet eksemplet gennemgås.

Eksempel #7.4:

Før allokering af A
`hobptr == 30;`
Allokering af objekt med størrelse 4 bytes

```
gem hobptr;
hobptr = hobptr + 4;
return gammel hobptr;
a = retur-værdi;
```

Referencen til objektet får altså værdien 30 (adressen på hvor i hoben objektet befinder sig), mens `hobptr` har fået værdien 34. Når et ny objekt allokeres, bliver referencens værdi derfor 34.

7.4 Metoder & metodekald

Metoder repræsenteres i assembler med procedurer. Da det vides, at klasse navne er unikke (sektion 1.3 side 5), kan metoderne navngives som `klassenavn_metodenavn`. Denne konstruktion gør det let at generere assembler-kode, når et metodekald mødes.

Der findes to slags metodekald; Metodekald i eget objekt og metodekald i andre objekter . Måden de to tilfælde håndteres på er næsten ens. I det følgende tages udgangspunkt i metodekald i andre objekter, som også er skitseret på figur 7.3 side 48. Den eneste forskel på metodekald i eget og andre objekter er, at metodekald i eget objekt ikke genererer kode der ændrer værdien af `bx` registret (this referencen for objektet).

- `bp` gemmes, da metoden definerer sin egen `bp`.
- `bx` gemmes, da `bx` skal pege på et andet objekt.
- Assembler understøtter ikke argumenter til procedurer, de push'es derfor på stakken.
- `bx` sættes til værdien af referencevariablen (der indeholder this adressen) og `call` udføres, hvilket medfører returadressen push'es.

I funktionen sker følgende

- `bp` sættes til `sp-2`, da dette er elementet efter retur-adressen på stakken.
- Antallet af lokale variable gange udføres `push 0`. De lokale variable initialiseres altså til 0.

Lige før proceduren forlades udføres

- Antallet af lokale variable gange udføres `pop cx`.

Det øverste element på stakken er returadressen, der hoppes til når `ret` mødes. Efter et metodekald, skal `bx, bp` gendannes, hvilket sker med:

- `pop bx, pop bp` udføres for at genskabe ordnen i det gamle objekt så det er i samme tilstand som før metodekaldet.

7.5 Udtryk og sætninger

Sætninger og udtryk opererer på værdier på stakken. Udtrykket `2+3` pop'er altså 2 og 3 adderer dem og push'er resultatet.

7.5.1 Boolsk repræsentation

Måden de boolske udtryk repræsenteres på er `0000 0000 0000 0000b` for false, og `1111111111111111b` for true, altså henholdvis `0h` og `FFFFh`. Tallene er valgt udfra det faktum at assemblerkommandoerne `neg, and, or` fungerer uden videre. Var værdierne valgt som hhv. 0 og 1 (`0000 0000 0000 0001b`), ville `neg` ikke fungere, da `neg 1` giver `FFFEh` (`1111 1111 1111 1110b`) fremfor `0`.

— Kapitel

8 Kodegenerering

Dette kapitel tager udgangspunk i, at læseren har forstået det tidligere kapitel om specielt lageradministrationen og hvorledes klassers indhold bliver repræsenteret. Kapitlet skitserer hvorledes `CodeGenerator` genererer kode for de resterende dele af Qjava.

8.1 Overordnet betragtning

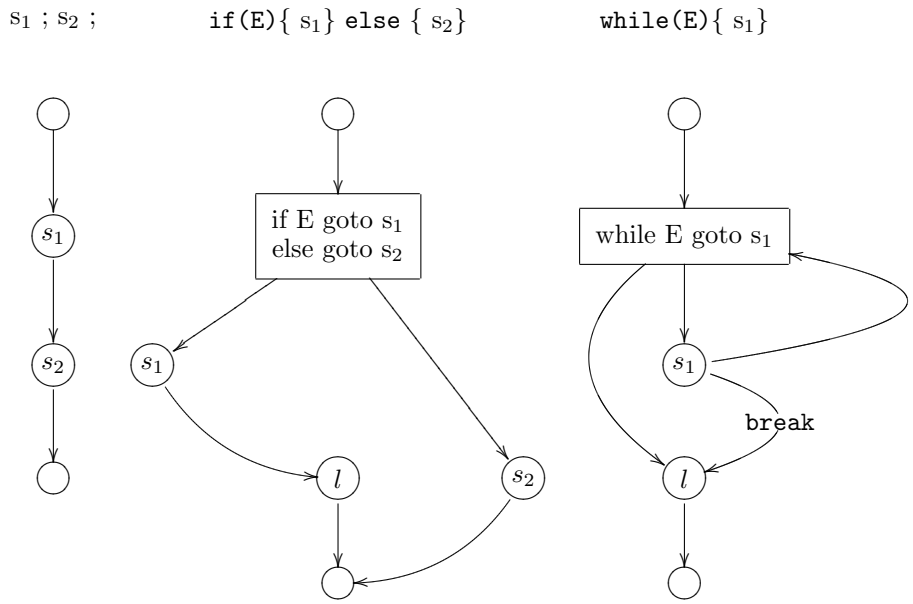
Før den egenlige kodegenerering skal finde sted, kan det være fordelagtigt at illustrere rutediagrammer, der viser hvorledes strukturerede programmer tager sig ud. Figur 8.1 side 54 viser ruterne programmer. `s` repræsenterer sætninger, dvs. ⟨sentences⟩ i grammatikken, mens `E` repræsenterer udtryk, ⟨E⟩ i grammatikken.

Diagrammet til venstre på figuren viser, at sætninger genereres kronologisk, og individuelt uden hensyn til hverken foregående eller kommende sætninger. Dette gælder for alle sætninger i Qjava, med to undtagelser, nemlig sætninger indeholdt i `if` og `while`.

Diagrammet i midten viser `if` der branch'er eller forgrener afviklingsforløbet. Figuren viser, at evalueringen af `if` genereres selvstændigt, og der derfra hoppes til $\texttt{s}_1$ eller $\texttt{s}_2$, der genereres som vist i det venstre rutediagram. Label `l` medtages da $\texttt{s}_1$ efter endt generering skal hoppe til slutningen af `if`. Hvis dette hop ikke blev foretaget, ville kodeblokken $\texttt{s}_2$ blive udført.

Rutediagrammet til venstre viser `while`-løkken. Koden $\texttt{s}_1$ skal genereres sammen med informationen om navnet på label `l`, til de tilfælde hvor `break` mødes.

Det bemærkes måske at metoder og klasser ikke er medtaget i rutediagrammerne. Grundene til dette er, for det første er de meget svære at skitsere, og for det andet kan de betragtes "indpakning af koden". Det eneste der er interessant under kodegenereringen af en metode er, hvor metodens argumenter og variable befinder sig, samt hvad navnet er på en label placeret i bunden af metoden (som anvendet der hoppes til hvis `return` mødes. Bemærk at viden om placeringen af metoden i en



Figur 8.1: *Rutediagram for Qjava, hvor de tomme cirker noterer mulig eksistens af anden kode. s noterer sætninger, E noterer udtryk og l noterer label der kan hoppes til.*

klasse, metodens funktionalitet osv. er fuldstændig underordnet. "Indpakningen" består så og sige af lokale variable, argumenter samt en label. Klasser er det i kodegenerations sammenhænge på dette niveau kun, at `bx` registret indeholder this-referencen. `bx` administreres ved tilgang til variable i andre objekter og ved funktionskald til andre objekter.

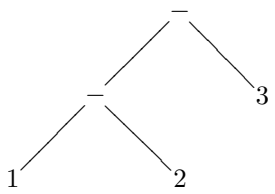
8.2 Kodegenerering

Som det ses udfra skitseringen af syntaksen for assembler i sektion 7.1.1 side 45, er det første problem at omforme Java udtryk til noget der i de fleste tilfælde har formen: `COMMANDO` a_1 `,` a_2 . Således bliver udtrykket `1-2-3` til

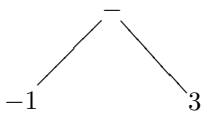
t_1 `= 1 - 2`
t_2 `=` t_1 `- 3`

Omvendt er dette ikke et stort problem, da f.eks. alle udtryk allerede er på denne form, i kraft af måden de repræsenteres på i parsertræet. Kodegenereringen for ovenstående additionsudtryk vil derfor være `generer venstre træ; generer højretræ`. Herefter vides det, at der findes to "mellemresultater" på stakken. Der

skrives "mellemresultater" da disse enten kan være tal, eller resultat af en anden beregning. Genereringen af venstretræet vil være resultatet af genereringen af `1, -, 2`, som værende en mellemregning. Genereringen af højretræet svarer til genereringen af `3`. De to "mellemregninger" pop'es fra stakken og subtraktionen udføres, hvorefter resultatet push'es på stakken.



bliver til



der tilsidst bliver til -4

Kodegenereringen fra parsertræ til assemblerkode foregår udfra følgende to grundprincipper

- Givet en række sætninger $(1 \ldots n)$, kan sætningerne oversættes i kronologisk rækkefølge. Indholdet i hver sætning, $0 \ldots n$ sætninger/udtryk, oversættes på samme vis kronologisk.
- Nettoeffekten af et udtryk, er præcis én ekstra værdi på stakken.

Det skal bemærkes, at første regel skal forstås rekursivt.

8.2.1 Eksempel på anvendelse af principperne

EKSEMPEL #8.1:

```
a = 2+3;

while(i < 100)
{
    i = i + 4+5;
    b = 6+7;
}
```

Ifølge første princip kan vi opsplitte oversættelsen i henholdsvis
`a = ...` og `while(...){...}`.
`while` kan ifølge det andet princip opsplittes til:
`(...), {...}` , altså en conditiondel og en krop,

hvor kroppen ({...}) kan opsplittes
`i = ...` og `b = ....`
I alt 4 dele:

- `a = 2+3`
- while condition: `i < 100`
- while krop: `i = i + 4 + 5`
- while krop: `b = 6+7`

Oversættelsen er skitsered på figur 8.2 side 57, hvor anden kolonne er dét der skal oversættes, tredie kolonne skitserer kodegenereringen på abstrakt plan, og sidste kolonne er et forsøg på at skitsere hvorledes dette kunne tage sig ud i assembler. # noterer et register. Linierne 2–7 viser, hvorledes mellemresultaterne 2 og 3 gemmes på stakken. Additionsoperatoren pop'er de to øverste elementer, adderer og push'er resultatet (der evt. kunne være et mellemresultat, hvis udtrykket `2+3` havde være længere. Det ses at resultatet af udtrykket var et ekstra element på stakken, nemlig 5. Assign operatoren på linierne 8–9 henter også blindt værdierne der skal anvendes fra stakken. Eksemplet vil ikke blive yderligere forklaret, da principperne er tydeligt nok illustreret.

For en god ordens skyld bør det bemærkes, at linierne 15–18 er en forsimpling af hvorledes kodegenereringen i Qjava fungerer. Forsimplingen er lavet for at bevare overblikket. Det der reelt sker i kodegenereringen er, at linierne 15–16 udføres, men på det der svarer til linie 16 hoppes ikke til `less`, men til et sted der push'er det boolske udtryk for true (`FFFFh`) og istedet for `jmp whileEndLabel` hoppes til et sted hvor der push'es det boolske udtryk for false (`0h`). Først herefter foretages en `push #1, cmp #1, 0` er `#1` lig `FFFFh` (true) hoppes til label `less` ellers hoppes til label `whileEndLabel`.

8.2.2 Metoder og metodekald

Genereringen af metoder sker ved følgende: For hver metode (Fncdef) der mødes ved traversering af parsertræet, dan følgende kode i code segmentet.

```
klassenavn_metodenavn PROC

; her genereres metodens krop

klassenavn_metodensnavn ENDP
```

Metodekald genereres let. Givet koden `A a = new A();` og mødes udtrykket `a.f();` under traverseringen, genereres følgende kode: `call typeVariablenHar_f` , hvor `typeVariablenHar` i dette tilfælde er "A", som findes med en opslagsmetode i symboltabellen Der genereres altså `call A_f`. Bemærk administrationen af registrene `bp, bx` er fjernet fra eksemplet.

Linie	Del	Abstrakt	pseudo assembler
1	`a = 2+3`	code(a =)	
2		code(2)	`push 2`
3		code(3)	`push 3`
4		code(ADD)	`pop #1`
5			`pop #2`
6			`add #1, #2`
7			`push #1`
8			`pop #1`
9			`mov variable a, #1`
10			`start_while:`
11	`i < 100`	code(i)	`push variable i`
12		code(100)	`push 100`
13		code(LESS)	`pop #1`
14			`pop #2`
15			`cmp #1, #2`
16			`jl less`
17			`jmp end_while`
18			`less:`
19	`i = i+4+5`	code(i =)	
20		code(i)	`push variable i`
20		code(4)	`push 4`
.			
.			
30	`b = 6 + 7`	code(b=)	
31		code(6)	`push 6`
.			
.			
40			`jmp start_while`
41			`end_while:`

Figur 8.2: *Figuren illustrerer hvorledes Java kan genereres til assembler.*

8.3 Byggeklodser

Som vist fra rutediagrammet og de efterfølgende eksempler, er oversættelsen i mange tilfælde blot at indsætte "byggeklodser" for de forskellige slags udtryk og sætninger sammen med en overordnet administration som beskrevet i det forrige kapitel. I de følgende undersektioner, gennemgås assemblerkoden for byggeklodserne for sætninger og udtryk. Da koden er meget simpel vil koden stå ukommenteret.

Generelt for implementationerne knytter sig nogle få forholdsregler.

- De monadiske operatorer (`!, -`) forventer, at dét de skal operere på, forefindes øverst på stakken. Implementationerne `pop`'er stakken, udfører operationen og `push`'er derefter resultatet.
- De duale operatorer (de restende operatorer med undtagelse af `=`) forventer, at dét de skal operere på, forefindes øvers og næstøverst på stakken. Endvidere forventes det øverste element at repræsentere højresiden af operatoren, når udtrykket skrives i sproget Qjava. Implementationerne `pop`'er de to øverste elementer fra stakken, udfører operationen og `push`'er derefter resultatet.
- Specielt for operatorene `|, ||, &, &&, <, <=, !=, ==` at disse ikke push'er en værdi, men et boolsk udtryk, som repræsenteres som henholdsvis `0h` og `FFFFh`. Videre begrundelse for dette findes i sektion 7.5.1 side 52.
- Generer kode til henholdsvis stak eller Hob tilgang beskrevet i de tidligere kapitler. ⟨mem⟩ (som forefindes i sektion 8.3.11 side 59) kan derfor være `hob[bx+index]` eller `bp-index`.

8.3.1 +

```
pop  dx
pop  ax
add  ax, dx
push ax
```

8.3.2 -

```
pop  dx
pop  ax
sub  ax, dx
push ax
```

8.3.3 - (monadisk)

```
pop  ax
neg  ax
push ax
```

8.3.4 *

```
pop  dx
pop  ax
mul  dx
push ax
```

8.3.5 /

```
pop  si
pop  ax
xor  dx, dx
div  si
push ax
```

8.3.6 %

```
pop  si
pop  ax
xor  dx, dx
div  si
push dx
```

8.3.7 !

```
pop  ax
not  ax
push ax
```

8.3.8 !=

```
pop  dx
pop  ax
cmp  ax, dx
jne  neq_1
xor  ax, ax    ; is ==
jmp  neq_end1
neq_1:
mov  ax, FFFFh ; is !=  true result
neq_end1:
push ax
```

8.3.9 <=

```
pop  dx
pop  ax
cmp  ax, dx
jle  lequal_1
xor  ax, ax    ; is not <=
jmp  lequal_end1
lequal_1:
mov  ax, FFFFh ; is <=  true result
lequal_end1:
push ax
```

8.3.10 <

```
pop  dx
pop  ax
cmp  ax, dx
jl   less_1);
xor  ax, ax    ; is not <
jmp  less_end1
less_1:
mov  ax, FFFFh ; is < // true result
less_end1:
push ax
```

8.3.11 =

```
pop ax
mov <MEM>, ax
```

8.3.12 ==

```
; equal
pop  dx
pop  ax
cmp  ax, dx
je   eq_1
xor  ax, ax  ; is != // false result
jmp  eq_end1
```

```
eq_1:
mov  ax, FFFFh ; is == // true result
eq_end1:
push ax
```

8.3.13 `|, ||`

```
pop  dx
pop  ax
or   ax, dx
push ax
```

8.3.14 `&, &&`

```
pop  dx
pop  ax
and  ax, dx
push ax
```

8.3.15 `break`

Hvis `break` kaldes udenfor en while løkke meldes fejl ved compilering, ellers genereres:

```
jmp while_end_label
```

8.3.16 `return`

Da `return` ikke returnerer værdier, hoppes til den sidste linie i assembler proceduren.

```
jmp fnc_end_label
```

8.3.17 `if`

`if` består af tre blokke henholdsvis
`if(condcode (1)){thencode(2)} else { elsecode (3)}`. Da enten `thencode` eller `elsecode` skal udføres skal der anvendes labels til at kunne hoppe til disse sektioner, endvidere skal der være label til i slutningen af `if` udtrykket (så `thencode` ikke også udfører `elsecode`), samt slutningen af funktionen skal kendes til `return`.

Først genereres condition koden (som princip 1 foreskriver) og evalueres. Resultat gemmes på stakken som princip 2 foreskriver. Resultatet `pop`'es og evalueres og der udføres hop til elsecode eller thencode alt efter resultatet. De to kodeblokke genereres som princip 1 foreskriver.

```
; tree == the parsetree of instanceof If
; generate condition -> eGen(tree.getCond());
pop  cx
cmp  cx, 0  ; compare condition code
je   else1

then1:
; generate then code ->  sentenceGen(tree.getThen());
jmp  ifend1

else1:
; generate else code ->sentenceGen(tree.getElse());

ifend1:
```

8.3.18 `while`

`while` består af to blokke henholdsvis:
`while(condcode (1)){whilecode (2)}`. Til koden skal der endvidere produceres et "startWhile" og et "slutWhile" label.

```
; tree == the parsetree of instanceof While
start_while1:
; generate condition code ->  sentenceGen(t.getCond());
pop  ax
cmp  ax, 0 ; compare condition code
je   end_while1

;generate while-body -> sentenceGen(tree.getWhile());
jmp  start_while1  ; loop once more

end_while1:
```

8.4 Bootstrapping

Funktionsdeklarationen `static void main()` erklærer starten på et program. Qjava understøtter ikke `static`, men ignorerer det af hensyn til kompabilitet med Java. Når oversætteren møder funktionen `main()` (fra sektion 1.3 side 5 er

det givet der kun er en `main()` pr. program. Genereres en label "START" (i opstartskoden genereres kaldet `jmp START`). Herefter allokeres objektet funktionen befinder sig i (`<SIZEOFCLASS>` erstattes af klassens størrelse). Derefter justeres registrene `bp, bx`. Slutteligt genereres kroppen af main-funktionen.

```
START:
    mov  cx, <SIZEOFCLASS>
    call new
    mov  bp, sp
    sub  bp, 2
    mov  bx, ax
    .
    .
```

8.5 Standard funktioner

Qjava implementerer dele af standardpakkerne der følger med Java. For `Integer, String` gælder det, at de har overholdt samme navnekonventioner som dannelsen af andre funktioner, hvilket har den fordel at hvis `String s; s = "hej ib";` gælder og udtrykket `s.length()` mødes, bliver `call`-kaldet genereret automatisk, som forklaret i eksemplet med `a.f()` kaldet i sektion 8.2.2 side 56.

Kodegenerering i forbindelse med metodekald til pakkerne `Integer, System` bliver håndteret individuelt for hver metode.

Da standardfunktionerne er simple, forklares deres implementation udelukkende. Der henvises derfor til slutningen af appendix J side 134 hvor implementationen er at finde.

8.5.1 Integer

Fra Integerpakken implementeres udelukkende metoden
`static String Integer.toString(int)` (klassen `Integer` implementeres ikke). Resultatet er internt, at en pointer til den dannede streng push'es på stakken. Implementationen gennemgår følgende faser i oversættelsen af et tal til en streng:

- Er tallet negativt gemmes denne information og tallet multipliceres med -1.
- Sålænge tallet ¿ 0; divider tallet med 10 og `push` resten ved divisionen, herved gemmes cifret længst mod højre.
- Alloker en ny `String` og fyld den vha. `pop`.

Ved anvendelse af stakken til temporær lagring af cifrene returneres cifrene i den omvendte og dermed korrekte rækkefølge.

8.5.2 String

String typen implementeres ved at være en klasse der indeholder et størrelsesfelt, der angiver længden af strengen, mens resten af felterne indeholder tekststrengens respektive bogstaver.

`String s = "blabla"`

Længde	6
T	b
e	l
g	a
n	b
	l
	a

Da hvert tegn, samt størrelsesfeltet repræsenteres ved 16 bit, fylder tekststrengen "Datamat!" derfor $(8 \cdot 2) + 2$ bytes i hoben, når den repræsenteres i et `String` objekt. Referencer til `String` objekter peger på det første element i instansen, nemlig længden af strengen, der for "Datamat!" er lig 8. Indexeringen af de enkelte bogstaver foregår som i Java ved at første tegn ligger på plads 0 og så fremdeles.

Bemærk at strengene ikke er 0-termineret. En strengs længde bestemmes udelukkende ved størrelsesfeltet.

Foruden selve typen implementeres endvidere metoderne `concat(), length()` og `charAt()` der alle har samme funktionalitet som i Java.

concat()

`String concat(String)` funktionen fungerer ved at oprette en ny `String` med summen af de to `String` instanser (adderer første element fra hver instans), hvorefter den nye streng fyldes med indholdet af de gamle strenge.

Der kontrolleres ikke for om den ene eller begge referencer er `null`

length()

`int length()` læser størrelsesfeltet i instansen og returnerer denne. Der kontrolleres ikke hvorvidt referencen er `null`

charAt()

`char charAt(int)` returnerer ASCII værdien af den pågældende possition. Da hvert bogstav fylder 2 bytes, og da instansen indeholder et størrelsesfelt, findes bogstavet i hoben ved `hob[bx+(possition*2)+2]`. Der kontrolleres ikke for ugyldige eller negative værdier af possitionen. Endvidere skal det bemærkes at `System.out.print()` ikke kan udskrive typen char.

8.5.3 System

System.exit

`System.exit()` skal øjeblikkeligt stoppe afviklingen af programmet. Dette gøres med et DOS interupt kald.

System.in.read

Metoden læser et tegn fra tastaturet og er implementeret med et DOS interrupt kald.

System.out.print

Metoden udskriver et `String` instans til skærmen. Implementationen læser størrelsesfeltet og udskriver derefter tegnene et for et med et DOS interrupt kald.

8.6 Registre

De mange registre der findes i 80x86 har ikke alle samme funktionalitet. Således kan visse registre anvendes til at referere til hukommelse mens andre ikke kan. Det er derfor ikke en tilfældighed at `bx` og `bp` registrene står for den overordnede administration. Nedenstående tabel viser i grove træk hvorledes registrene anvendes af compileren.

Registre	Anvendelse
ab, cx, dx, si	Aritmetik og mellemresultater
bx, bp	`this` pointer for klasser, funktioer og variable.
si, di	Pointere relative offset, f.eks. bx
sp	Indeholder adressen på det øverste element på stakken.

8.7 Opstarts- og Afslutnings- kode

Ligesom Java kræver en "`static void main()`"-metode, kræver DOS en opstartskode før assemblerprogrammer kan afvikles.

Opstartskode

Opstartskoden fortæller assembleren hvilken memorymodel der anvendes, hvor stor stakken skal være og hvor data-segmentet befinder sig i memory. Slutteligt hoppes til label "START", hvorfra kroppen af funktionen "`static void main()`" er placeret.

```
DOSSEG
.MODEL SMALL
.STACK 200h

.DATA
    hobptr  dw 0
    hob     dw 32605 dup(0) ; allocate hob
    ; error messages
    oomstr db 10,13,'Out of memory! Can not allocate another class.','$',0,0

.CODE
     mov ax, @data
     mov ds, ax
     jmp START
```

Afslutningskode

Pogrammer kan på et hvilket som helst tidspunkt afsluttes med DOS interrupt 21 med argument `4ch`.

```
mov ah,4ch
int 21h
```

Slutteligt kræver assembleren ordet "`END`" som sidste ord i sourcekoden.

8.8 Implementation

Udover implementeringen af standardfunktionerne, blev `CodeGenerator` implementeret ved en metode for hver knudetype der fandtes i parsertræet. Den faste struktur var derfor klart en fordel, da alle felter var eksplicitte. Implementationen foregik da også næsten smertefrit, og ændres strukturen i parsertræet er det forholdsvis enkelt at gøre det samme i kodegeneratoren. Den eneste afvigelse fra parsertræet er hvis $\langle E_{10} \rangle$ genkendes som en funktionskald, oprettes en `fncCall` knude, som sendes over i metoden `fncCallGen`, der genererer et funktionskald.

— Kapitel

9 Kørselsvejledning

For at skabe en *.exe fil, der kan afvikles under DOS, skal man igennem flere stadier, denne korte vejledning vil vise hvilke og hvordan. Vejledningen tager udgangspunkt i et korrekt installeret Javamiljø (JDK) samt assemblermiljø (TASM).

.java ⟶ .asm

Først skal ens Java sourcekode compileres til assemblerkode, dette gøres ved at skrive

```
C:\>java Qjava filnavn
```

Dette afvikler Qjava compileren, der læser sourcekoden "filnavn.java", og skriver filen "filnavn.asm". Bemærk, at der ingen endelse findes på filen der gives som argument til compileren.

.asm ⟶ .exe

Sidste stadie kræver både en assemblering og en linkning. Dette udføres ved at skrive

```
C:\>tasm filnavn.asm
```

```
C:\>tlink filnavn.obj
```

Hvis alt har forløbet problemfrit, findes nu "filnavn.exe", der kan afvikles.

— Kapitel

10 Afprøvning

Dette kapitel vil vise test af compilerens enkelte dele samt tilsidst den færdige version.

- Lexer, sektion 10.1 side 68.
- Parser , sektion 10.2 side 70.
- Symbol tabel , sektion 10.3 side 74.
- Kodegenerering, sektion 10.4 side 75.

Oversigt

Alle testkørsler er, medmindre andet er noteret, udført efter principperne "ekstern testning" der kort er beskrevet i [Hansen;1995]. Testmetoden tager ikke udgangspunkt i programkoden, men i programkodens funktionalitet. Koden er altså reduceret til en "blackbox", hvor det er transformationen fra input til output er i fokus. Der testes således for om basistilfælde kendes, men ikke kombinationer imellem disse. Det vil sige, hvis programmet kan genkende tallet `42`, kan den også genkende alle andre tal, og genkender den `1+2`, genkender den også alle andre kombinationer af to tal og + operatoren.

I lexeren testes der for 1) kan alle keyword/operatorer genkendes, 2) kan strenge/tal genkendet, og 3) seppereres ordene rigtigt.

Parseren testes med en, efter grammatikken syntatisk korrekt, program kode. Der testes altså for 1) om parseren kan parse koden problemfrit, 2) om parseren genkender de korrekte produktioner og 3) om den tildeler prioriteterne for operatorene korrekt.

Symboltabellen testet for om dens metoder fungerer korrekt.

Afprøvningsstrategien for kodegeneratoren er ikke en tilbundsgående undersøgelse af alle semantiske tilfælde. Målet med testen er at se om der kan genereres fungerende kode for operatorer, standardfunktioner og basistilfælde af grammatikkens sætninger og udtryk. Afgrænsningerne i afprøvningen afbegrænser dermed også pålideligheden af compileren til afgrænsningen af afprøvningen.

10.1 Lexer

Før selve testargumentationen fremføres, forklares måden testresultaterne er forelagt på. I Tabel 10.1 er hver celle opdelt i to dele. "#" noterer hvor mange tokens lexeren har genkendt, mens "Val" angiver den hvad token blev genkendt som. I visse tilfælde er der knyttet ekstra information til et token, disse informationer er placeret imellem "()".

Til afprøvningen af klassen er tre forskellige testdata filer anvendt. Den første testdata fil er opdelt i tre faser:

Eksempel #10.1:

```
/* Test datafil 1
 * for den lexikalske analyse...
 */

// Første fase - tegnsekvenser
, . ; : ( ) { } [ ]
+ - * / < % = != == <=
&& || & | ! break char class
else if int new null return
String void while

// Anden fase - værdi-test (char, tal og strenge)
'A' 12 "a str"

// Tredie fase - variabelnavn/separatorer test
foo2  2foo (foo foo( yif
ify 2int ((
a.b    c.d()    a. +
```

Første fase består i at genkende alle reserverede tegnsekvenser (hvis kommentarer genkendes ignoreres disse). Dette er Tokens 0–36 i tabel 10.1. Det bemærkes at de to sæt kommentarer i starten af testfilen ikke har givet anledning til noget output, dvs. lexeren genkender kommentarer. Endvidere ses det at 1 og 2 returns samt 1 space betragtes som separator og "whitespace" hvorfor de ignoreres.

Anden fase består i værdi-genkendelse, altså om lexeren forstår tal, strenge og tegn. Dette er token 37–39. Det ses testdata blev genkendt som hhv. `A`, `12` og `Str val` som i testfilen.

Den sidste fase er en kombineret test af om variabelnavne kan genkendes, og hvad der er seperatorer. Token 40 viser, at tal i slutningen af en tegnsekvens ikke er en separator. Token 41–42 viser, at tal er en sepperator, hvis det kommer først i tegnsekvensen, dette ses ved tegnsekvensen opsplittes i to Tokens. Token 43–46 er en lignende test, hvor der istedet anvendes en operator istedet for tal, i dette tilfælde en "(". De to tegnsekvenser giver fire tokens, hvorfor det konkluderes, at operatorer altid virker som sepperatorer. Token 46–47 viser, at lexeren genkender den længst mulige tegnsekvens, da `if`-delen i begge tegnsekvenser ikke resulterede i et `if`-token. Token 49–50 viser tal og keywords separeres korrekt. Token 51–52 viser dobbelt operator virker, 53–54 viser genkendelsen af ⟨name⟩, mens 26, 30, 34, 57 viser genkendelsen af ⟨id⟩.

#	Val	#	Val	#	Val	#	Val
0	,	1	.	2	;	3	:
4	(	5	)	6	{	7	}
8	[	9	]	10	+	11	−
12	*	13	/	14	<	15	%
16	=	17	!=	18	==	19	¡=
20	&&	21	||	22	&	23	|
24	!	25	break	26	id (char)	27	class
28	else	29	if	30	id (int)	31	new
32	null	33	return	34	id (String)	35	void
36	while	37	chVal('A')	38	intVal(12)	39	StrVal(a str)
40	id (foo2)	41	intVal(2)	42	id (foo)	43	(
44	id (foo)	45	id (foo)	46	(	47	id (yif)
48	id (ify)	49	intVal(2)	50	id (int)	51	(
52	(	53	name (a.b)	54	name (c.d)	55	(
56	)	57	id (a.)	58	+	59	¡EOF¿

Tabel 10.1: *Testudskrift for klassen* `Lexer`.

Testkørsel 2

Resultatet af testkørsel 2 er:
`Error(1): Char definition more than 1 character long.`

Testkørsel 3

Resultatet af testkørsel 3 er:
`Error(1): keyword "switch"is not yet supported.`

For begge testkørsler gælder det, at compileren stoppede kørslen efter udskrift.

EKSEMPEL #10.2:

```
/* 2. testfil */
'aa'
```

EKSEMPEL #10.3:

```
/* 3. testfil */
switch i;
```

10.1.1 Ikke-problematiske fejl

En fejl den anvendte testmetode ikke fangede er tilfælde såsom `<␣=` Altså hvor de to tokens klart er separeret af en space (␣). Lexeren oversætter dog det angive eksempel til et `<=` token. For implementationen af compileren er dette ikke noget problem da det ikke strider imod principperne for fejlmelding i kravsspecifikationen. Endvidere tillader Javasyntaxen ikke en sådan konstruktion. Slutteligt kunne man med rette hævde, at skriver man `<␣=` mener man faktisk `<=`.

Forklaringen på opførslen finder vi i koden, hvor der her kun vises de for problemstillingen væsenligste dele (for en fuld kodeudskrift af lexerens testkode henvises til appendix C side 98. `fetchOperator()` bliver kaldt i de situationer hvor Lexeren har læst et "<" — altså situationer hvor der muligvis er tale om et sammensat token.

EKSEMPEL #10.4:

```
class Lex extends StreamTokenizer implements TokenNames
{
...
  private Token fetchOperator(char c) throws IOException
  {
    int nt = super.nextToken();
...
    String s = "" + c + (char) nt;
...
    if(s.equals("<=")) return( new Token(LEQUAL) );
...
```

Som det ses, er det `StreamTokenizer` der står for indlæsningen, og da denne 1) ignorerer spaces 2) stopper og returnerer for hver operator, har man ingen mulighed for at vide om `StreamTokenizer`'en har mødt `<␣=` eller `<=` konstruktionen.

Da der i forhold til kravsspecifikationen ikke opstår vanskeligheder, uddybes problemstillingen ikke videre.

10.2 Parser

Testen af parseren tager udgangspunkt i følgende testfil,

EKSEMPEL #10.5:

```
/*
 * test for parseren 23.11.99
 */

class A
{
    int i;// vardef i classscope
    ;
    void a() // fncdef uden argumenter
    {
        a(); // fnccall til eget og andre objekter
        b(1, 2,3);
        B.c();

        if(a) // if
        {
            aaa = 2; // assign
        }
        else
        {
            B.j = 3; // assign i andet objekt
        }
    }
```

```
    void b(int k, String l) // fncdef med argumenter
    {
        break;
        return;

        while(i != 2) // en while
        {
            return(); // return med parenteser
        }

        ; // tom linie (kun ";")
    }
}


// multible klasser
class B
{
    void c()
    {
        // vardef i function scope
        int j;
        int k;

        // E-E11     testes
        e = 1 || 2;
        e1 = 3 && 4;
        e2 = 5 | 6;
        e3 = 7 & 8;
        e4_1 = 9 == 10;
        e4_2 = 11 != 12;
        e5_1 = 13 < 14;
        e5_2 = 15 <= 16;
        e6_1 = 17 + 18;
        e6_2 = 19 - 20;
        e7_1 = 21 * 22;
        e7_2 = 23 / 24;
        e7_3 = 25 % 26;

        e8 = new B();
        e9_1 = !27;
        e9_2 = -28;
        e10_1 = foo;
        e10_2 = foo.bar;
        e10_3 = foo(29+30);
        e10_4 = foo(31, 32);
        e10_5 = "en streng";
        e11_1 = 2;
        e11_2 = (3);

        // en kombinations af flere E'er
        if( (a != 2 && b == 3) || (c < -2|3) || !(d <= e&4%5*6+7- -8/9) )
        {
            int x;
        }
        else
        {
```

```
            }

            i = 2+3*4;
            i = (2+3)*4;
            i = 2-3-4;
        }
    }
```

For at kunne vurdere resultaterne, fortæller parseren hvorledes den opfatter et givent udtryk, f.eks. med `(assign)`. Endvidere sættes parenteser om alle monadiske og duale operatorer for at kunne vise prioriteten i evalueringsrækkefølgen. For læsbarhedens skyld, er testoutput modificeret med ekstra indrykninger og tilpasning af linieskift.

Gennemgangen af afprøvningen foregår, af hensyn til overblikket, udfra grammatikken på figur 4.3 side 32, istedet for en kronologisk opremsning af eksemplet. Der henvises iøvrigt til hele kapitel 4 side 24 for gennemgangen af grammatikken. I gennemgangen vises det, at hver non-terminal eksisterer i eksemplet, og at alle definitionerne på non-terminalen eksisterer (f.eks. kan ⟨sentece⟩ repræsenteres ved både ⟨while⟩ og ⟨fnccall⟩ osv). Argumentationsformen er meget kompakt i den forstand, at non-terminalen opskrives, hvorefter beviset for at non-terminalen eksisterer, og at alle definitioner er afprøvet. Non-terminalens definition gennemgås ikke, da en sådan gennemgang er at finde i kapitel 4.

⟨S⟩ Eksemplet indeholder klasserne `A`, `B`.

⟨classdef⟩ Klassen `A` dækker definitionen.

⟨classcontents⟩ `A.i` opfylder ⟨vardef⟩, `A.a()` opfylder ⟨fncdef⟩ og "`;`" forefindes i klasse scope i `A`.

⟨vardef⟩ Variablen `A.i` opfylder ⟨vardef⟩

⟨fncdef⟩ Funktionerne `A.a()`, `A.b()` defineres hhv. uden og med argumenter.

⟨sentences⟩ ⟨vardef⟩, ⟨fnccall⟩, ... , "`;`" forefindes i eksemplet.

⟨fnccall⟩ Øverst i `A.a()` repræsenterer `a()`, `B.c()` hhv. ⟨id⟩ og ⟨name⟩. kombinationen. `b(1, 2,3);` viser, der accepteres multible argumenter.

⟨if⟩ `if(a)...` i `A.a()` og `if( (a != 2...)` nederst i `B.c()` viser `if` med og uden `else`-krop.

⟨while⟩ Eksisterer i `A.b()`.

⟨break⟩ Findes i `A.b();`.

⟨return⟩ Findes med og uden `()` i `A.b()`.

⟨assign⟩ Variablen `aaa` tildeles i `A.a()`.

⟨E⟩-⟨E_{11}⟩ Er repræsenteret i `B.c()`

Resultatet viser de to næstsidste tildelingssætninger (2+3*4) i eksemplet, at prioriteten for evaluering kan ændres ved hjælp af parenteser. Den sidste tildelingssætning (2-3-4) viser, at udtryk med flere operatore med ens prioritet, placeres korrekt i parsertræet. Slutteligt er det lange sammensatte if-udtryk medtaget for at vise, at operatorene kan sammensættes uden problemer.

```
class A
{
    (vardef) int i

    (fncdef) void a()
    {
        (fnccall) a()
        (fnccall) b(1, 2, 3)
        (fnccall) B.c()
        if(a)
        {
            (assign) aaa = 2
        }
        else
        {
            (assign) B.j = 3
        }
    }

    (fncdef) void b(int k, String l)
    {
        break
        return
        while((i != 2))
        {
            return
        }
    }
}

class B
{
    (fncdef) void c()
    {
        (vardef) int j
        (vardef) int k
        (assign) e = (1 || 2)
        (assign) e1 = (3 && 4)
        (assign) e2 = (5 | 6)
        (assign) e3 = (7 & 8)
        (assign) e4_1 = (9 == 10)
        (assign) e4_2 = (11 != 12)
        (assign) e5_1 = (13 < 14)
        (assign) e5_2 = (15 <= 16)
        (assign) e6_1 = (17 + 18)
        (assign) e6_2 = (19 - 20)
        (assign) e7_1 = (21 * 22)
        (assign) e7_2 = (23 / 24)
        (assign) e7_3 = (25 % 26)
        (assign) e8 = new B()
        (assign) e9_1 = (! 27)
        (assign) e9_2 = (- 28)
```

```
        (assign) e10_1 = foo
        (assign) e10_2 = foo.bar
        (assign) e10_3 = foo((29 + 30))
        (assign) e10_4 = foo(31, 32)
        (assign) e10_5 = "en streng"
        (assign) e11_1 = 2
        (assign) e11_2 = 3
        if(((((a != 2) && (b == 3)) || ((c < (- 2)) | 3)) ||
        (! ((d <= e) & ((((4 % 5) * 6) + 7)         - ((- 8) / 9))))))
        {
            (vardef) int x
        }
        else
        {
        }

        (assign) i = (2 + (3 * 4))
        (assign) i = ((2 + 3) * 4)
        (assign) i = ((2 - 3) - 4)
    }
}
```

10.3 SymbolTable

Med udgangpunkt i eksemplet i sektion 10.2, afprøves symboltabellen.

Efter endt parsing udføres følgende kode, hvor `sbtb` er `SymbolTable`, altså symboltabellen.

```
System.out.println("1. A = "         + sbtb.classSize("A"));
System.out.println("2. B = "         + sbtb.classSize("B"));
System.out.println("3. B.c() = "     + sbtb.fncSize("B","c"));
System.out.println("4. B.c().x = "   + sbtb.varIndex("B","c","x"));
System.out.println("5. B.k = "       + sbtb.varIndex("B", null, "k"));
System.out.println("6. vartype i = " + sbtb.varType("A", "a", "i"));
sbtb.add("char", "A", null, "c", 42);
System.out.println("7. A.c = "       + sbtb.varIndex("A", null, "c"));
```

Hvilket gav følgende output

```
1. A = 2
2. B = 0
3. B.c() = 6
4. B.c().x = 4
5. B.k = -1
6. vartype i = int
7. A.c = 42
```

Forklaringen af linierne:

1. `A` indeholder `i` == 2 bytes.
2. `B` indeholder ingen felter.

3. Variablen `x` er defineret som den tredie variabel i scope'et `B.c()` og får indexværdien 4 (variable indexeres 0, 2, 4, 6, ...). Det bemærkes, at selvom `x` befinder sig i et `if`-scope, anvendes det i compileren som værende erklæret i `B.c()`'s scope.
4. Variablen `k` findes ikke i `B`'s klasse scope.
5. Der returneres den rigtige type, da `i` erklæres som `int`.
6. Metoden `SymbolTable.add()`, hvor index nummeret selv kan fastsættes virker.

10.4 CodeGenerator

```
class A
{
    String space;
    String strfalse;
    String strtrue;

    void f()
    {
        int i;
        int j;

        // 00000000 00000100 (4) OR 00000000 00000010 (2) = 00000000 00000110 (6)
        System.out.print( Integer.toString(4|2) );  // | test
        System.out.print(space);

        // 00000000 00001100 (12) AND 00000000 00001010 (10) = 00000000 00001000 (8)
        System.out.print( Integer.toString(12&10) );// & test
        System.out.print(space);

        System.out.print( Integer.toString(1+1) );  // + test (2)
        System.out.print( Integer.toString(4-1) );  // - test (3)
        System.out.print( Integer.toString(2*3) );  // * test (6)
        System.out.print( Integer.toString(25/5) ); // / test (5)
        System.out.print( Integer.toString(23%8) ); // % test (7)
        System.out.print( Integer.toString(6- -2) );// - (fortegn) test (8)
        System.out.print(space);


        // ! test på bits 11111111 11111000 (65528) = 00000000 00000111 (7)
        System.out.print( Integer.toString(!65528) );
        System.out.print(space);
        System.out.print(space);



        // == test 1
        j = 1;
        if(j == 1){System.out.print(strtrue);}
        else{System.out.print(strfalse);}

        // == test 2
        j = 0;
```

```
if(j == 1){System.out.print(strtrue);}
else{System.out.print(strfalse);}
System.out.print(space);
System.out.print(space);

// != test 1
j = 0;
if(j != 1){System.out.print(strtrue);}
else{System.out.print(strfalse);}


// != test 2
j = 1;
if(j != 1){System.out.print(strtrue);}
else{System.out.print(strfalse);}
System.out.print(space);
System.out.print(space);


// < test 1
j = 0;
if(j < 1){System.out.print(strtrue);}
else{System.out.print(strfalse);}

// < test 2
j = 1;
if(j < 1){System.out.print(strtrue);}
else{System.out.print(strfalse);}
System.out.print(space);
System.out.print(space);


// <= test 1
j = 0;
if(j <= 1){System.out.print(strtrue);}
else{System.out.print(strfalse);}


// <= test 2
j = 1;
if(j <= 1){System.out.print(strtrue);}
else{System.out.print(strfalse);}


// <= test 3
j = 2;
if(j <= 1){System.out.print(strtrue);}
else{System.out.print(strfalse);}
System.out.print(space);
System.out.print(space);


// ! test på boolske udtryk 1, false
if(!(1 == 1)){System.out.print(strtrue);}
else{System.out.print(strfalse);}


// ! test på boolske udtryk 2, false
if(!(1 == 0)){System.out.print(strtrue);}
```

```
else{System.out.print(strfalse);}
System.out.print(space);
System.out.print(space);



// && test 1
j = 0;
if(j == 1 && j == 0){System.out.print(strtrue);}
else{System.out.print(strfalse);}


// && test 2
j = 0;
if(j == 1 && j == 1){System.out.print(strtrue);}
else{System.out.print(strfalse);}


// && test 3
j = 0;
if(j == 0 && j == 0){System.out.print(strtrue);}
else{System.out.print(strfalse);}
System.out.print(space);
System.out.print(space);



// || test 1
j = 0;
if(j == 1 || j == 0){System.out.print(strtrue);}
else{System.out.print(strfalse);}


// || test 2
j = 0;
if(j == 1 || j == 1){System.out.print(strtrue);}
else{System.out.print(strfalse);}


// || test 3
j = 0;
if(j == 0 || j == 0){System.out.print(strtrue);}
else{System.out.print(strfalse);}
System.out.print(space);
System.out.print(space);



// while test, udskriver 0
j = 0;
while(j == 0)
{
    System.out.print( Integer.toString(j));
    j = j + 1;
}


// while test, udskriver intet
j = 6;
```

```
    while(j < 5)
    {
        System.out.print( Integer.toString(j));
        j = j + 1;
    }
    System.out.print(space);


    // while test, udskriver 1..5
    j = 0;
    while(j < 5)
    {
        System.out.print( Integer.toString(j));
        j = j + 1;
    }
    System.out.print(space);
    System.out.print(space);


    // test af break, udskriver intet
    j = 0;
    while(j < 5)
    {
        break;
        System.out.print( Integer.toString(j));
    }


    // test af return, udskriver ikke mere
    return;
    j = 42;
    System.out.print( Integer.toString(j));
}


// test af metode med argumenter
void g(int x, int y)
{
    int pp;
    pp = 3;
    System.out.print( Integer.toString(pp));
    System.out.print( Integer.toString(x));
    System.out.print( Integer.toString(y));
    System.out.print(space);
}


void main()
{
    String s;
    String s2;
    s = "Datama";
    s2 = "t!";
    space = " ";

    int i;
    i = s.length();
    System.out.print( Integer.toString(i) );
    System.out.print(space);
```

```
        s = s.concat(s2);
        System.out.print(s);
        System.out.print(space);

        i = s.length();
        System.out.print( Integer.toString(i) );
        System.out.print(space);

        // print ascii value of t == 116
        char c;
        c = s.charAt(2);
        System.out.print( Integer.toString(c));
        System.out.print(space);

        strfalse = "F";
        strtrue = "T";



        f();

        B bptr;
        bptr = new B();

        g(2,1);

        bptr.g();
    }
}


class B
{
    void g()
    {
        String s;
        s = "hej fra objekt B";
        System.out.print(s);
    }
}
```

Assemblerkoden for ovenstående programkode fylder ikke mindre end 2100 liniers kode, hvilket ville svare til ca. 37 sider hvis den skulle bringes i rapporten. Udbyttet ville tillige være tvivlsomt efter koden er genereret efter helt faste principper og således gentager sig selv i de mange sætningskonstruktioner der er næsten identiske. Derfor bringes udelukkende resultatet af kodens afvikling (hvor ny linie er indsat før test af `while`)

```
6 Datamat! 8 116 6 8 236578 7  TF  TF  TF  TTF  FT  FFT  TFT
0 01234  321 hej fra objekt B
```

Sammenstilles resultatet med ovenstående eksempel, ses det hurtigt, at generatoren producerer den rigtige kode. Specielt ved de logiske operatorer og `while` sætninger, er det vigtigt at teste mere end et tilfælde.

Principielt mangler der en test af `if`, men efter at have noteret alle ovenstående resultater som korrekte er det rimeligt at antage, at også kodegenereringen for `if` fungerer.

Visse test fremgår ikke eksplicit, hvorfor de kort opridses her: Variablene `space, i` er henholdsvis fra klasse og funktions scope. Metoderne `f(), g()` repræsenterer henholdsvis metoder med 0 og multible argumenter. Metoden `B.g()` repræsenterer kald af metode i andet objekt.

10.4.1 `Sytem.in.read(), System.exit()`

Den følgende kode venter på brugeren taster en tast, og skriver derefter ASCII værdien af tasten ud på skærmen. Ved at give input "A", udskrev programmet `65` på skærmen. Ifølge opslagt i en ASCII tabel, er dette korrekt. Tallet blev kun udskrevet en gang på skærmen, hvilket viser, `System.exit()` metoden ligeledes fungerer.

```
class InReadTest
{
    static void main()
    {
        char c;
        c = System.in.read();

        // print ascii value
        System.out.print( Integer.toString(c) );
        Syste.exit();
        System.out.print( Integer.toString(c) );
    }
}
```

10.5 Andre test

`while`-løkker implementeres således, at i deres slutning indsættes en kendt label, der hoppes til, hvis udtrykket er falsk og whileløkken skal stoppes, eller der mødes en `break`. I princippet er dette uproblematisk. Men i assembler i "protected mode", er der en grænse for hvor langt et hop må være. Bliver der derfor for langt mellem hop og label, melder assembleren fejl. En sådan fejl opnås hvis følgende lille program forsøges assembleres. TASM skriver

```
**Error** foobar.asm(213) Relative jump out of range by 003Dh bytes
Error messages:    1
Warning messages:  None
Passes:            1
Remaining memory:  411k
```

EKSEMPEL #10.6:

```
class A
{
    void main()
    {
        String space;
        space = "*";
        String no;

        int p;
        int div;
        p = 1;

        while(p < 100)
        {
            div = 2;

            while(div != p && p%div != 0)
            {
                div = div + 1;
            }

            if(div == p)
            {
                no = toString(p);
                System.out.print(no);
                System.out.print(space);
            }
            else{}

            p = p + 1;
        }
    }
}
```

Lignende begrænsninger findes i `if` hvis "else" blokken ligger for langt fra starten af if, eller hvis funktioner er for stor (så label'en der skal hoppes til hvis `return` mødes er for langt fra hoppet.)

Man kunne argumentere, at compileren ikke formår at producere funktionel kode, når selv et så lille program som ovenstående eksempel ikke virker. Der er dog flere løsningsforslag til problemet.

- Man kunne lave en fase der talte kommandoer fra hop til label, og automatisk delte hoppet op i mindre og dermed tilladelige hop.
- Compileren kunne producere kode til en anden memory mode — med tab af DOS' standardfunktioner såsom udskrift til skærm eller læsning af tastatur.

Ingen af løsningerne er specielt elegante. løsningen er, at opdele funktioner løkker mv. i mindre dele. Ovenstående eksempel kunne opsplittes i

EKSEMPEL #10.7:

```
.
.
    while(p < 100)
```

```
{
    div = 2;
    f();
.
.
}

f()
{
    while(div != p && p%div != 0)
    {
    .
    .
```

Dette er f.eks. gjort i kapitel 11.1.

10.6 Hobens begrænsninger

Hobens begrænsninger blev også afmålt. Dette foregik med et program der genererede tal og udskrev disse på skærmen (hvorved String-objekter blev allokeret). Resultatet var at hoben ikke kan sættes til en vilkårlig stor størrelse. Trods testmaskinen indeholdt 96 MB ram, kunne hobstørrelsen ikke overskride 65210 bytes. Blev hobens størrelse forøget til 65212 bytes gav TASM's linker følgende fejlmeddelelse.

```
Turbo Link  Version 3.01 Copyright (c) 1987, 1990 Borland International
Error: Invalid initial stack offset
```

Sættes hoben størrelse endnu højere, f.eks. 65810 gav TASM følgende fejl

```
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland International

Assembling file:   prime.asm
*Warning* foobar.asm(8) Location counter overflow
**Error** foobar.asm(28) Constant too large
Error messages:    1
Warning messages:  1
Passes:            1
Remaining memory:  410k
```

Sidstnævnte skyldes assembleringen danner 16 bit kode.

Anvendes en hob på 65210 bytes kunne programmet udskrive alle tal fra 0 til 3480. Da hvert tal på grund af strengrepræsentationen fylder 2+(2 · tallets cifre) bytes i hoben bliver dette
$2581 \cdot 10 + 900 \cdot 8 + 90 \cdot 6 + 10 \cdot 4 = 37190$ bytes. Dette tal ligger langt fra de 65210, der må altså foregå nogle ukendte mekanismer når assemblerkoden udføres. Trods denne lidt bekymrende beregning, er det faktum, at der blev skabt og anvendt næsten 3500 objekter, hvilket er rigeligt til små programmer.

10.7 konklusioner af test

- Hoben kan kun antage en begrænset størrelse, dog rigeligt stor til de små programmer, kravsspecifikationen foreskriver.
- Assemblerprogrammerne opsættes til at blive afviklet i såkaldt "real mode", hvilket restrikterer længden af hvor langt der kan hoppes, og dermed størrelse af `if, while`, og funktioner. Disse problemer kan man dog programmere sig ud af.

— Kapitel

11 Hastighedstest

Fokus i dette projekt er ikke, om der kan genereres hurtigere kode end JDK, trods tesen der igangsatte projektet var, at selv håbløs assembler ville være meget hurtigere end JDK. Der er derfor ikke anvendt nævneværdige ressourcer på hverken hastighedsmålinger eller udfærdigelse af forskellig type testkode. Udgangspunktet for hastighedstestene er et program der finder alle primtal mellem 3–32000 på den værst tænkelige måde. Man kan næppe udfra dette drage konklusioner af større omfang. Der er mange faktorer der ikke er taget højde for, f.eks. hvor meget betyder det at programmet er på få linier kode? Hvor meget betyder alle funktionskaldene? Hvor meget betyder det at Qjava kører 16 bit, mens JDK kører 32 bit? Disse spørgsmål vil fortsat være ubesvarede.

Testmaskinen er en K6-200 mhz MMX med 96 MB RAM og Windows 98. Den anvendte JDK er v1.3b og TASM 3.1 til assemblering af Qjava koden.

Første gang et java program udføres kopieres nogle generelle bibloteker til memory, hvilket skulle gøre opstarten hurtigere ved flere kørsler. Til JDK 1.3b har man koncentreret sig om at reducere opstartstiden. På testmaskinen blev opstartstiden målt til ca. 1 sekundt.

I testkørsler klarede JDK testen på 10 sekunder, altså ca. 9 sekunder uden opstart. For Qjava var resultatet noget overraskende 11 sekunder!

Da den genererede assembler producerede åbenlys uproduktiv kode, blev output fra Qjava optimeret i hånden udfra følgende to principper

Princip 1

Mødes koden (hvor `#` noterer et register)

```
push #1
...
pop #1
```

kan koden helt fjernes, hvor det forudsættes, at "..." ikke refererer `#1`

Princip 2

Mødes koden (hvor `X` noterer en værdi)

```
mov #1, X
push #1
...
pop #2
```

hvor "..." ikke tilgår `#1` eller `#2` kan det optimeres til

```
mov #2, X
...
```

Følgende eksempel

EKSEMPEL #11.1:

```
mov #1, X
push #1
mov #1, X2
push #1
pop #2
pop #1
```

kan ved brug af disse to principper reduceres til

først (princip 2)

```
mov #1, X
push #1
mov #2, X2
pop #1
```

derefter (princip 1)

```
mov #1, X
mov #2, X2
```

Eksemplet er ikke tilfældigt valgt. Kodeblokken anvendes hvergang der skal foretages en binær eller dual operation, f.eks. addition eller sammenligning.

Resultatet med optimering efter princip 1 gav en køretid på 10 sekunder. Da princip 2 også blev taget i anvendelse, blev køretiden 8 sekunder. Qjava blev

altså hurtigere end JDK. Slutteligt blev koden optimeret yderligere manuelt ved at fjerne de boolske repræsentationer, og anvende `inc` istedet for "`add #, 1`" og andre småting, men hvor strukturen i oversættelsen blev bibeholdt. Dette gav en køretid på 6 sekunder.

Ved anvendelse af lettere optimering, kan anvendelse af Qjava give mindre tidsbesparelse. Dog viser testen at besparelsen aldrig bliver i størrelsesordenen en faktor og derfor er til at overse.

11.1 Primtalstest.java

Nedenstående kode blev anvendt til hastighedstest. Da DOS ikke kan håndtere en while-løkke i en while-løkke sådan som dette genereres i Qjava, løses problemet med metoden `loop()`. For ikke udskrivningsalgoritmerne for skærmudskrivning skulle have betydning, udskrives der ikke i programmet.

```
class Prime
{
    static void main()
    {
        Prime ptr;
        ptr = new Prime();

        ptr.loop();
    }


    void loop()
    {
        int p;
        p = 3;

        while(p < 32001)
        {
            isPrime(p);
            p = p + 1;
        }
    }


    void isPrime(int p)
    {
        int i;
        i = 2;

        while(i < p)
        {
            if(p%i == 0)
            {
                return;
            }
            else
            {
```

```
                i = i + 1;
            }
        }


        if(i == p)
        {   // we've got ourselves a prime!
        }
        else{}
    }
}
```

— Kapitel

12 Perspektivering

Perspektiveringen omhandler følgende tre emner

- Eksempler på bedre kodegenerering, sektion 12.1 side 88
- Hvorledes Qjava kunne udvides, sektion 12.2 side 90
- Erfaringer med kørsler, hvor tesen fra indledningen diskuteres, sektion 12.3 side 91

Oversigt —

I dette kapitel diskuteres forskellige forbedringsmuligheder for forholdsmæssigt lette ting at implementere. Kapitlet skal ikke forstås som "Qjava compileren BURDE indeholde alle disse forslag, og fordi den ikke gør det er den dårlig". Kapitlet prøver at perspektivere elementer i Java, der kan implementeres inden for overskuelig tid. Det faktum, at forslagene er let-implementérbare, bør istedet være et argument for, at den realisering der har fundet sted, ikke binder sig snævert til den givne kravsspecifikation, men at den kan være fundament for mere omfangsrige versioner af Qjava.

12.1 Kodegenerering

I dette afsnit foreslås forbedringer med hensyn til koden der genereres.

12.1.1 32 bit

Fra Intel 80386 processoren kan registrene indeholde 32 bit fremfor 16. Antallet og navnene på registrene er fortsat bevaret (pånær et "e" foran registernavnene). Registret `ax` tilgås altså `eax` hvis alle 32 bit ønskes tilgået.

Foruden denne minimale ændring skal der højst sandsynligt genereres en lidt anderledes opstartskode, samt registrenes funktionalitet skal undersøges.

En 32 bit compiler vil betyde, at talområdet udvides til $\pm 2^{31} = \pm 2,1 \cdot 10^9$. Samtidig med programmerne vil kunne behandle større tal, vil referencer kunne referere over et større område, og dermed tillade en større hob.

12.1.2 Statisk garbage collection

En "fuldblods" garbage collection er en stor opgave at implementere. Langt mere overskueligt er det, at lave en statisk garbage collection. Det vil sige, at compileren fjerner det øverste element på hoben, hvis den ved, at instansen ikke længere anvendes.

Dette vil reducere lagerforbruget specielt for `Integer.toString()` metoden, når den anvendes i forbindelse med udskrift. Det næste eksempel viser et tilfælde hvor hoben hurtigt fyldes.

EKSEMPEL #12.1:

```
int i;
.
.
i = 0;
while(i < 10000)
{
    System.out.print( Integer.toString(i) );
    .
    .
}
```

`i` er måske over-forsimplet, men i mange beregningsmæssige sammenhænge udskrives mellemresultater og resultater til skærmen, hvorefter de igennem resten af programmet udelukkende anvendes som tal.

I forbindelse med `System.out.print(Integer.toString(i))` kald, kunne compileren bagefter kalde en deallokeringsmetode. Denne kunne udformes som en standardfunktion ved navn `denew` som blot tæller `hobptr` ned med værdien i register cx, altså det omvendte af `new`.

EKSEMPEL #12.2:

```
denew PROC
    sub  hobptr, cx
    ret
denew ENDP
```

12.1.3 Lager

Istedet for hoben indeholdt repræsentationer af instanser af objekter, kunne den istedet indeholde adresserne på hvor dataene befinder sig i memory, Hoben bliver altså en reference-tabel over objekternes 'this'. På den måde undgås anvendelsen en fast mængde lager. Før implementationen for alvor kan være en forbedrign bør hele compileren køre i 32 bit, så en større del af lageret kan nås.

12.2 Sprogudvidelse

Qjava kunne på en række udvides så det bliver mere modulært og objektorienteret.

12.2.1 Access modifiers

For indkapsling af kode kan blive en realitet, mangler der Qjava muligheden for at angive hvem der må tilgå et objekts felter og metoder. Implementeringen af `public, private, final` kan foregå ved, at der til hver funktion og variabel knyttes et flag der angiver adgangen til denne. Dette kunne let administreres i symboltabellen, der blot skal udvide sin `Symbol` klasse, og hvor symboltabellens metoder skal udvides med de nye flag. Flagene kunne repræsenteres ved bit. Første bit kunne angive true/false `final`, anden bit true/false `private` og tredie bit true/false `public`. Flaget 5 (`0000000000000101b`) angiver dermed `public final`.

Det eneste der skal ændres er kodegenereringen for metodekald, der nu først slår op i symboltabellen før et eventuelt metodekald genereres.

12.2.2 Konstruktør

Konstruktører skal genereres som alle andre metoder til klasser, med det forbehold, at specificeres ingen konstruktør i en klasse genereres automatisk en tom procedure i assemblerkoden automatisk. Konstruktøren skal automatisk `call`'es efterfølgende genereringen af `new` kald.

Ønskes ⟨vardef⟩ udbygget til også at acceptere værditildeling, vil implementationen af konstruktøren forsimple dette problem drastisk. Koden for værditildelingen skal blot placeres der hvor konstruktørens kode placeres.

12.2.3 `Static` variable og metoder

Ved implementationen af `static` klassevariable og metoder kunne et flag til hver funktion/variabel anvendes til angivelse af true/false `static`. Hvis både access modifiers og `static` implementeres samtidig kunne true/false `static` placeres på 6 bit (således 2 bit er reserveret til henholdsvis `package` og `protected`, så accessmodifiers ligger i en samlet klump i flag-variablen).

Klassevariable (`static` variable) kan ikke placeres på hoben, da de eksisterer udenfor instanser af klassen. Der er derfor ingen naturlige steder en reference til placeringen på hoben kan lagres. Til gengæld har klassevariable samme egenskaber som globale variable i sprog som C/C++. De kan derfor placeres i DATA segmentet på følgende vis

```
Klassenavn.variabelnavn  DW 0
```

Hvis der under kodegenereringen tilgås en variabel med `static` bitten tændt, kan værdien af variablen tilgås med `variabelnavn`, f.eks. `mov ax, variabelnavn`. Grunden til der står "variabelnavn" og klassenavnet er udeladt skyldet, at klasse-

navn og variabelnavn under den leksikalske analyse er genkendt som et ⟨name⟩ og derfor er et ord. Ved at navngive variablen med "." i DATA segmentet, kan variabelnavnet anvendes direkte under kodegenereringen uden en navnekonvertering skal finde sted.

For metoder er implementeringen endnu lettere. Metoder genereres allerede automatisk i kodegenereringsfasen. Det eneste der skal implementeres er kontrol ved funktionskald, som hvis `static` bitten er tændt konverterer alle "." til `_` i navnet på metodekaldet, og ellers udfører funktionskaldet som alle andre metodekald.

12.3 Erfaringer fra kørsel

Udfra erfaringerne med kørsler at dømme, må tesen fremlagt i indledningen konkluderes at være falsk. Omend der ikke skulle voldsomme optimeringer ind i billedet før native koden var hurtigere, er Java virtual machines idag så langt fremme med runtime compilering, at det ligger inden for rækkevidden af native kode.

Det vides ikke om JDK i realiteten compilerede primtalsprogrammet én gang og udførte det fra en "optimerings cache" lignende ting eller ej. Men faktum er, at hvad enten JDK udfører ved små programmer, er det mere effektivt end uoptimeret genereret assemblerkode. Fremtiden for Qjava compileren kan idag derfor ligge på et meget lille sted! Havde Qjava compileren været konstrueret for få år tilbage, ja så sent som i 1998, ville resultaterne ganske sikkert være helt anderledes. Inden for årene 1997–1999, er der sket en mangedobling i afviklingshastigheden af Java programmer.

Bytecode giver ydermere mulighed for ting der er meget svært realiserbare i assembler. I Java giver det mening, at udføre et program, der ikke "kender sig selv", forstået på den måde, at de funktionaliteter der ikke forefindes i den aktuelle kode kan hentes fra ekstern kilde f.eks. harddisk eller internettet. Et tekstbehandlingsprogram kunne derfor udelukkende bestå af en menu og opbygning af et vindue. De restende funktionaliteter kan så hentes når de skal anvendes. På denne måde kan store programmer opnå hurtigere opstartstid end de samme programmer skrevet i assembler! Den sidste store forskel der hér skal nævnes er bytecodes egenskab ved at være kørbar på alle større platforme vel og mærke uden ændringer finder sted i koden. Dette er en umulighed for assembler, for selvom assembleren genereres så generel som muligt vil den altid være stærkt knyttet til den pågældende arkitektur.

Vel nok især det sidste punkt gør, at flere nye programmeringssprog ikke skabes med assembler og linker, men istedet producerer bytecode. Udviklerne har på denne måde for det første sparet en masse arbejde der skulle gå til udvikling af assembler, linker og debug'er, men vigtigere, antallet af potentielle bruger har stort set nået sit maksimum fra starten. Det kan tænkes, at bytecode-teknologien kan vælte Microsofts monopolagtige possition inden for styresystemer, da valg af styresystem om få år sikkert ingen betydning har for udbud af software. Paradoksalt er det, at dette ligeledes nivellerer betydning af hvilket programmeringssprog der anvendes — sålænge dette sprog genererer bytecode. Det kan derfor, ihvertfald i

princippet, tænkes, at et nyt programmeringssprog i fremtiden, formår at blive mere anvendt end Java selv på grund af "Write once, run anywhere"-idéen.

— Kapitel

13 Konklusion

Igennem kapitlerne 3–8 er Qjava sproget og Qjava compileren blevet konstrueret efter retningslinierne i kapitel 2. Resultatet er blevet en velfungerende compiler, der genererer assemblerkode, der direkte kan assembleres og afvikles. Qjava compileren opfylder fuldt ud kravsspecifikationen givet i sektion 1.1 side 2 og undersektioner.

Det eneste der kan synes problematisk er begrænsningerne for `if, while` og metoders størrelse. Disse problemer skal dog ene og alene tilskrives assemblerspecifikke og Microsoft DOS specifikke problemer, og er ikke "oversætterteoretiske problemer". I samme boldgade opstod der mindre problemer med hvilke registre der kunne udføre hvilke funktioner.

I kapitel 11 side 84 blev tesen fra indledningen om at uoptimeret native assemblerkode er væsenligt hurtigere end afviklingen af bytecode falsificeret. Det viste sig, at for et lille, men beregningsmæssigt tungt, program var afviklingshastigheden kortere for JDK. JDK var 9 sekunder om at afvikle programmet, mens assemblerkoden Qjava compileren producerede, var 11 sekunder om samme job. Ved hjælp af to simple optimeringsprincipper angivet, i kapitel 11, blev køretiden for assemblerkoden dog reduceret til 8 sekunder. Og yderligere 2 sekunder blev vundet ved gennemførelse af mere vanskelig optimering. De henholdsvis 6 og 8 sekunder er hurtigere end JDK, men som tiderne viser, bliver tidsgevinsten ved anvendelse af native assembler nok aldrig en størrelsesorden eller i nærheden af hvad der på forhånd var forventet.

Omvendt er dette budskab for Java programmører utrolig possitivt, da teser som den indledningen præsenterede kan manes i jorden. Rapporten kunne måske ligefrem være med til at skabe en holdningsændring hos de mennesker der stadig mener Java er et langsomt sprog — i hvertfald når diskusionen omhandler mindre programmer.

> *Det må derfor konkluderes, at anvendeligheden af Qjava compileren, set i et tidsbesparelses-perspektiv, ikke er specielt stor. Der skal implementere meget effektive optimeringsalgoritmer, hvis målsætningen med Qjava compileren skal opfyldes.*

Qjava compileren producerer assemblerkode med kommentarer, hvilket kan gøre compileren anvendelig for ikke tilsigtede målgrupper. Disse målgrupper kunne være kursister ved oversætterteknik-kurser, eller kurser i Intel 8086 assembler, da det er muligt at følge Qjava programmernes gang igennem kommentarerne i assemblerkoden.

I forhold til modul 1 på datalogi/RUC målsætning om at lære Java og forstå sprogets sammenhænge, passer dette projekt godt, da det netop er sproget og ikke kodegenereringen, der er fokus. Projektet er endvidere et godt oplæg til et modul 2 projekt, der kunne udbygge compileren med semantik kontrol, advanceret kodegenerering, lager-administration og kodeoptimering.

Litteratur

[Aho et al;1986] Hoe, Alfred V. & Ravi Sethi & Jeffrey D. Ullman,
"Compilers — Principles, techniques, and tools", 1986 Addison-Wesley publishing company

[Gamma et al;1995] Gamma, Erich & Richard Helm & Ralph Johnson & John Vlissides & Grady Booch (Designer),
"Design Patterns : Elements of Reusable Object-Oriented Software", 1. ed, 1995 Addison-Wesley publishing company, ISBN: 0201633612

[Hansen;1995] Hansen, H. B., *"Programafprøvning Pascal Version"*,
Roskilde Universitetscenter 1995, ISSN: 0908–5491

[Appel;1998] Appel, Andrew W.,
"Modern compiler implementation in java",
Cambridge University press, Melbourne Australia 1998, ISBN: 0–521–58388–8

[Arnold et al;1997] Arnold, Ken & James Gosling,
"The java^{TM}programming language",
Addison-Wesley publishing company, 1997, ISBN 0–201–63455–4

— Bilag —

Qjava A

```
public class Qjava
{
   static void main(String[] args)
    {
        CompilerFacade cf;

        if(args.length == 1)
            cf = new CompilerFacade(args[0]+".java", args[0]+".asm");
        else
            System.out.println("Qjava v1.0 by Kasper B. Graversen (c) 1999 - This is freeware\n
                                Usage: Qjava <filename without extension>");
    }
}
```

— Bilag —

B CompilerFacade

```
import java.io.*;

public class CompilerFacade
{
    CompilerFacade(String input, String output)
    {
        try
        {
            // lex the code
            Lexer lexer = new Lexer(input);

            SymbolTable symboltable = new SymbolTable();

            // build a parsertree and SymbolTable
            Parser parser = new Parser(lexer, symboltable);
            Tree parsetree = parser.parse();

            // generate code
            CodeGenerator codegenerator = new CodeGenerator(symboltable);
            String asmSource = codegenerator.generate(parsetree);

            // write assembler code
            FileWriter fp = new FileWriter(output);
            PrintWriter oup = new PrintWriter(fp);
            oup.println(asmSource);
            oup.close();
        }
        catch(FileNotFoundException e)
        {   System.out.print("Inputfile not found!\n" + e); }

        catch(IOException e)
        {   System.out.print("IO error!\n" + e);}

        catch(Exception e)
        {   System.out.print("Internal error!\n" + e);}
    }
}
```

C kode Lexer

```
import java.io.*;


public class Lexer extends StreamTokenizer implements TokenNames
{

    boolean pushBackFlag = false; // used in pushBack()
    Token T;                      // used to make pushBack() method

// Constructor
Lexer(String fname) throws FileNotFoundException
{
    super( new FileReader(fname) );

    // setup StreamTokenizer
    slashSlashComments(true);
    slashStarComments(true);
    whitespaceChars((int) '\t', (int) '\t');
    whitespaceChars((int) ' ',  (int) ' ');
    ordinaryChar((int) '/');         // or ``/'' is a comment
    ordinaryChar((int) '.');         // or ``.'' is a number
    ordinaryChar((int) '-');         // so StreamTokenizer gives <-><4> instead of <-4>
    wordChars((int) '_',(int) '_');  // or ``_'' works as a sepperator
}


public Token getNextToken() throws IOException
{
    // T == null if pushBack() is issued before getNextToken()
    // (first call could be a pushback call)
    if(pushBackFlag == true && T != null)
    {
        pushBackFlag = false;
        return(T);
    }

    int read = super.nextToken();

    switch(read)
```

```
    {
        case TT_WORD:
            if(sval.equals("break")) {T = new Token(BREAK, lineno()); break;}
            if(sval.equals("class")) {T = new Token(CLASS, lineno()); break;}
            if(sval.equals("else"))  {T = new Token(ELSE,  lineno()); break;}
            if(sval.equals("if"))    {T = new Token(IF, lineno());    break;}
            if(sval.equals("new"))   {T = new Token(NEW, lineno());   break;}
            if(sval.equals("null"))  {T = new Token(NULL, lineno());  break;}
            if(sval.equals("return")){T = new Token(RETURN, lineno());break;}
            if(sval.equals("void"))  {T = new Token(VOID, lineno());  break;}
            if(sval.equals("while")) {T = new Token(WHILE, lineno()); break;}

            // Error if token isn't supported get next token
            if(unsupportedToken() == true)
                return(getNextToken());

            // extract <ID> or <NAME>
            T = extractIdOrName();
        break;


        case TT_NUMBER:
            T = new Token(VAL_INT, (int) nval, lineno());
        break;


        default:
            if(read == TT_EOF) {T = new Token(EOF, lineno()); break;}

            if(isIllegalChar((char) read) == false)
            {
                switch((char) read)
                {
                    // char length must be 0 or 1
                    case '\'': if(sval.length() <= 1) T = new Token(VAL_CHAR, sval, lineno());
                               else errorAndExit("Error", "Char definition more than 1 character long.",
                               lineno());
                               break;

                    case '\"': T = new Token(VAL_STRING, sval, lineno()); break;

                    // These operators can be a compound of a longer operator.
                    case '&':
                    case '|':
                    case '<':
                    case '!':
                    case '=': T = fetchOperator((char) read); break;

                    default: T = c2Token((char)read); break;
                }
            }
            else
                errorAndExit("Error", "Illegal character \"" + read + "\"", lineno());
    }

    return(T);
}
```

```
// sets a flag so next time getNextToken() is issued it will return last read token
// instead of reading a new token.
public void pushBack()
{
    pushBackFlag = true;
}



private Token extractIdOrName() throws IOException
{
    int state = 0;                          // for the statemachine
    StringBuffer buf = new StringBuffer();
    buf.append(sval);                       // save sval before we read on

    int nt = super.nextToken();

    if((char) nt == '.')
    {
        buf.append("."); state = 1;
    }

    while(true)
    {
        // statemachine
        switch(state)
        {
        // stop
        case 0: super.pushBack(); return(new Token(ID, buf.toString(), lineno()) );

        // after "." is read, and next is <ID> then <NAME> is found -- otherwise return <ID>
        case 1: nt = super.nextToken();
                if(nt == TT_WORD)
                {
                    buf.append(sval); state = 2;
                }
                else
                    state = 0;
                break;

        // <NAME> is found, now look for { <.> <ID> }
        case 2: nt = super.nextToken();
                if((char) nt == '.')
                {
                    buf.append("."); state = 3;
                }
                else state = 9; // return <NAME>
                break;

        case 3: nt = super.nextToken();
                if(nt == TT_WORD)
                {
                    buf.append(sval); state = 2;
                }
                else state = 9; // return <NAME>
                break;
```

```
        // return <NAME>
        case 9: super.pushBack(); return( new Token(NAME, buf.toString(), lineno()) );

        } // EOswitch()
    } // EOwhile()
}


// converts a character to a token.
private Token c2Token(char c)
{
    switch(c)
    {
        case ',': return(new Token(COMMA, lineno()) );
        case '.': return(new Token(DOT, lineno()) );
        case ';': return(new Token(SEMICOLON, lineno()) );
        case ':': return(new Token(COLON, lineno()) );
        case '(': return(new Token(LPAR, lineno()) );
        case ')': return(new Token(RPAR, lineno()) );
        case '{': return(new Token(LBRACE, lineno()) );
        case '}': return(new Token(RBRACE, lineno()) );
        case '[': return(new Token(LSQBRACE, lineno()) );
        case ']': return(new Token(RSQBRACE, lineno()) );

        case '+': return(new Token(PLUS, lineno()) );
        case '-': return(new Token(MINUS, lineno()) );
        case '*': return(new Token(MULT, lineno()) );
        case '/': return(new Token(DIV, lineno()) );
        case '%': return(new Token(MODULO, lineno()) );
        case '<': return(new Token(LESS, lineno()) );
        case '=': return(new Token(SET, lineno()) );

        case '&': return(new Token(BITAND, lineno()) );
        case '|': return(new Token(BITOR, lineno()) );
        case '!': return(new Token(NOT, lineno()) );

        default: errorAndExit("Error", "Unsuported character \""+ c + "\" found.", lineno());
    }

    // We will never get here, but Java wants the line...
    return( new Token(COMMA, lineno()));
}

// check if character is illegal
private boolean isIllegalChar(char c)
{
    if(c == '@' || c == '#' || c == '$' || c == '§' || c == '½' || c == '~')
        return(true);
    else
        return(false);
}


// check if keyword is unsupported by the compiler
private boolean unsupportedToken()
{
    boolean illegal = false;
```

```
    if(sval.equals("case"))     illegal = true; else
    if(sval.equals("catch"))    illegal = true; else
    if(sval.equals("continue")) illegal = true; else
    if(sval.equals("double"))   illegal = true; else
    if(sval.equals("extends"))  illegal = true; else
    if(sval.equals("final"))    illegal = true; else
    if(sval.equals("float"))    illegal = true; else
    if(sval.equals("for"))      illegal = true; else
    if(sval.equals("goto"))     illegal = true; else
    if(sval.equals("interface"))illegal = true; else
    if(sval.equals("package"))  illegal = true; else
    if(sval.equals("switch"))   illegal = true; else
    if(sval.equals("throw"))    illegal = true; else
    if(sval.equals("try"))      illegal = true; else
    if(sval.equals("static") || sval.equals("public") ||
      sval.equals("private") || sval.equals("protected") )
    {
        System.out.println("Warning("+lineno()+"): \""+sval+"\" is not supported.");
        return(true);
    }


    if(illegal)
        errorAndExit("Error", "keyword \"" + sval + "\" is not yet supported.", lineno());

    return(false);
}



// check if token is a compound of another token, ie < is a compound of <=
private Token fetchOperator(char c) throws IOException
{
    int nt = super.nextToken();

    // the operator is a single operator
    if(nt != TT_WORD && nt != TT_NUMBER && nt != TT_EOF)
    {
        String s = "" + c + (char) nt;
        if(s.equals("!=")) return( new Token(NEQUAL,lineno()) );
        if(s.equals("==")) return( new Token(EQUAL, lineno()) );
        if(s.equals("<=")) return( new Token(LEQUAL,lineno()) );
        if(s.equals("&&")) return( new Token(AND,   lineno()) );
        if(s.equals("||")) return( new Token(OR,    lineno()) );

        // Warn of un-implemented operators
        if(s.equals("+=") || s.equals("-=") || s.equals("*=") || s.equals("/=") ||
           s.equals("++") || s.equals("--") || s.equals(">="))
            errorAndExit("Error", "Keyword \"" + s + "\" is not supported!", lineno());

        // Token was not a compound token, so we need to pushback last read token.
        super.pushBack();
        return(c2Token(c));
    }
    else
    {
        // The second token was not an operator, so push it back
        // and return the operator as a token
        super.pushBack();
```

```
        return(c2Token(c));
    }
}


private void errorAndExit(String s, String s2, int lineno)
{
    System.out.println(s+"("+lineno+"): "+s2);
    System.exit(0);
}


/*
//main exists only for testpurposes - note output is in LaTeX format
static void main(String[] args)
{
    try
    {
        int taeller = 0;
        Lexer read = new Lexer("d:\\modul1\\proj\\lextester.txt");
        Token t;

        do {
            t = read.getNextToken();
            System.out.print(taeller++ + " & ");

            switch(t.id)
            {
                case COMMA:    System.out.print(","); break;
                case DOT:    System.out.print("."); break;
                case SEMICOLON:System.out.print(";"); break;
                case COLON:    System.out.print(":"); break;
                case LPAR:    System.out.print("("); break;
                case RPAR:    System.out.print(")"); break;
                case LBRACE:System.out.print("\\{"); break;
                case RBRACE:System.out.print("\\}"); break;
                case LSQBRACE:System.out.print("["); break;
                case RSQBRACE:System.out.print("]"); break;
                case PLUS:    System.out.print("$+$"); break;
                case MINUS:    System.out.print("$-$"); break;
                case MULT:    System.out.print("*"); break;
                case DIV:    System.out.print("/"); break;
                case MODULO:System.out.print("\\%"); break;
                case LESS:    System.out.print("$<$"); break;
                case SET:    System.out.print("="); break;
                case EQUAL:    System.out.print("=="); break;
                case NEQUAL:System.out.print("!="); break;
                case LEQUAL:System.out.print("<="); break;
                case AND:    System.out.print("\\&\\&"); break;
                case OR:    System.out.print("$||$"); break;
                case BITAND:System.out.print("\\&"); break;
                case BITOR:    System.out.print("$|$"); break;
                case NOT:    System.out.print("!"); break;
                case BREAK:    System.out.print("break"); break;
                case ELSE:    System.out.print("else"); break;
                case IF:    System.out.print("if"); break;
                case WHILE:    System.out.print("while"); break;
                case CLASS:    System.out.print("class"); break;
                case NEW:    System.out.print("new"); break;
```

```
                case RETURN:System.out.print("return"); break;
                case ID:    System.out.print("id ("+ t.sval +")"); break;
                case NAME:    System.out.print("name ("+ t.sval +")"); break;

                case VOID:    System.out.print("void"); break;
                case NULL:    System.out.print("null"); break;

                case VAL_CHAR:    System.out.print("chVal('"+ t.sval +"')"); break;
                case VAL_INT:    System.out.print("intVal("+ t.nval +")"); break;
                case VAL_STRING:System.out.print("StrVal("+ t.sval +")"); break;
                case EOF:        System.out.print("<EOF>"); break;

                default: System.out.print("OUPS!! cant decode "+t.id + " id!\n");
            }


            if(taeller%4 == 0)
                System.out.print("\\\\ \\hline \n");
            else
                System.out.print(" & ");
        } while(t.id != EOF);

        System.out.println("\\\\ \\hline \n");
    }
    catch(Exception e)
    {
        System.out.println(e);
        System.exit(0);
    }
}
*/

} // EOClass
```

— Bilag

D Token

```
public class Token
{
    /** Værdierne Token klassen kan indeholde */
    int id;
    int nval;
    String sval;
    int lineno;

    // Konstruktoren
    Token(int id, int lineno)
    { this.id = id; this.lineno = lineno;}

    Token(int id, int nval, int lineno)
    { this.id = id; this.nval = nval; this.lineno = lineno;}

    Token(int id, String sval, int lineno)
    { this.id = id; this.sval = sval; this.lineno = lineno;}

}
```

— Bilag

TokenNames

E

```
public interface TokenNames
{
    // Tegn
    int
    COMMA        =  1, // ,
    DOT          =  2, // .
    SEMICOLON    =  3, // ;
    COLON        =  4, // :
    LPAR         =  5, // (
    RPAR         =  6, // )
    LBRACE       =  7, // {
    RBRACE       =  8, // }
    LSQBRACE     =  9, // [
    RSQBRACE     = 10, // ]


    // Operatorer
    PLUS         = 50, // +
    MINUS        = 51, // -
    MULT         = 52, // *
    DIV          = 53, // /
    MODULO       = 54, // %
    LESS         = 55, // <
    SET          = 57, // =
    EQUAL        = 58, // ==
    NEQUAL       = 59, // !=
    LEQUAL       = 60, // <=
    AND          = 62, // &&
    OR           = 63, // ||
    BITAND       = 64, // &
    BITOR        = 65, // |
    NOT          = 66, // !


    // Kommandoer
    BREAK        = 80,
    ELSE         = 81,
    IF           = 83,
    WHILE        = 84,
```

```
    CLASS       = 85,
    NEW         = 86,
    RETURN      = 87,

    ID          = 100,
    NAME        = 101,


    // Andet
    VOID        = 169,
    NULL        = 170,
    VAL_CHAR    = 171,
    VAL_INT     = 172,
    VAL_STRING  = 173,
    EOF         = 200; // End Of File
}
```

— Bilag

F

Parser

```
import java.io.*;
import java.util.Vector;


public class Parser implements TokenNames
{
    private Lexer lexer;
    private boolean isEOF = false;
    private Token T;        // indeholder det aktuelle token


    // for constructing the SymbolTable (sbtb)
    private String sbtbFncname = null;
    private String sbtbClassname;
    private SymbolTable symbolTable;



// Constructor
Parser(Lexer lexer, SymbolTable symbolTable) throws IOException
{
    if(lexer != null)
    {
        this.lexer = lexer;
        T = lexer.getNextToken();   // læs første token
        this.symbolTable = symbolTable;
    }
    else
        errorAndExit("Lexer not available\n");
}


public Tree parse() throws IOException
{
    return( S() );// start parsing
}


private Tree S() throws IOException
```

```
{
    return( classdef () );
}


private Tree classdef() throws IOException
{
    Classdef tree = new Classdef(T.lineno);

    while(isEOF == false)
    {
        eat(CLASS);
        tree.add(T.sval);
        sbtbClassname = T.sval;

        eat(ID);                // verificer className er en <ID>
        eat(LBRACE);            // verificer "{"
        tree.add( classcontents() );
        eat(RBRACE);            // verificer "}"
    }

    return(tree);
}


private Tree classcontents() throws IOException
{
    Classcontents tree = new Classcontents(T.lineno);

    if(T.id != ID && T.id != VOID)
        errorAndExit("Error", "Expected <ID> or \"void\" got token #"+T.id, T.lineno);

    // vardef || fncdef
    while(T.id == ID || T.id == VOID || T.id == SEMICOLON)
    {
        if(T.id != SEMICOLON)
        {   // vardef
            if(T.id == ID)
                tree.add( vardef() );
            else
                 tree.add( fncdef() );
        }
        else
            eat(SEMICOLON);
    }

    return(tree);
}


private Tree vardef() throws IOException
{
    String type = T.sval;
    eat(ID);

    String name = T.sval;
    eat(ID);

    symbolTable.add(type, sbtbClassname, sbtbFncname, name);
```

```
    eat(SEMICOLON);
    return( new Vardef(type, name, T.lineno) );
}



private Tree fncdef() throws IOException
{
    Vector v = new Vector();

    eat(VOID);
    String name = T.sval;
    sbtbFncname = T.sval;
    eat(ID);
    eat(LPAR);

    // EBNF: [<ID> <ID> { "," <ID> <ID> } ]
    if(T.id == ID)
    {
        v.addElement(T.sval);
        eat(ID);

        v.addElement(T.sval);
        eat(ID);

        // EBNF: { "," <ID> <ID> }
        while(T.id == COMMA)
        {
            eat(COMMA);
            v.addElement(T.sval);
            eat(ID);
            v.addElement(T.sval);
            eat(ID);
        }
    }

    eat(RPAR);

    eat(LBRACE);
    Tree s = sentences();
    eat(RBRACE);

    sbtbFncname = null;
    return( new Fncdef(name, v, s, T.lineno) );
}



private Tree sentences() throws IOException
{
    Sentences tree = new Sentences(T.lineno);

    if(T.id != IF && T.id != WHILE && T.id != BREAK && T.id != RETURN && T.id != ID &&
       T.id != NAME && T.id != RBRACE && T.id != SEMICOLON)
        errorAndExit("Error", "Expecting either: \"if\", \"while\", \"break\", \"return\",
                     <ID>, <NAME>, \";\", or \"}\"", T.lineno);
```

```
    while(T.id == IF || T.id == WHILE || T.id == BREAK || T.id == RETURN ||
          T.id == ID || T.id == NAME || T.id == SEMICOLON)
    {
        switch(T.id)
        {
            case SEMICOLON: eat(SEMICOLON); break;
            case IF:    tree.add(if_());    break;
            case WHILE: tree.add(while_()); break;
            case BREAK: tree.add(break_()); break;
            case RETURN:tree.add(return_());break;
            case ID:
            case NAME: int lookAhead = peek(); // read next char
                switch(lookAhead)
                {
                    case -1:   errorAndExit("Error", "Unexpected End Of File", T.lineno);
                    case ID:   tree.add(vardef()); break;
                    case LPAR: tree.add(fnccall());break;
                    case SET:  tree.add(assign()); break;
                    default: errorAndExit("Error", "Expected <ID>, \")\" or \"=\"", T.lineno);
                }
                break;

            default:
                errorAndExit("Error", "Something beyond my comprehension took place...!", T.lineno);
        }
    }

    return(tree);
}



private Tree fnccall() throws IOException
{
    String name = T.sval;

    Fnccall tree;

    if(T.id == ID)
    {
        eat(ID);
        tree = new Fnccall(name, true, T.lineno); // local call
    }
    else
    {
        eat(NAME);
        tree = new Fnccall(name, false, T.lineno); // not local call
    }

    eat(LPAR);

    // EBNF: [ <E> { , <E> } ]
    if(T.id != RPAR)
    {
        tree.add( E() );

        while(T.id == COMMA)
        {
            eat(COMMA);
```

```
            tree.add( E() );
        }
    }

    eat(RPAR);
    eat(SEMICOLON);

    return(tree);
}


private Tree if_() throws IOException
{
    eat(IF);
    eat(LPAR);
    Tree condCode = E();
    eat(RPAR);

    eat(LBRACE);
    Tree thenCode = sentences();
    eat(RBRACE);

    eat(ELSE);
    eat(LBRACE);
    Tree elseCode = sentences();
    eat(RBRACE);

    return( new If(condCode, thenCode, elseCode, T.lineno) );
}



private Tree while_() throws IOException
{
    eat(WHILE);
    eat(LPAR);
    Tree condCode = E();
    eat(RPAR);
    eat(LBRACE);
    Tree whileCode = sentences();
    eat(RBRACE);

    return( new While(condCode, whileCode, T.lineno) );
}


private Tree break_() throws IOException
{
    eat(BREAK);
    eat(SEMICOLON);

    return( new Break(T.lineno) );
}


private Tree return_() throws IOException
{
    eat(RETURN);
```

```
    if(T.id == LPAR) { eat(LPAR); eat(RPAR);}
    eat(SEMICOLON);

    return( new Return(T.lineno) );
}


private Tree assign() throws IOException
{
    int op;
    String name = T.sval;

    if(T.id == ID)
    {   op = ID;
        eat(ID);
    }
    else
    {   op = NAME;
        eat(NAME);
    }

    eat(SET);
    Tree E = E();
    eat(SEMICOLON);

    return( new Assign(name, op, E, T.lineno) );
}


private Tree E() throws IOException
{
    Tree t = E1();

    while(T.id == OR)
    {
        eat(OR);
        OpDual tree = new OpDual(OR, t, E1(),T.lineno);
        t = tree;
    }

    return(t);
}

private Tree E1() throws IOException
{
    Tree t = E2();

    while(T.id == AND)
    {
        eat(AND);
        OpDual tree = new OpDual(AND, t, E2(),T.lineno);
        t = tree;
    }
    return(t);
}


private Tree E2() throws IOException
{
```

```
    Tree t = E3();

    while(T.id == BITOR)
    {
        eat(BITOR);
        OpDual tree = new OpDual(BITOR, t, E3(), T.lineno);
        t = tree;
    }
    return(t);
}


private Tree E3() throws IOException
{
    Tree t = E4();
    while(T.id == BITAND)
    {
        eat(BITAND);
        OpDual tree = new OpDual(BITAND, t, E4(), T.lineno);
        t = tree;
    }
    return(t);
}


private Tree E4() throws IOException
{
    Tree t = E5();

    while(T.id == EQUAL || T.id == NEQUAL)
    {
        int op;

        if(T.id == EQUAL)
        {
            eat(EQUAL); op = EQUAL;
        }
        else // !=
        {
            eat(NEQUAL); op = NEQUAL;
        }
        OpDual tree = new OpDual(op, t, E5(), T.lineno);
        t = tree;
    }

    return(t);
}



private Tree E5() throws IOException
{
    Tree t = E6();

    while(T.id == LESS || T.id == LEQUAL)
    {
        int op;

        if(T.id == LESS)
```

```
        {
            eat(LESS); op = LESS;
        }
        else // <=
        {
            eat(LEQUAL); op = LEQUAL;
        }
        OpDual tree = new OpDual(op, t, E6(), T.lineno);
        t = tree;
    }

    return(t);
}


private Tree E6() throws IOException
{
    Tree t = E7();

    while(T.id == PLUS || T.id == MINUS)
    {
        int op;

        if(T.id == PLUS)
        {
            eat(PLUS); op = PLUS;
        }
        else // -
        {
            eat(MINUS); op = MINUS;
        }
        OpDual tree = new OpDual(op, t, E7(), T.lineno);
        t = tree;
    }
    return(t);
}




private Tree E7() throws IOException
{
    Tree t = E8();

    while(T.id == MULT || T.id == DIV || T.id == MODULO)
    {
        int op = 0; // = 0 to make Java happy

        if(T.id == MULT)
        {
            eat(MULT); op = MULT;
        }

        if(T.id == DIV)
        {
            eat(DIV); op = DIV;
        }

        if(T.id == MODULO)
```

```
        {
            eat(MODULO); op = MODULO;
        }
        OpDual tree = new OpDual(op, t, E8(), T.lineno);

        t = tree;
    }

    return(t);
}


private Tree E8() throws IOException
{
    if(T.id == NEW)
    {
        eat(NEW);
        OpCall tree = new OpCall(NEW, T.sval, T.lineno);
        eat(ID);
        eat(LPAR); eat(RPAR);

        return(tree);
    }
    else
    {
        return( E9() );
    }
}


private Tree E9() throws IOException
{
    if(T.id == NOT || T.id == MINUS)
    {
        int op;

        if(T.id == NOT)
        {
            eat(NOT); op = NOT;
        }
        else
        {
            eat(MINUS); op = MINUS;
        }
        return( new OpMonadic(op, E10(),T.lineno) );
    }
    else
    {
        return( E10() );
    }
}


private Tree E10() throws IOException
{
    if(T.id != ID && T.id != NAME)
    {
        return( E11() );
    }
```

```
    else
    {
        OpCall tree;

        if(T.id == ID)
        {
            tree = new OpCall(ID, T.sval, T.lineno);
            eat(ID);
        }
        else
        {
            tree = new OpCall(NAME, T.sval, T.lineno);
            eat(NAME);
        }

        if(T.id == LPAR)
        {
            eat(LPAR);

            tree.setIsFncCall(); // this is a fnc call not a variable

            if(T.id != 6)
            {
                tree.add( E() );

                while(T.id == COMMA)
                {
                    eat(COMMA);
                    tree.add( E() );
                }
            }

            eat(RPAR);
        }

        return(tree);
    }
}


private Tree E11() throws IOException
{
    String s;
    int n;
    switch(T.id)
    {
        case VAL_STRING:
            s = T.sval; eat(VAL_STRING);
            return( new OpConst(VAL_STRING, s, T.lineno) );

        case VAL_CHAR:
            s = T.sval; eat(VAL_CHAR);
            return( new OpConst(VAL_CHAR, s, T.lineno) );

        case VAL_INT:
            n= T.nval; eat(VAL_INT);
            return( new OpConst(VAL_INT, n, T.lineno) );

        case LPAR:
```

```
            eat(LPAR);
            Tree t = E();
            eat(RPAR);
            return(t);

        // null is for now just treated as a zero
        case NULL:
            eat(NULL);
            return( new OpConst(VAL_INT, 0, T.lineno) );


        default:
            errorAndExit("\nError", "Unexpected token: " + T.id + " (" + T.sval + " " + T.nval + ")",
                        T.lineno);
    }
    return(null);// Only here because of Java<tm>
}



private int peek() throws IOException
{
    if(T.id == EOF)
        return(-1);

    Token tmp = lexer.getNextToken();
    lexer.pushBack();

    return(tmp.id);
}



private void eat(int id) throws IOException
{
    if(T.id != id)
        errorAndExit("Error", "Expected token #"+id+ " got #" + T.id, T.lineno);

    T = lexer.getNextToken();

    if(T.id == EOF)
        isEOF = true;
}


private void errorAndExit(String error)
{
    System.out.println(error);
    System.exit(0);
}


private void errorAndExit(String s, String s2, int l)
{
    System.out.println(s +"("+l+"): "+ s2);
    System.exit(0);
}
```

```
/* For testing only

public static void main(String s[])
{
    try
    {   SymbolTable myst = new SymbolTable();
        Parser a = new Parser(new Lexer("d:\\modul1\\proj\\parsertester.txt"), myst);

        Tree root = a.parse();

        //gennnemløb af parsetræet
        System.out.println("\n\ngennemgang\n----------\n");
        System.out.print(root.toString() );

        System.out.println("\nTest af symbolTable\n");
        System.out.println("A = " + myst.classSize("A"));
        System.out.println("B = " + myst.classSize("B"));

        System.out.println("B.c() = " + myst.fncSize("B","c"));
        System.out.println("B.c().x = " + myst.varIndex("B","c","x") );
        System.out.println("B.k = " + myst.varIndex("B", null, "k") );
        System.out.println("vartype i = " + myst.varType("A", "a", "i") );
        myst.add("char", "A", null, "c", 42);
        System.out.println("A.c = " + myst.varIndex("A", null, "c") );

    }
    catch(Exception e)
    {
        System.out.print("cought " +e);
    }
}
*/

} // EOC
```

G Tree

```
import java.util.Vector;

public abstract class Tree
{
    public abstract String toString();

    protected void errorAndExit(String s)
    {System.out.print(s); System.exit(0);}

    protected int lineno; // line # of element in source
    public int lineno(){return(lineno);}
}


class Classdef extends Tree
{
    private int getCounter = 0;
    private Vector classList = new Vector(); // touple af (navn, contents)

    // constructor
    Classdef(int lineno){super.lineno = lineno;}

    public void add(Object o){classList.addElement(o);}

    public void resetGet(){getCounter = 0;}

    public Object getNext()
    {
        if(getCounter > -1 && getCounter < classList.size())
            return(classList.elementAt(getCounter++));
        return(null);
    }

    public void pushBack(){getCounter--;}

    public String toString()
    {
        StringBuffer buf = new StringBuffer();
```

```
        for(int i = 0; i < classList.size(); i++)
            buf.append("class " + (String) classList.elementAt(i++) + "\n{\n"
                       + classList.elementAt(i).toString() + "}\n\n");

        return(buf.toString());
    }
}


class Classcontents extends Tree
{
    private int getCounter = 0;
    private Vector defList = new Vector(); // list of vardef/fncdef

    // constructor
    Classcontents(int lineno){super.lineno = lineno;}

    public void add(Object o){defList.addElement(o);}

    public void resetGet(){getCounter = 0;}

    public Tree getNext()
    {
        if(getCounter > -1 && getCounter < defList.size())
            return((Tree) defList.elementAt(getCounter++));
        return(null);
    }

    public void pushBack(){getCounter--;}

    public String toString()
    {
        StringBuffer buf = new StringBuffer();

        for(int i = 0; i < defList.size(); i++)
            buf.append( defList.elementAt(i).toString() + "\n");

        return(buf.toString());
    }
}


class Vardef extends Tree
{
    private String name;
    private String type;

    // constructor
    Vardef(String t, String n, int lineno)
    {type=t; name=n; super.lineno = lineno;}

    public String getName(){ return(name); }
    public String getType(){ return(type); }

    public String toString()
    {
        return("(vardef) " + type + " " + name);
    }
}
```

```
class Fncdef extends Tree
{
    private String name;
    private Vector argList; // contains pair of <ID>, == 2*n number of <ID>
    private Tree sentences;

    // constructor
    Fncdef(String name, Vector list, Tree s, int lineno)
    {this.name = name; argList = list; sentences = s; super.lineno = lineno;}


    public String getName(){return(name);}
    public Vector getArgList(){return(argList);}
    public Sentences getSentence(){return((Sentences) sentences);}

    public String toString()
    {
        // navn
        StringBuffer buf = new StringBuffer("\n(fncdef) void " + name + "(");

        // argumenter
        for(int i = 0; i < argList.size(); i++)
        {
            if(i != 0) buf.append(", ");
            buf.append((String) argList.elementAt(i++) + " " +
                       (String) argList.elementAt(i));
        }

        // indhold
        buf.append(")\n{\n" + sentences.toString() + "}");

        return(buf.toString());
    }
}



class Sentences extends Tree
{
    private int getCounter = 0;
    private Vector sList = new Vector(); // liste of <sentences>

    // constructor
    Sentences(int lineno){super.lineno = lineno;}
    public void add(Object o){sList.addElement(o);}

    public Tree getNext()
    {
        if(getCounter > -1 && getCounter < sList.size())
            return((Tree) sList.elementAt(getCounter++));
        return(null);
    }

    public void pushBack(){getCounter--;}

    public String toString()
    {
```

```
        StringBuffer buf = new StringBuffer();

        for(int i = 0; i < sList.size(); i++)
            buf.append( sList.elementAt(i).toString() + "\n");

        return(buf.toString());
    }
}


class Fnccall extends Tree
{
    private boolean localCall;
    private String name;
    private Vector argList = new Vector();

    // constructor
    Fnccall(String name, boolean localCall, int lineno)
    {this.name = name; this.localCall = localCall; super.lineno = lineno;}

    // to convert an OpCall to a Fnccall
    Fnccall(String name, Vector argList, boolean localCall)
    {this.name = name; this.argList = argList; this.localCall = localCall;}

    public String getName(){return(name);}
    public Vector getArgList(){return(argList);}
    public boolean isCallLocal(){return(localCall);}


    public void add(Object o){argList.addElement(o);}

    public String toString()
    {
        StringBuffer buf = new StringBuffer("\n(fnccall) "+ name + "(");

        for(int i = 0; i < argList.size(); i++)
        {
            if(i > 0) buf.append(", ");
            buf.append(argList.elementAt(i).toString() + "\n");
        }

        return(buf.toString() + ")");
    }
}


class If extends Tree
{
    private Tree condCode;
    private Tree thenCode;
    private Tree elseCode;

    // constructor
    If(Tree c, Tree t, Tree e, int lineno)
    { condCode = c; thenCode = t; elseCode = e; super.lineno = lineno;}

    public Tree getCond(){return(condCode);}
    public Tree getThen(){return(thenCode);}
    public Tree getElse(){return(elseCode);}
```

```
    public String toString()
    {
        return("if(" + condCode.toString() + ")\n{\n"
                     + thenCode.toString() + "}\nelse\n{\n"
                     + elseCode.toString() + "\n}\n");
    }
}


class While extends Tree
{
    private Tree condCode;
    private Tree whileCode;

    //constructor
    While(Tree c, Tree w, int lineno){condCode = c; whileCode = w;super.lineno = lineno;}

    public Tree getCond(){return(condCode);}
    public Tree getWhile(){return(whileCode);}

    public String toString()
    {
        return("while(" + condCode.toString() + ")\n{"
                        + whileCode.toString()+ "}\n");
    }
}

class Break extends Tree
{
    // constructor
    Break(int lineno){super.lineno = lineno;}

    public String toString(){return("break");}
}


class Return extends Tree
{
    //constructor
    Return(int lineno){super.lineno = lineno;}

    public String toString(){return("return");}
}


class Assign extends Tree
{
    int op; // determine wether we have <ID> or <NAME>
    private String name;
    private Tree E;

    // constructor
    Assign(String name, int op, Tree E, int lineno)
    {this.name = name; this.op = op; this.E = E; super.lineno = lineno;}


    public String getName(){return(name);}
    public Tree getE(){return(E);}
```

```
    public int getOp(){return(op);}

    public String toString()
    {
        return("(assign) " + name + " = " + E.toString());
    }
}


class OpMonadic extends Tree implements TokenNames
{
    private int op;     // == TokenName
    private Tree right;

    // constructor
    OpMonadic(int op, Tree right, int lineno)
    {this.op = op; this.right = right; super.lineno = lineno;}

    public int getOp(){return(op);}
    public Tree getR(){return(right);}

    public String toString()
    {
        switch(op)
        {
            case NOT:   return("(! " + right.toString()+")" );
            case MINUS: return("(- " + right.toString()+")" );
            default: errorAndExit("Strange operator (#"+op+") in 'opMonadic' made me stop compiling!\n");
        }
        return("dummy to make Java<tm> happy");
    }
}


class OpDual extends Tree implements TokenNames
{
    private int op;        // == TokenName
    private Tree left, right;

    // constructor
    OpDual(int op, Tree left, Tree right, int lineno)
    { this.op = op; this.left = left; this.right = right; super.lineno = lineno;}

    public Tree getL(){return(left);}
    public Tree getR(){return(right);}
    public int getOp(){return(op);}

    public String toString()
    {
        StringBuffer buf = new StringBuffer("("+left.toString() + " ");
        switch(op)
        {
            case PLUS:   buf.append("+"); break;
            case MINUS:  buf.append("-"); break;
            case MULT:   buf.append("*"); break;
            case DIV:    buf.append("/"); break;
            case SET:    buf.append("="); break;
            case NEQUAL: buf.append("!="); break;
            case EQUAL:  buf.append("=="); break;
```

```
            case LEQUAL: buf.append("<="); break;
            case LESS:   buf.append("<"); break;
            case AND:    buf.append("&&"); break;
            case BITAND: buf.append("&"); break;
            case OR:     buf.append("||"); break;
            case BITOR:  buf.append("|"); break;
            case MODULO: buf.append("%"); break;
            default: errorAndExit("Strange operator (#"+op+") in 'opDual' made me stop compiling!\n");
        }
        return(buf + " " + right.toString()+")" );
    }
}



// functioncalls samt NEW
class OpCall extends Tree implements TokenNames
{
    private boolean fncCall = false;

    private int op;            // anvendes til at bestemme om det er et "new" eller et funktionskald
    private String name;
    private Vector argList = new Vector();

    // constructor
    OpCall(int op, String name, int lineno)
    {this.op = op; this.name = name; super.lineno = lineno;}

    public boolean isFncCall(){return(fncCall);}
    public void setIsFncCall(){fncCall = true;}

    public int getOp(){return(op);}
    public String getName(){return(name);}
    public Vector getArgList(){return(argList);}


    public void add(Object o){argList.addElement(o);}

    public String toString()
    {
        if(op == NEW)
            return("new " + name + "() ");
        else
        {
            StringBuffer buf = new StringBuffer(name);

            if(argList.size() > 0)
            {    buf.append("(");
                for(int i = 0; i < argList.size(); i++)
                {
                    if(i != 0) buf.append(", ");
                    buf.append(argList.elementAt(i).toString());
                }
                return(buf.toString() + ")");
            }
            else
                return(buf.toString());
        }
    }
```

```
}


class OpConst extends Tree implements TokenNames
{
    private int op;        // tokenName
    private int nval;
    private String sval;

    // constructor
    OpConst(int op, int nval, int lineno)
    {this.op = op; this.nval = nval; super.lineno = lineno;}

    OpConst(int op, String sval, int lineno)
    {this.op = op; this.sval = sval; super.lineno = lineno;}



    // for the code generator
    public int getOp(){return(op);}
    public int getNval(){return(nval);}
    public String getSval(){return(sval);}

    public String toString()
    {
        switch(op)
        {
            case VAL_CHAR:   return("'"+sval+"' ");
            case VAL_STRING: return("\""+sval+"\" ");
            case VAL_INT:    String s = "" + nval; return(s);
            default: errorAndExit("Strange operator (#"+op+") in 'opConst' made me stop compiling!\n");
        }
        return("dummy to make Java<tm> happy");
    }
}
```

SymbolTable H

```
import java.util.Vector;

public class SymbolTable
{
    private Vector list = new Vector(); // contains elements

    protected class Symbol
    {
        int index;
        String fromClass, fromFnc, name, type;

        // konstruktør
        Symbol(String t, String fC, String fF, String n, int i)
        {type = t; fromClass = fC; fromFnc = fF; name = n; index = i;}
    }



    public void add(String type, String fromClass, String fromFnc, String name, int index)
    {
        this.add(type, fromClass, fromFnc, name);

        Symbol s = (Symbol) list.lastElement();
        s.index = index;

        list.setElementAt(s, list.size()-1); // overwrite element with the new index value
    }


    public void add(String type, String fromClass, String fromFnc, String name)
    {
        Symbol s;
        int size = 0; // the index possition

        if(fromFnc == null)
        {
            // first find no. variables in same scope
            for(int i = 0; i < list.size(); i++)
            {
```

```
            s = (Symbol) list.elementAt(i);
            // same class but no function-scope
            if(fromClass.equals(s.fromClass) && fromFnc == null)
                size++;
        }
    }
    else
    {
        // first find no. variables in same scope
        for(int i = 0; i < list.size(); i++)
        {
            s = (Symbol) list.elementAt(i);
            // same class but no function-scope
            if(fromClass.equals(s.fromClass) && fromFnc.equals(s.fromFnc) )
                size++;
        }
    }

    s = new Symbol(type, fromClass, fromFnc, name, size*2);

    list.addElement(s);
}


public void remove(String type, String fromClass, String fromFnc, String name)
{
    Symbol s;

    for(int i = 0; i < list.size(); i++)
    {
        s = (Symbol) list.elementAt(i);

        // we must handle null-pointers sepperatly
        if(fromFnc == null)
        {
            if(fromClass.equals(s.fromClass) && s.fromFnc == null &&
               name.equals(s.name) )
            {
                list.removeElementAt(i);
                return;
            }
        }
        else
        if(fromClass.equals(s.fromClass) && fromFnc.equals(s.fromFnc) &&
           name.equals(s.name) )
        {
            list.removeElementAt(i);
            return;
        }
    }
}


/* return values:
 * -1 - var not found
 *  0 - wrong type
 *  1 - type is ok.
 */
public int typeCheck(String type, String fromClass, String fromFnc, String name)
```

```
{
    Symbol s;
    // find element (class and name)
    for(int i = 0; i < list.size(); i++)
    {
        s = (Symbol) list.elementAt(i);
        if(fromClass.equals(s.fromClass) && name.equals(s.name) )
        {
            // check if variable is from the same function
            if(fromFnc != null && !fromFnc.equals(s.fromFnc) )
                continue;

            // compare types
            if(type.equals(s.type) )
                return(1);
            else
                return(0);
        }
    }
    return(-1);
}


// returns the type of a given variable
public String varType(String fromClass, String fromFnc, String name)
{
    Symbol s;
    // find element (class and name)
    for(int i = 0; i < list.size(); i++)
    {
        s = (Symbol) list.elementAt(i);
        if(fromClass.equals(s.fromClass) && name.equals(s.name) )
            return(s.type);
    }
    return(null);
}

// returns the size of function in bytes == 2 * #local variables
public int fncSize(String className, String fncName)
{
    Symbol s;
    int i, size = 0;

    // summerize all variables with same classname with no [tilhørsforhold] to a function
    for(i = 0; i < list.size(); i++)
    {
        s = (Symbol) list.elementAt(i);
        if(className.equals(s.fromClass) && fncName.equals(s.fromFnc))
            size++;
    }

    // we *2 in indexsize, since all elements is implemented as
    // 16 bits == 2 elements on the stack/hob
    return(size*2);
}


// returns size of class in bytes = 2 * #local variables
public int classSize(String className)
```

```
    {
        Symbol s;
        int i, size = 0;

        // summerize all variables with same classname with no tilhørsforhold to a function
        for(i = 0; i < list.size(); i++)
        {
            s = (Symbol) list.elementAt(i);
            if(className.equals(s.fromClass) && s.fromFnc == null)
                size++;
        }
        // we *2 in indexsize, since all elements is implemented as
        // 16 bits == 2 elements on the stack/hob
        return(size*2);
    }


    public int varIndex(String fromClass, String fromFnc, String name)
    {
        Symbol s;
        // find element (class and name)
        for(int i = 0; i < list.size(); i++)
        {
            s = (Symbol) list.elementAt(i);
            if(fromClass.equals(s.fromClass) && name.equals(s.name) )
            {

                // check if variable is from the same function (if from a function)
                if(fromFnc == null && s.fromFnc == null)
                    return(s.index);
                else
                if(fromFnc != null && fromFnc.equals(s.fromFnc) )
                    return(s.index);
            }
        }

        // failed to find variable,

        return(-1); // failed to find variable!
    }


/*
// for test only!
static void main(String[] a)
{
        SymbolTable s = new SymbolTable();

        s.add("int", "A", "f", "i");
        s.add("int", "A", null, "k");
        s.add("char", "A", "f", "j");

        s.add("int", "A", "g", "j");
        s.add("int", "B", null, "k");
        s.add("char", "A", null, "x");

        System.out.println("A = " + s.classSize("A") );
        System.out.println("A.f() = " + s.fFncSize("A", "f") );
        System.out.println("pos A.f.j = " + s.varIndex("A", "f", "j") );
```

```
        System.out.println("pos B.k = " + s.varIndex("B", null, "k") );
        System.out.println("A.g() = " + s.fncSize("A", "g") );

        if(s.typeCheck("int", "A", null, "i") == true)  System.out.println("i er int");
        else System.out.println("i != int ");

        if(s.typeCheck("char", "A", "f", "j") == true)  System.out.println("A.f().j er char");
        else System.out.println("i != int ");

        if(s.typeCheck("int", "A", "f", "j") == true)   System.out.print("A.f.j er int");
        else System.out.println("A.f.j != int ");

}
*/

}
```

— Bilag —

AsmBlock I

```
import java.util.Vector;

public class AsmBlock
{
    private Vector b = new Vector();

    public void add(String s){b.addElement(s);}

    public String toString()
    {
        StringBuffer buf = new StringBuffer();

        for(int i = 0; i < b.size(); i++)
        {
            String s = (String) b.elementAt(i);

            // if not a label, insert spaces
            if((s.endsWith(":") || s.endsWith("PROC") || s.endsWith("ENDP") || s.endsWith("DOSSEG") || s.endsWith
                 buf.append("    ");
            buf.append( s + "\n");
        }

        return(buf.toString());
    }
}
```

— Bilag —

CodeGenerator J

```
import java.util.Vector;

public class CodeGenerator implements TokenNames
{
    // user-specifik compileroptions
    private String HOBSIZE = "32605";

    // to generate unique labels
    private int X = 0;

    // temporary storage of program divided into the two assembler segments
    // and a block of startup code
    private AsmBlock segptr; // points to the AsmBlock to be written to
    private AsmBlock startcode=new AsmBlock(); // upstart code
    private AsmBlock stdfnc  = new AsmBlock(); // standard functions
    private AsmBlock dataseg = new AsmBlock(); // data segment code
    private AsmBlock codeseg = new AsmBlock(); // code segment code
    private AsmBlock maincode= new AsmBlock(); // code containing static void main()
    // other
    String className;   // name of class we are currently working in
    String fncName;     // name of fnc we are currently working in
    SymbolTable sbtb;   // symbol table containing all variables in parsetree


    // konstruktør
    CodeGenerator(SymbolTable sbtb){this.sbtb = sbtb;}


    private void errorAndExit(String s)
    {System.out.print("\n" + s); System.exit(0);}


    private void errorAndExit(String s, String s2, int l)
    {System.out.print(s + "("+l+"): " + s2); System.exit(0);}

    public String generate(Tree parsetree)
    {

        generateStartCode();
```

```
    segptr = codeseg; // write the whole thing to codeseg
    sGen((Classdef) parsetree);
    return( startcode.toString() + dataseg.toString() +
            stdfnc.toString()    + codeseg.toString() +
            maincode.toString()
          );
}


private void sGen(Classdef tree)
{
    // a classdefinition is defined as two sepperate elements in the parsetree
    className = (String) tree.getNext();

    while(className != null)
    {
        fncName = null;
        classdefGen((Classcontents) tree.getNext() );

        className = (String) tree.getNext(); // read next elem
    }
}


private void classdefGen(Classcontents tree)
{   classcontentsGen(tree); }


private void classcontentsGen(Classcontents tree)
{
    Tree t = tree.getNext();

    while(t != null)
    {
        if(t instanceof Vardef)
        {
            // do nothing, as all the variables are allocated on beforehand.
        }
        else // instanceof Fncdef
        {
            // special case if functionname == "main"
            if( ((Fncdef) t).getName().equals("main") )
                fncdefMainGen((Fncdef) t);
            else
                fncdefGen((Fncdef) t);
        }

        t = tree.getNext();
    }
}



private void vardefGen(Vardef tree)
{
    return; // variables are handed by newGen() and fncdefGen()
}
```

```
private void fncdefMainGen(Fncdef tree)
{
    int offset = 0;
    fncName = "main";
    String fnc_endlabel = className + "_" + fncName + "_end";

    segptr = maincode;

    segptr.add("\n\nSTART:");
    segptr.add("mov  ax, @data ; setup DOS program");
    segptr.add("mov  ds, ax");
    segptr.add("; allocate object main resides in");
    segptr.add("mov  cx, "+sbtb.classSize(className) );
    segptr.add("call new");
    segptr.add("mov  bp, sp     ; adjust bp to point at functions local variable");
    segptr.add("sub  bp, 2      ;");
    segptr.add("mov  bx, ax    ; store main's 'this' ");
    segptr.add("; execute mains contents");

    // put all fnc local variables on stack with value = 0
    int fncsize = sbtb.fncSize(className, fncName)/2;
    for(int i = 0; i < fncsize; i++)
        segptr.add("push 0    ; push local variables to stack ");

    // generate all the sentences in function
    sentencesGen((Sentences) tree.getSentence(), null, fnc_endlabel);
    segptr.add("jmp  SystemExit");
    segptr.add("END");

    fncName = null;
    segptr = codeseg; // write again to the codeseg segment
}
```

```
/*
 * all the argumentvariables have been put on stack - we must added them to the
 * SymbolTable in the fncCallGen() so their possition on the stack can be calculated
 */
private void fncdefGen(Fncdef tree)
{
    int offset = 0;
    fncName = tree.getName();
    Vector argList = tree.getArgList();

    String fnc_endlabel = className + "_" + fncName + "_end";

    segptr.add("\n\n"+className+"_" + fncName + " PROC" ); // declare PROC
    segptr.add("mov  bp, sp ; adjust bp to point at functions local variable");
    segptr.add("sub  bp, 2");

    // put all fnc local variables on stack with value = 0
    int fncsize = sbtb.fncSize(className, fncName);
    for(int i = 0; i < fncsize/2; i++)
        segptr.add("push 0 ; put all local variables on stack");
```

```
    // put arglist variable names in SymbolTable with their corresponding
    // index # on the stack.
    // index # = first local variable (2bytes) + return adr (2bytes)
    // the index is also negated, as argument-variables are fetched with [bp+x]
    // unlike local variables which are fetched with [bp-x]
    for(int index=0, i=0; i < argList.size(); i++)
    {
        sbtb.add((String) argList.elementAt(i++), className, fncName,
                 (String) argList.elementAt(i), -(4+index) );
        index += 2;
    }


    // generate all the sentences in function
    sentencesGen((Sentences) tree.getSentence(), null, fnc_endlabel);



    // fnc end_label must be set in front of the de-stacking of variables
    segptr.add(fnc_endlabel + ":");

    // de-stack local variables
    for(int i = 0; i < fncsize/2; i++)
        segptr.add("pop  ax   ; de-stack local variables");

    // end function
    segptr.add("ret");
    segptr.add(className+"_" + fncName + " ENDP" ); // declare PROC

    fncName = null;

    // clear SymbolTable of argument variables
    for(int i = 0; i < argList.size(); i++)
    {
        sbtb.remove((String) argList.elementAt(i++), className,
                 fncName, (String) argList.elementAt(i));
    }
}



// input <sentences>, endlabel == if/while or fnc if break is called outside,
private void sentencesGen(Sentences tree, String endlabel, String fnc_endlabel)
{
    Tree t = tree.getNext();
    while(t != null)
    {
        if(t instanceof Vardef)
            vardefGen((Vardef) t);
        else
        if(t instanceof Fnccall)
            fnccallGen((Fnccall) t);
        else
        if(t instanceof If)
            ifGen((If) t, fnc_endlabel);
        else
        if(t instanceof While)
            whileGen((While) t, fnc_endlabel);
```

```
        else
        if(t instanceof Break)
            breakGen((Break) t, endlabel);
        else
        if(t instanceof Return)
            segptr.add("jmp  "+fnc_endlabel+" ; return");
        else
        if(t instanceof Assign)
            assignGen((Assign) t);

        t = tree.getNext();
    }
}



private void fnccallGen(Fnccall tree)
{
    String name = tree.getName();

    segptr.add("; fnccallGen");

    // standard functions
    if(tree.getName().equals("System.out.print") )
    {
        segptr.add("push bp");
        segptr.add("push bx");
        eGen((Tree)tree.getArgList().elementAt(0));
        segptr.add("call SystemOutPrint");
        segptr.add("pop  cx ; de-stack argument");
        segptr.add("pop  bx");
        segptr.add("pop  bp");
        return;
    }

    if(tree.getName().equals("System.in.read"))
    {
        segptr.add("call SystemInRead");
        segptr.add("push ax ; store read letter");
        return;
    }

    if(tree.getName().equals("System.exit"))
    {
        segptr.add("jmp  SystemExit");
        return;
    }

    if(tree.getName().equals("Integer.toString"))
    {
        // generate first argument and put result in ax
        segptr.add("push bp");
        segptr.add("push bx");
        eGen((Tree)tree.getArgList().elementAt(0));
        segptr.add("call Integer_toString");
        segptr.add("pop  cx ; de-stack arguments");
        segptr.add("pop  bx");
        segptr.add("pop  bp");
        segptr.add("push ax ; store generated String");
```

```
    return;
}


// else non-local non standard function

String callFnc   ="";// used in if isCallLocal == false
String callClass ="";// used in if isCallLocal == false

if(tree.isCallLocal() == false)
{
    // we must first extract Obj pointer name  == classptr
    int sepp = name.indexOf('.');
    String classptr = name.substring(0, sepp);
    callFnc = name.substring(sepp+1);

    // find the pointers type to determine which class to call
    callClass = sbtb.varType(className, fncName, classptr);

    // reference variable is defined in class scope
    if(callClass == null)
    {
        callClass = sbtb.varType(className, null, classptr);
        int stackPos = sbtb.varIndex(className, null, classptr);

        segptr.add("push bp");
        segptr.add("push bx");
        segptr.add("mov  bx, hob[" + stackPos + "] ; push callers class' 'this'");
    }
    else
    {
        // fetch 'this' for the class that the pointer points at
        int stackPos = sbtb.varIndex(className, fncName, classptr);

        if(stackPos != -1)
        {
            segptr.add("push bp");
            segptr.add("push bx");
            segptr.add("mov  bx, [bp-" + stackPos + "] ; push callers class' 'this'");
        }
        else
            errorAndExit("Error", "Functioncall '"+name+"()' is unknown!", tree.lineno());
    }
}
else // local non standard function
{
    segptr.add("push bp");
    // since the call is local, 'this' (bx) is set up correctly
}


/* push all arguments to stack (local variables will be pushed as the
 * first thing in the funktion)
 * The names will later be connected with the stack possition (in the
 * begining of the function, but generated in fncdefGen() )
 * All we do here is putting the values on the stack in reverse order, so
 * the varIndex in SymbolTable works correctly
 */
Vector v = tree.getArgList();
```

```
    for(int i = v.size()-1; i >= 0 ; i--)
        eGen((Tree) v.elementAt(i));

    if(tree.isCallLocal() == true)
        segptr.add("call "+className+"_"+name);
    else
    {
        segptr.add("call " + callClass +"_"+ callFnc);
    }


    // since functions normaly doesn't support return-value we
    // have to do something special in the cases where we need a
    // returnvalue
    for(int i = 0; i < v.size(); i++)
        segptr.add("pop  cx ; de-stack arguments");

    if(tree.isCallLocal() == false)
    {
        segptr.add("pop  bx ; restore this class' 'this'");
        segptr.add("pop  bp");
    }
    else // only restore bp in local calls
    {
        segptr.add("pop bp");
    }

    // we do not have returnvalues for functions, but in these special
    // cases we need it and so "emulate" it.
    if(callFnc.equals("length") ||
       callFnc.equals("charAt") ||
       callFnc.equals("concat")
      )
        segptr.add("push  ax ; 'emulated' return value on the stack");

}



private void ifGen(If tree, String fnc_endlabel)
{
    int x = X++;
    final String ifend_label = "endif_"+X;
    final String else_label  = "else_"+X;
    final String then_label  = "then_"+X;

    // gen IF
    eGen(tree.getCond());
    segptr.add("pop  cx     ; start if_"+x);
    segptr.add("cmp  cx, 0");
    segptr.add("je   " + else_label); // if CX == 0

    // gen THEN
    segptr.add(then_label+":");
    sentencesGen((Sentences) tree.getThen(), null, fnc_endlabel);
    segptr.add("jmp  " + ifend_label);

    // gen ELSE
    segptr.add(else_label+":");
```

```
    sentencesGen((Sentences) tree.getElse(), null, fnc_endlabel);
    segptr.add(ifend_label+":");
}



private void whileGen(While tree, String fnc_endlabel)
{
    X++;
    String while_startlabel = "start_while"+X;
    String while_endlabel = "end_while"+X;

    // condition
    segptr.add(while_startlabel+":");
    segptr.add("; condition code");
    eGen(tree.getCond() );

    segptr.add("pop  ax ; compare condition code");
    segptr.add("cmp  ax, 0");
    segptr.add("je   " + while_endlabel + " ; condition is false");
    sentencesGen((Sentences) tree.getWhile(), while_endlabel, fnc_endlabel);
    segptr.add("jmp  " + while_startlabel + " ; loop once more");
    segptr.add(while_endlabel+":");
}


private void breakGen(Break tree, String endlabel)
{
    if(endlabel == null)
    {
        String fnc;
        if(fncName != null)
            fnc = fncName + "() ";
        else
            fnc = " ";

        errorAndExit("Error", "\"break\" is only allowed inside while-scopes!", tree.lineno());
    }
    else
        segptr.add("jmp  "+endlabel+" ; break");
}



private void assignGen(Assign tree)
{   segptr.add("; assign");

    eGen(tree.getE());
    segptr.add("pop  ax ; get value (assign)");

    if(tree.getOp() == ID)
    {
        String varname = tree.getName();
        int index = sbtb.varIndex(className, null, varname);

        if(index >= 0)
        {
            segptr.add("mov  hob[bx+"+index+"], ax ; set class variable " + varname);
```

```
            }
            else// not a class variable, but a fnc variable
            {
                index = sbtb.varIndex(className, fncName, varname);

                if(index == -1)
                    errorAndExit("Error","Variable "+varname+" not defined in current scope.",
                                 tree.lineno());

                // if variable is from arguments, they must be fetched differently
                if(index < 0)
                {
                    segptr.add("mov  [bp+"+(index*-1)+"], ax ; set argument variable " + varname);
                }
                else
                    segptr.add("mov  [bp-"+index+"], ax ; set local variable " + varname);
            }
        }
        else // getOp() == NAME
        {
            String name = tree.getName();

            // we must first extract Obj pointer name
            int sepp = name.indexOf('.');
            String classptr = name.substring(0, sepp);
            String varname = name.substring(sepp+1);

            // find the class ptr is pointing at
            String callClass = sbtb.varType(className, fncName, classptr);

            int classptrIndex = sbtb.varIndex(className, null, varname);
            int varindexOtherClass = sbtb.varIndex(callClass, null, varname);

            if(varindexOtherClass >= 0)
            {
                // get value p points at (in p.a)
                segptr.add("push bx");
                segptr.add("                              ; value of " + name);
                segptr.add("mov  bx, [bp-"+(classptrIndex)+"] ; this of other class");
                // get value of a (in p.a)
                segptr.add("mov  hob[bx+"+varindexOtherClass+"], ax ; set value of variable
                                                                      in other class");
                segptr.add("pop  bx");
            }
            else
            {
                errorAndExit("Error", "Variable \""+name+"\" does not exist.", tree.lineno());
            }
        }
    }


    // generate <E>
    private void eGen(Tree e)
    {
        if(e instanceof OpMonadic)  opMonadicGen((OpMonadic) e);
        else
        if(e instanceof OpDual)     opDualGen((OpDual) e);
        else
```

```
    if(e instanceof OpConst)    opConstGen((OpConst) e);
    else
    if(e instanceof OpCall)     opCallGen((OpCall) e);
    else
        errorAndExit("Internal error: Tried to generate <E> - but tree was no <E>!");
}


private void opMonadicGen(OpMonadic tree)
{   segptr.add("; monadic begin");
    eGen(tree.getR() );
    segptr.add("pop  ax");

    switch(tree.getOp())
    {
        case MINUS:
            segptr.add("neg  ax");
            segptr.add("push ax");
            break;

        case NOT:
            segptr.add("not  ax");
            segptr.add("push ax");
            break;
    }
}



private void opDualGen(OpDual tree)
{
    eGen(tree.getL() );
    segptr.add("; ");
    eGen(tree.getR() );

    switch(tree.getOp())
    {
        case OR:
        case BITOR:
            segptr.add("pop  dx");
            segptr.add("pop  ax");
            segptr.add("or   ax, dx");
            segptr.add("push ax");
            break;

        case AND:
        case BITAND:
            segptr.add("pop  dx");
            segptr.add("pop  ax");
            segptr.add("and  ax, dx");
            segptr.add("push ax");
            break;

        case EQUAL:
            X++;
            segptr.add("; equal");
            segptr.add("pop  dx");
            segptr.add("pop  ax");
            segptr.add("cmp  ax, dx");
```

```
    segptr.add("je   eq_"+X);
    segptr.add("xor  ax, ax  ; is !="); // false result
    segptr.add("jmp  eq_end"+X);
    segptr.add("eq_"+X+":");
    segptr.add("mov  ax, 65535 ; is =="); // true result
    segptr.add("eq_end"+X+":");
    segptr.add("push ax");
    break;

case NEQUAL:
    X++;
    segptr.add("pop  dx");
    segptr.add("pop  ax");
    segptr.add("cmp  ax, dx");
    segptr.add("jne  neq_"+X); // remember 2 != 3 is true
    segptr.add("xor  ax, ax  ; is ==");
    segptr.add("jmp  neq_end"+X);
    segptr.add("neq_"+X+":");
    segptr.add("mov  ax, 65535 ; is !="); // true result
    segptr.add("neq_end"+X+":");
    segptr.add("push ax");
    break;

case LESS:
    X++;
    segptr.add("pop  dx");
    segptr.add("pop  ax");
    segptr.add("cmp  ax, dx");
    segptr.add("jl   less_"+X);
    segptr.add("xor  ax, ax ; is not <");
    segptr.add("jmp  less_end"+X);
    segptr.add("less_"+X+":");
    segptr.add("mov  ax, 65535 ; is <"); // true result
    segptr.add("less_end"+X+":");
    segptr.add("push ax");
    break;

case LEQUAL:
    X++;
    segptr.add("pop  dx");
    segptr.add("pop  ax");
    segptr.add("cmp  ax, dx");
    segptr.add("jle  lequal_"+X);
    segptr.add("xor  ax, ax  ; is not <=");
    segptr.add("jmp  lequal_end"+X);
    segptr.add("lequal_"+X+":");
    segptr.add("mov  ax, 65535 ; is <="); // true result
    segptr.add("lequal_end"+X+":");
    segptr.add("push ax");
    break;

case PLUS:
    segptr.add("pop  dx");
    segptr.add("pop  ax");
    segptr.add("add  ax, dx");
    segptr.add("push ax");
    break;

case MINUS:
```

```
            segptr.add("pop  dx");
            segptr.add("pop  ax");
            segptr.add("sub  ax, dx");
            segptr.add("push ax");
            break;


        case MULT:
            segptr.add("pop  dx");
            segptr.add("pop  ax");
            segptr.add("mul  dx");
            segptr.add("push ax");
            break;


        case DIV:
            segptr.add("pop  si");
            segptr.add("pop  ax");
            segptr.add("xor  dx, dx");
            segptr.add("div  si");
            segptr.add("push ax");
            break;

        case MODULO:
            segptr.add("pop  si");
            segptr.add("pop  ax");
            segptr.add("xor  dx, dx");
            segptr.add("div  si");
            segptr.add("push dx");
            break;
    }
}

private void opConstGen(OpConst t)
{
    switch(t.getOp())
    {
        case VAL_INT:
            segptr.add("mov  ax, " + t.getNval() );
            segptr.add("push ax");
            break;

        case VAL_CHAR:
            segptr.add("mov  ax, '" + t.getSval() + "'");
            segptr.add("push ax");
            break;

        case VAL_STRING:
            String s = t.getSval();
            int len = s.length();
            segptr.add("; string - allocate class and return pointer on stack");
            segptr.add("mov  cx, "+((2*len)+2) + " ; length of 2*str + 2 ");
            segptr.add("call new");
            segptr.add("mov  si, ax ; mov (adr of obj in hob) to si");

            segptr.add("mov  hob[si], "+len+" ; insert strlength");
            for(int i = 0, index = 2; i < len; i++, index+=2)
                segptr.add("mov  hob[si+"+index+"], '"+ s.charAt(i)+ "'");
```

```
            segptr.add("push ax");
            break;

        default: errorAndExit("Internal error in opConstGen()");
    }
}



private void opCallGen(OpCall tree)
{
    // variable from this object
    if(tree.getOp() == ID && tree.isFncCall() == false)
    {
        String varname = tree.getName();
        int index = sbtb.varIndex(className, fncName, varname);

        // variable is declared in class-scope
        if(index == -1)
        {
            index = sbtb.varIndex(className, null, varname);

            // if still unknown the variable does not exist
            if(index == -1)
                errorAndExit("Error", "variable \"" + tree.getName() +
                             "\" is not defined within the scope it was used.", tree.lineno());

            segptr.add("mov  ax, hob[bx+"+index+"] ; get class variable " + varname);
            segptr.add("push ax");
            return;
        }
        if(index < 0)
        {
            segptr.add("mov  ax, [bp+"+(index*-1)+"] ; get argument variable " + varname);
            segptr.add("push ax");
        }
        else
        {
            segptr.add("mov  ax, [bp-"+(index)+"] ; get local variable " + varname);
            segptr.add("push ax");
        }
    }
    else
    // variable from other object
    if(tree.getOp() == NAME && tree.isFncCall() == false)
    {

        // we must first extract Obj pointer name
        int sepp = tree.getName().indexOf('.');
        String classptr = tree.getName().substring(0, sepp); // name of class, "p" in p.a
        String varname = tree.getName().substring(sepp+1);   // name of var,   "a" in p.a

        // find the class ptr is pointing at
        String callClass = sbtb.varType(className, fncName, classptr);

        if(callClass != null)
        {
            int classptrIndex = sbtb.varIndex(className, null, varname);
            int varindexOtherClass = sbtb.varIndex(callClass, null, varname);
```

```
            // if still unknown the variable does not exist
            if(varindexOtherClass == -1)
                errorAndExit("Error", "variable \""+varname+"\" is not defined in class "
                             +callClass, tree.lineno());

            // get value p points at (in p.a)
            segptr.add("mov  ax, [bp-"+(classptrIndex)+"] ; this of other class");
            // get value of a (in p.a)
            segptr.add("mov  ax, hob[ax+"+varindexOtherClass+"] ; get value of variable
                                                                  in other class");
            segptr.add("push ax");
        }
        else
            errorAndExit("Error", "variable \""+classptr+"\" is not defined within the
                                                              scope it is being used.",
                         tree.lineno());
    }
    else
    if(tree.getOp() == NEW)
    {
        segptr.add("; new");
        segptr.add("mov  cx, "+ sbtb.classSize(tree.getName()) + " ; size of class " +
                    tree.getName() );
        segptr.add("call new");
        segptr.add("push ax ; save adr pointer of object");
    }
    else // fnccall with <ID>/<NAME>
    {
        Fnccall tmp;

        if(tree.getOp() == NAME)
            tmp = new Fnccall(tree.getName(), tree.getArgList(), false);
        else
            tmp = new Fnccall(tree.getName(), tree.getArgList(), true);

        fnccallGen(tmp);
    }
}

// All the code necesary for a program
private void generateStartCode()
{
    startcode.add("DOSSEG");
    startcode.add(".MODEL SMALL");
    startcode.add(".STACK 100h");

    dataseg.add("\n\n.DATA");
    dataseg.add("hobptr  dw 0");
    dataseg.add("hob     dw "+HOBSIZE+" dup(0) ; allocate hob");
    dataseg.add("; error messages");
    dataseg.add("oomstr db  10,13,'Out of memory! Can not allocate another class.','$',0,0");


    stdfnc.add("\n\n.CODE");
    stdfnc.add("jmp  START\n");

    stdfnc.add("; ------------ STANDARD FUNCTIONS BEGIN ----------:");
    stdfnc.add("\nnew PROC");
```

```
stdfnc.add("mov  ax, hobptr     ; get hop size");
stdfnc.add("push ax            ; save pos. of class in stack");
stdfnc.add("add  ax, cx        ; add size of obj");
stdfnc.add("cmp  ax, "+HOBSIZE);
stdfnc.add("jg   OutOfMem        ; if ax < HOBSIZE jump to OutOfMemory");
stdfnc.add("mov  hobptr, ax     ; save hop size");
stdfnc.add("pop  ax             ;   ax = pos. of class in hob");
stdfnc.add("ret");
stdfnc.add("OutOfMem:");
stdfnc.add("mov  dx, OFFSET oomstr; write out of mem string to screen");
stdfnc.add("mov  ah, 9h");
stdfnc.add("int  21h");
stdfnc.add("jmp  SystemExit");
stdfnc.add("new  ENDP");


// prints String's only!
stdfnc.add("\nSystemOutPrint PROC");
stdfnc.add("mov  bp, sp");
stdfnc.add("sub  bp, 2");
stdfnc.add("mov  bx, [bp+4]     ; get adr of str obj");
stdfnc.add("mov  cx, hob[bx]    ; strlen");
stdfnc.add("mov  si, bx         ; source-print ptr");
stdfnc.add("add  si, 2          ; get past length field");
stdfnc.add("printloop:");
stdfnc.add("mov  dx, hob[si]");
stdfnc.add("add  si, 2          ; next char");
stdfnc.add("mov  ah, 2h         ; print char");
stdfnc.add("int  21h");
stdfnc.add("loop printloop");
stdfnc.add("ret");
stdfnc.add("SystemOutPrint ENDP");


stdfnc.add("\nSystemInRead PROC");
stdfnc.add("mov  ah, 1h ; read with screen echo");
stdfnc.add("int  21h");
stdfnc.add("mov  ah, 0 ; clear AH as only AL contains userinput");
stdfnc.add("ret");
stdfnc.add("SystemInRead ENDP");


stdfnc.add("\nSystemExit PROC");
stdfnc.add("mov  ah, 4ch");
stdfnc.add("int  21h");
stdfnc.add("ret");
stdfnc.add("SystemExit ENDP");


stdfnc.add("\nInteger_toString PROC");
stdfnc.add("mov  bp, sp");
stdfnc.add("sub  bp, 2");
stdfnc.add("mov  ax, [bp+4]    ; get number (first argument)");
stdfnc.add("xor  cx, cx        ; cx counts size of str");
stdfnc.add("xor  di, di        ; flag for negative numbers");
stdfnc.add("cmp  ax, 0         ; is negative?");
stdfnc.add("jge  nonneg");
stdfnc.add("inc  di            ; di == 1 == number is negative");
stdfnc.add("inc  cx");
```

```
stdfnc.add("neg  ax           ; make the number positive");
stdfnc.add("nonneg:");
stdfnc.add("mov  si, 10       ; div with reg SI as many times as possible");
stdfnc.add("getDigits:");
stdfnc.add("xor  dx, dx");
stdfnc.add("div  si");
stdfnc.add("add  dx, 48       ; conv digit to ASCII");
stdfnc.add("push dx");
stdfnc.add("inc  cx");
stdfnc.add("cmp  ax, 0        ; are we done?");
stdfnc.add("jg   getDigits\n");
stdfnc.add("push cx           ; store cx");
stdfnc.add("shl  cx, 1        ; calc str size = (2*strlen)+2");
stdfnc.add("add  cx, 2");
stdfnc.add("call new          ; alloc new str");
stdfnc.add("pop  cx           ; get original strlen");
stdfnc.add("mov  si, ax       ; ptr to str in hob");
stdfnc.add("mov  hob[si], cx  ; store size of str");
stdfnc.add("cmp  di, 0        ; is no negative?");
stdfnc.add("je   toStr");
stdfnc.add("add  si, 2        ; next pos");
stdfnc.add("mov  hob[si], '-' ; write '-' sign");
stdfnc.add("dec  cx");
stdfnc.add("toStr:");
stdfnc.add("add  si, 2        ; next pos");
stdfnc.add("pop  dx           ; get digit");
stdfnc.add("mov  hob[si], dx  ; store size of str");
stdfnc.add("loop toStr");
stdfnc.add("ret");
stdfnc.add("Integer_toString ENDP");



stdfnc.add("\nString_charAt PROC");
stdfnc.add("mov  bp, sp");
stdfnc.add("sub  bp, 2");
stdfnc.add("mov  si, bx       ; pos in hob where obj begins");
stdfnc.add("mov  ax, [bp+4]   ; get (first) argument telling pos to get");
stdfnc.add("shl  ax, 1        ; ax = ax * 2 every char is 2 elems");
stdfnc.add("add  si, ax       ; pos in hob to fetch character");
stdfnc.add("mov  ax, hob[si+2] ; +2 as first field contains str_len");
stdfnc.add("ret");
stdfnc.add("String_charAt ENDP");


// bx is source ptr
stdfnc.add("\nString_concat PROC");
stdfnc.add("mov  bp, sp");
stdfnc.add("sub  bp, 2");
stdfnc.add("mov  cx, hob[bx] ; size of caller str");
stdfnc.add("mov  si, [bp+4]  ; pos in hob of 2nd str len");
stdfnc.add("add  cx, hob[si] ; add 2nd str len");
stdfnc.add("push cx");
stdfnc.add("add  cx, 2      ; add space for sizefield");
stdfnc.add("call new        ; alloc new String object");
stdfnc.add("pop  cx         ; total strsize");
stdfnc.add("mov  si, ax      ; can't just use the AX");
stdfnc.add("mov  hob[si], cx ; set concat'ed String size\n");
```

```
        stdfnc.add("mov  cx, hob[bx] ; size of str1");
        stdfnc.add("add  bx, 4       ; start in 2nd pos (but why +4 instead of +2 ??)");
        stdfnc.add("mov  di, ax      ; destination ptr = adr of new String obj");
        stdfnc.add("add  di, 2       ; start in 2nd pos");
        stdfnc.add("strcat1:");
        stdfnc.add("mov  si, [bx]    ; we can't just use [bx]");
        stdfnc.add("mov  hob[di], si");
        stdfnc.add("add  bx, 2");
        stdfnc.add("add  di, 2");
        stdfnc.add("loop strcat1\n");
        stdfnc.add("mov  si, [bp+4]  ; pos in hob of 2nd str len");
        stdfnc.add("mov  cx, hob[si] ; size of 2nd str len");
        stdfnc.add("mov  bx, [bp+4]  ; point at start of 2nd string");
        stdfnc.add("add  bx, 4       ; start in 2nd pos (but why +4??)");
        stdfnc.add("strcat2:");
        stdfnc.add("mov  si, [bx]    ; we can't just use [bx]");
        stdfnc.add("mov  hob[di], si");
        stdfnc.add("add  bx, 2");
        stdfnc.add("add  di, 2");
        stdfnc.add("loop strcat2");
        stdfnc.add("ret");
        stdfnc.add("String_concat ENDP");


        stdfnc.add("\nString_length PROC ; str_len length()");
        stdfnc.add("mov  ax, hob[bx] ; first field in string");
        stdfnc.add("ret");
        stdfnc.add("String_length ENDP");

        stdfnc.add("; ------------ STANDARD FUNCTIONS END ----------:");
    }
}
```

— Bilag

Qjava output

K

Nedenstående kode er output'et fra Qjava compileren der oversatte primtalstesten i kapitel 11.1 side 86, hvor standardfunktionerne er fjernet. Disse kan ses i sin fulde længde i slutningen af appendix J side 134 da de blindt indsættes i alle genererede programmer.

```
Prime_loop PROC
    mov  bp, sp ; adjust bp to point at functions local variable
    sub  bp, 2
    push 0 ; put all local variables on stack
    ; assign
    mov  ax, 3
    push ax
    pop  ax ; get value (assign)
    mov  [bp-0], ax ; set local variable p
start_while1:
    ; condition code
    mov  ax, [bp-0] ; get local variable p
    push ax
    ;
    mov  ax, 32001
    push ax
    pop  dx
    pop  ax
    cmp  ax, dx
    jl   less_2
    xor  ax, ax ; is not <
    jmp  less_end2
less_2:
    mov  ax, 65535 ; is <
less_end2:
    push ax
    pop  ax ; compare condition code
    cmp  ax, 0
    je   end_while1 ; condition is false
    ; fnccallGen
    push bp
    mov  ax, [bp-0] ; get local variable p
    push ax
    call Prime_isPrime
    pop  cx ; de-stack arguments
```

```
    pop bp
    ; assign
    mov  ax, [bp-0] ; get local variable p
    push ax
    ;
    mov  ax, 1
    push ax
    pop  dx
    pop  ax
    add  ax, dx
    push ax
    pop  ax ; get value (assign)
    mov  [bp-0], ax ; set local variable p
    jmp  start_while1 ; loop once more
end_while1:
Prime_loop_end:
    pop  ax   ; de-stack local variables
    ret
Prime_loop ENDP


Prime_isPrime PROC
    mov  bp, sp ; adjust bp to point at functions local variable
    sub  bp, 2
    push 0 ; put all local variables on stack
    ; assign
    mov  ax, 2
    push ax
    pop  ax ; get value (assign)
    mov  [bp-0], ax ; set local variable i
start_while3:
    ; condition code
    mov  ax, [bp-0] ; get local variable i
    push ax
    ;
    mov  ax, [bp+4] ; get argument variable p
    push ax
    pop  dx
    pop  ax
    cmp  ax, dx
    jl   less_4
    xor  ax, ax ; is not <
    jmp  less_end4
less_4:
    mov  ax, 65535 ; is <
less_end4:
    push ax
    pop  ax ; compare condition code
    cmp  ax, 0
    je   end_while3 ; condition is false
    mov  ax, [bp+4] ; get argument variable p
    push ax
    ;
    mov  ax, [bp-0] ; get local variable i
    push ax
    pop  si
    pop  ax
    xor  dx, dx
    div  si
```

```
    push dx
    ;
    mov  ax, 0
    push ax
    ; equal
    pop  dx
    pop  ax
    cmp  ax, dx
    je   eq_6
    xor  ax, ax  ; is !=
    jmp  eq_end6
eq_6:
    mov  ax, 65535 ; is ==
eq_end6:
    push ax
    pop  cx     ; start if_4
    cmp  cx, 0
    je   else_5
then_5:
    jmp  Prime_isPrime_end ; return
    jmp  endif_5
else_5:
    ; assign
    mov  ax, [bp-0] ; get local variable i
    push ax
    ;
    mov  ax, 1
    push ax
    pop  dx
    pop  ax
    add  ax, dx
    push ax
    pop  ax ; get value (assign)
    mov  [bp-0], ax ; set local variable i
endif_5:
    jmp  start_while3 ; loop once more
end_while3:
    mov  ax, [bp-0] ; get local variable i
    push ax
    ;
    mov  ax, [bp+4] ; get argument variable p
    push ax
    ; equal
    pop  dx
    pop  ax
    cmp  ax, dx
    je   eq_8
    xor  ax, ax  ; is !=
    jmp  eq_end8
eq_8:
    mov  ax, 65535 ; is ==
eq_end8:
    push ax
    pop  cx     ; start if_6
    cmp  cx, 0
    je   else_7
then_7:
    jmp  endif_7
else_7:
```

```
endif_7:
Prime_isPrime_end:
    pop  ax   ; de-stack local variables
    ret
Prime_isPrime ENDP


START:
    mov  ax, @data ; setup DOS program
    mov  ds, ax
    ; allocate object main resides in
    mov  cx, 0
    call new
    mov  bp, sp     ; adjust bp to point at functions local variable
    sub  bp, 2      ;
    mov  bx, ax    ; store main's 'this'
    ; execute mains contents
    push 0    ; push local variables to stack
    ; assign
    ; new
    mov  cx, 0 ; size of class Prime
    call new
    push ax ; save adr pointer of object
    pop  ax ; get value (assign)
    mov  [bp-0], ax ; set local variable ptr
    ; fnccallGen
    push bp
    push bx
    mov  bx, [bp-0] ; push callers class' 'this'
    call Prime_loop
    pop  bx ; restore this class' 'this'
    pop  bp
    jmp  SystemExit
END
```

— Bilag

Asm primtalstest L

Dette er primtalsprogrammet fra kapitel 11.1 side 86 “som en assembler programmør ville have skrevet programmet”.

```
DOSSEG
.MODEL SMALL
.STACK 200h

.DATA

.CODE

     mov ax,@DATA
     mov ds,ax

mov cx, 31999
again:
mov si, 2 ; si = 2
primeTsTStart:
cmp si, cx  ; while cx > si
je primeTstEnd ;

mov ax, cx
xor dx, dx
div si
cmp dx, 0  ; if cx % si == 0 then stop testing current number
je primeTstEnd:
inc si ; si++
jmp primeTstStart
primeTstEnd:

loop again ; if cx > 0

; END OF SHOW
    mov ah,4ch
    int 21h

END
```