

**UiO : Institutt for informatikk**  
Det matematisk-naturvitenskapelige fakultet

# **AlboC og kompilatoren hans**

Kompendium for INF2100

Stein Krogdahl, Dag Langmyhr  
Høsten 2014





# Innhold

<b>Forord</b>	<b>9</b>
<b>1 Innledning</b>	<b>11</b>
1.1 Hva er emnet INF2100? . . . . .	11
1.2 Hvorfor lage en kompilator? . . . . .	12
1.3 Om kompilatorer og liknende verktøy . . . . .	12
1.3.1 Preprosessorer . . . . .	13
1.3.2 Interpretering . . . . .	14
1.3.3 Kompilering og kjøring av Java-programmer . . . . .	14
1.4 Språkene i oppgaven . . . . .	15
1.4.1 Programmeringsspråket AlboC . . . . .	15
1.4.2 Datamaskinen x86 og dens maskinspråk . . . . .	15
1.4.3 Assembleren . . . . .	16
1.4.4 Oversikt over de ulike språkene i oppgaven . . . . .	16
1.5 Oppgaven og dens tre deler . . . . .	17
1.5.1 Del 0 . . . . .	17
1.5.2 Del 1 . . . . .	17
1.5.3 Del 2 . . . . .	17
1.6 Programmering av lister, trær etc . . . . .	18
1.7 Krav til samarbeid og gruppetilhørighet . . . . .	18
1.8 Kontroll av innlevert arbeid . . . . .	18
1.9 Delta på øvingsgruppene . . . . .	19
<b>2 Programmering i språket AlboC</b>	<b>21</b>
2.1 Kjøring . . . . .	21
2.1.1 Kompilering med C-kompilatoren . . . . .	21
2.2 AlboC-program . . . . .	22
2.2.1 Variabler . . . . .	23
2.2.2 Funksjoner . . . . .	23
2.2.3 Setninger . . . . .	24
2.2.4 Uttrykk . . . . .	26
2.2.5 Andre ting . . . . .	29
<b>3 Datamaskinen x86</b>	<b>31</b>
3.1 Minnet . . . . .	31
3.2 Prosessoren x86 . . . . .	31
3.3 Assemblerkode . . . . .	32
3.3.1 Assemblerdirektiver . . . . .	33
<b>4 Kodegenerering</b>	<b>35</b>

---

4.1	Konvensjoner . . . . .	35
4.1.1	Registre . . . . .	35
4.1.2	Navn . . . . .	35
4.2	Oversettelse av uttrykk . . . . .	36
4.2.1	Operander i uttrykk . . . . .	36
4.2.2	Operatorer i uttrykk . . . . .	37
4.3	Oversettelse av setninger . . . . .	38
4.3.1	Oversettelse av tomme setninger . . . . .	38
4.3.2	Oversettelse av tilordningssetninger . . . . .	38
4.3.3	Oversettelse av kallsetninger . . . . .	39
4.3.4	Oversettelse av for-setninger . . . . .	39
4.3.5	Oversettelse av if-setninger . . . . .	39
4.3.6	Oversettelse av return-setninger . . . . .	39
4.3.7	Oversettelse av while-setninger . . . . .	39
4.4	Oversettelse av funksjoner og funksjonskall . . . . .	40
4.4.1	Oversettelse av funksjonsdeklarasjoner . . . . .	40
4.4.2	Oversettelse av funksjonskall . . . . .	41
4.5	Deklarasjon av variabler . . . . .	41
4.5.1	Deklarasjon av globale variabler . . . . .	41
4.5.2	Deklarasjon av lokale variabler . . . . .	42
4.5.3	Deklarasjon av parametre . . . . .	42
<b>5</b>	<b>Implementasjonen</b>	<b>43</b>
5.1	Modulen <b>alboc</b> . . . . .	43
5.2	Modulen <b>chargenerator</b> . . . . .	43
5.3	Modulen <b>scanner</b> . . . . .	44
5.4	Modulen <b>syntax</b> . . . . .	45
5.5	Modulen <b>types</b> . . . . .	45
5.6	Modulen <b>code</b> . . . . .	46
5.7	Modulen <b>error</b> . . . . .	46
5.8	Modulen <b>log</b> . . . . .	46
<b>6</b>	<b>Prosjektet</b>	<b>47</b>
6.1	På egen datamaskin . . . . .	47
6.2	Tegnsett . . . . .	48
6.3	Del 0 . . . . .	49
6.4	Del 1 . . . . .	49
6.5	Del 2 . . . . .	51
6.5.1	Sjekking . . . . .	51
<b>7</b>	<b>Koding</b>	<b>67</b>
7.1	Suns anbefalte Java-stil . . . . .	67
7.1.1	Klasser . . . . .	67
7.1.2	Variabler . . . . .	67
7.1.3	Setninger . . . . .	68
7.1.4	Navn . . . . .	68
7.1.5	Utseende . . . . .	69
<b>8</b>	<b>Dokumentasjon</b>	<b>71</b>

8.1	JavaDoc . . . . .	71
8.1.1	Hvordan skrive JavaDoc-kommentarer . . . . .	71
8.1.2	Eksempel . . . . .	72
8.2	«Lesbar programmering» . . . . .	72
8.2.1	Et eksempel . . . . .	73
<b>Register</b>		<b>81</b>

# Figurer

1.1	Sammenhengen mellom AlboC, kompilator, assembler og en x86-maskin . . . . .	16
2.1	Eksempel på et AlboC-program . . . . .	22
2.2	Jernbanediagram for $\langle\text{program}\rangle$ . . . . .	23
2.3	Jernbanediagram for $\langle\text{declaration}\rangle$ . . . . .	23
2.4	Jernbanediagram for $\langle\text{type}\rangle$ . . . . .	23
2.5	Jernbanediagram for $\langle\text{var decl}\rangle$ . . . . .	23
2.6	Jernbanediagram for $\langle\text{func decl}\rangle$ . . . . .	24
2.7	Jernbanediagram for $\langle\text{param decl}\rangle$ . . . . .	24
2.8	Jernbanediagram for $\langle\text{func body}\rangle$ . . . . .	24
2.9	Jernbanediagram for $\langle\text{statm list}\rangle$ . . . . .	24
2.10	Jernbanediagram for $\langle\text{statement}\rangle$ . . . . .	24
2.11	Jernbanediagram for $\langle\text{empty statm}\rangle$ . . . . .	24
2.12	Jernbanediagram for $\langle\text{assign-statm}\rangle$ . . . . .	25
2.13	Jernbanediagram for $\langle\text{assignment}\rangle$ . . . . .	25
2.14	Jernbanediagram for $\langle\text{lhs-variable}\rangle$ . . . . .	25
2.15	Jernbanediagram for $\langle\text{call-statm}\rangle$ . . . . .	25
2.16	Jernbanediagram for $\langle\text{for-statm}\rangle$ . . . . .	25
2.17	Jernbanediagram for $\langle\text{for-control}\rangle$ . . . . .	25
2.18	Jernbanediagram for $\langle\text{if-statm}\rangle$ . . . . .	25
2.19	Jernbanediagram for $\langle\text{else-part}\rangle$ . . . . .	26
2.20	Jernbanediagram for $\langle\text{return-statm}\rangle$ . . . . .	26
2.21	Jernbanediagram for $\langle\text{while-statm}\rangle$ . . . . .	26
2.22	Jernbanediagram for $\langle\text{expression}\rangle$ . . . . .	26
2.23	Jernbanediagram for $\langle\text{rel opr}\rangle$ . . . . .	27
2.24	Jernbanediagram for $\langle\text{term}\rangle$ . . . . .	27
2.25	Jernbanediagram for $\langle\text{term opr}\rangle$ . . . . .	27
2.26	Jernbanediagram for $\langle\text{factor}\rangle$ . . . . .	27
2.27	Jernbanediagram for $\langle\text{factor opr}\rangle$ . . . . .	27
2.28	Jernbanediagram for $\langle\text{primary}\rangle$ . . . . .	27
2.29	Jernbanediagram for $\langle\text{prefix opr}\rangle$ . . . . .	27
2.30	Jernbanediagram for $\langle\text{operand}\rangle$ . . . . .	28
2.31	Jernbanediagram for $\langle\text{variable}\rangle$ . . . . .	28

2.32	Jernbanediagram for {address} . . . . .	28
2.33	Jernbanediagram for {function call} . . . . .	28
2.34	Jernbanediagram for {expr list} . . . . .	28
2.35	Jernbanediagram for {inner expr} . . . . .	28
2.36	Jernbanediagram for {name} . . . . .	29
2.37	Jernbanediagram for {number} . . . . .	29
3.1	Hovedkortet i en datamaskin . . . . .	32
3.2	Instruksjonslinje i assemblerkode . . . . .	32
5.1	Modulene i kompilatoren . . . . .	44
5.2	Oppsett for de enkelte Java-pakkene . . . . .	44
5.3	Parseringstre for programmet i figur 6.2 på side 49 . . . . .	46
6.1	De ulike delene i prosjektet . . . . .	48
6.2	Et minimalt AlboC-program <code>mini.alboc</code> . . . . .	49
6.3	Skanning av <code>mini.alboc</code> . . . . .	50
6.4	Parsering av <code>mini.alboc</code> . . . . .	50
6.5	Utskrift av treet til <code>mini.alboc</code> . . . . .	51
6.6	Navnebinding i <code>mini.alboc</code> . . . . .	51
6.7	Typesjekk i <code>mini.alboc</code> . . . . .	52
6.8	Generert kodefил for <code>mini.alboc</code> . . . . .	52
6.9	Et litt større AlboC-program <code>gcd.alboc</code> . . . . .	54
6.10	Skanning av <code>gcd.alboc</code> (del 1) . . . . .	55
6.11	Skanning av <code>gcd.alboc</code> (del 2) . . . . .	56
6.12	Parsering av <code>gcd.alboc</code> (del 1) . . . . .	57
6.13	Parsering av <code>gcd.alboc</code> (del 2) . . . . .	58
6.14	Parsering av <code>gcd.alboc</code> (del 3) . . . . .	59
6.15	Parsering av <code>gcd.alboc</code> (del 4) . . . . .	60
6.16	Parsering av <code>gcd.alboc</code> (del 5) . . . . .	61
6.17	Parsering av <code>gcd.alboc</code> (del 6) . . . . .	62
6.18	Utskrift av treet til <code>gcd.alboc</code> . . . . .	62
6.19	Navnebinding i <code>gcd.alboc</code> . . . . .	63
6.20	Typesjekking i <code>gcd.alboc</code> . . . . .	63
6.21	Generert kodefил for <code>gcd.alboc</code> (del 1) . . . . .	64
6.22	Generert kodefил for <code>gcd.alboc</code> (del 2) . . . . .	65
7.1	Suns forslag til hvordan setninger bør skrives . . . . .	68
8.1	Java-kode med JavaDoc-kommentarer . . . . .	72
8.2	«Lesbar programmering» – kildefilen <code>bubble.w0</code> del 1 . . . . .	74
8.3	«Lesbar programmering» – kildefilen <code>bubble.w0</code> del 2 . . . . .	75
8.4	«Lesbar programmering» – utskrift side 1 . . . . .	76
8.5	«Lesbar programmering» – utskrift side 2 . . . . .	77
8.6	«Lesbar programmering» – utskrift side 3 . . . . .	78
8.7	«Lesbar programmering» – utskrift side 4 . . . . .	79

---

# Tabeller

2.1	AlboCs biblioteksfunksjoner . . . . .	29
3.1	x86-instruksjoner brukt i prosjektet . . . . .	34
3.2	Assemblerdirektiver . . . . .	34
4.1	Kode for å hente en verdi inn i %EAX . . . . .	36
4.2	Kode for å hente en adresse inn i %EAX . . . . .	37
4.3	Kode generert av adresseoperand i uttrykk . . . . .	37
4.4	Kode generert av unære operatorer i uttrykk . . . . .	37
4.5	Kode generert av binære operatorer i uttrykk . . . . .	38
4.6	Kode generert av tom setning . . . . .	38
4.7	Kode generert av tilordning . . . . .	38
4.8	Kode generert av if-setning . . . . .	39
4.9	Kode generert av return-setning . . . . .	40
4.10	Kode generert av while-setning . . . . .	40
4.11	Kode generert av funksjonsdeklarasjon . . . . .	40
4.12	Kode generert av funksjonskall . . . . .	41
4.13	Kode generert av globale variabeldeklarasjoner . . . . .	41
6.1	Opsjoner for logging . . . . .	48
6.2	Typeregler for setninger . . . . .	53
6.3	Typeregler for uttrykk . . . . .	53
7.1	Suns forslag til navnevalg i Java-programmer . . . . .	69



# Forord

Dette kompendiet er laget for emnet *INF2100 – Prosjektoppgave i programmering*. Selve kurset er et av de eldste ved Ifi, men innholdet har allikevel blitt fornyet jevnlig.

Det opprinnelige kurset ble utviklet av *Stein Krogdahl* rundt 1980 og dreide seg om å skrive en kompilator som oversatte det Simula-lignende språket *Minila* til kode for en tenkt datamaskin *Flink*; implementasjonsspråket var Simula. I 1999 gikk man over til å bruke Java som implementasjonsspråk, og i 2007 ble kurset fullstendig renovert av *Dag Langmyhr*: Minila ble erstattet av en minimal variant av C kalt *RusC* og datamaskinen Flink ble avløst av en annen ikkeksisterende maskin kalt *Rask*. I 2010 ble det besluttet å lage ekte kode for Intel-prosessoren x86 slik at den genererte koden kunne kjøres direkte på en datamaskin. Dette medførte så store endringer i språket RusC at det fikk et nytt navn: *C<* (uttales «c less»). Ønsker om en utvidelse førte i 2012 til at det ble innført datatyper (int og double) og språket fikk igjen et nytt navn: *Cb* (uttales «c flat»).

Tilbakemelding fra studentene avslørte at de syntes det ble veldig mye fiklig å lage kode for double, så i 2014 ble språket endret enda en gang. Under navnet *AlboC* har det nå pekere i stedet for flyt-tall.

Målet for dette kompendiet er at det sammen med forelesningsplansjene skal gi studentene tilstrekkelig bakgrunn til å kunne gjennomføre prosjektet.

Forfatterne vil ellers takke studenten *Bendik Rønning Opstad* for verdifulle innspill om forbedringer av dette kompendiet og studentene *Einar Løvhøi*, *Antonsen*, *Marius Ekeberg*, *Arne Olav Hallingstad*, *Sigmund Hansen*, *Simen Heggestøy*, *Brendan Johan Lee*, *Håvard Koller Noren*, *Vegard Nossum*, *David J Oftedal*, *Mikael Olausson*, *Cathrine Elisabeth Olsen*, *Christian Andre Finnøy Ruud*, *Ryhor Sivuda*, *Herman Torjussen*, *Christian Tryti*, *Jørgen Vigdal*, *Olga Voronkova* og *Aksel L Webster* som har påpekt skrivefeil i tidligere utgaver. Om flere studenter finner feil, vil de også få navnet sitt på trykk.

Blindern, 8. august 2014  
*Stein Krogdahl      Dag Langmyhr*



*Teori er når ingenting virker og alle vet hvorfor. Praksis er når altting virker og ingen vet hvorfor.*

*I dette kurset kombineres teori og praksis – ingenting virker og ingen vet hvorfor.*

— Forfatterne

## Kapittel 1

# Innledning

### 1.1 Hva er emnet INF2100?

Emnet INF2100 har betegnelsen *Prosjektoppgave i programmering*, og hovedideen med dette emnet er å ta med studentene på et så stort programmeringsprosjekt som mulig innen rammen av de ti studiepoeng kurset har. Grunnen til at vi satser på ett stort program er at de fleste ting som har å gjøre med strukturering av programmer, oppdeling i moduler etc, ikke oppleves som meningsfylte eller viktige før programmene får en viss størrelse og kompleksitet. Det som sies om slike ting i begynnerkurs, får lett preg av litt livsfjern «programmeringsmoral» fordi man ikke ser behovet for denne måten å tenke på i de små oppgavene man vanligvis rekker å gå gjennom.

Ellers er programmering noe man trenger trening for å bli sikker i. Dette kurset vil derfor ikke innføre så mange nye begreper omkring programmering, men i stedet forsøke å befeste det man allerede har lært, og demonstrere hvordan det kan brukes i forskjellige sammenhenger.

«Det store programmet» som skal lages i løpet av INF2100, er en **kompilator**. En kompilator oversetter fra ett datamaskinspråk til et annet, vanligvis fra et såkalt **høy nivå programmeringsspråk** til et **maskinspråk** som datamaskinen elektronikk kan utføre direkte. Nedenfor skal vi se litt på hva en kompilator er og hvorfor det å lage en kompilator er valgt som tema for oppgaven.

Selv om vi konsentrerer dette kurset omkring ett større program vil ikke dette kunne bli noe virkelig *stort* program. Ute i den «virkelige» verden blir programmer fort vekk på flere hundre tusen eller endog millioner linjer, og det er først når man skal i gang med å skrive slike programmer, og ikke minst senere gjøre endringer i dem, at strukturen av programmene blir helt avgjørende. Det programmet vi skal lage i dette kurset vil typisk bli på tre-fire tusen linjer.

I dette kompendiet beskrives stort sett bare selve programmeringsoppgaven som skal løses. I tillegg til dette kan det komme ytterligere krav, for eksempel angående bruk av verktøy eller skriftlige arbeider som skal leveres. Dette vil i så fall bli opplyst om på forelesningene og på kursets hjemmeside.

## 1.2 Hvorfor lage en kompilator?

Når det skulle velges tema for en programmeringsoppgave til dette kurset, var det først og fremst to kriterier som var viktige:

- Oppgaven må være overkommelig å programmere innen kursets ti studiepoeng.
- Programmet må angå en problemstilling som studentene kjenner, slik at det ikke går bort verdifull tid til å forstå hensikten med programmet og dets omgivelser.

I tillegg til dette kan man ønske seg et par ting til:

- Det å lage et program innen et visst anvendelsesområde gir vanligvis også bedre forståelse av området selv. Det er derfor også ønskelig at anvendelsesområdet er hentet fra databehandling, slik at denne bivirkningen gir økt forståelse av faget selv.
- Problemområdet bør ha så mange interessante variasjoner at det kan være en god kilde til øvingsoppgaver som kan belyse hovedproblemstillingen.

Ut fra disse kriteriene synes ett felt å peke seg ut som spesielt fristende, nemlig det å skrive en kompilator, altså et program som oppfører seg omrent som en Java-kompilator eller en C-kompilator. Dette er en type verktøy som alle som har arbeidet med programmering, har vært borti, og som det også er verdifullt for de fleste å lære litt mer om.

Det å skrive en kompilator vil også for de fleste i utgangspunktet virke som en stor og uoversiktig oppgave. Noe av poenget med kurset er å demonstrere at med en hensiktsmessig oppsplitting av programmet i deler som hver tar ansvaret for en avgrenset del av oppgaven, så kan både de enkelte deler og den helheten de danner, bli høyst medgjørlig. Det er denne erfaringen, og forståelsen av hvordan slik oppdeling kan gjøres på et reelt eksempel, som er det viktigste studentene skal få med seg fra dette kurset.

Vi skal i neste avsnitt se litt mer på hva en kompilator er og hvordan den står i forhold til liknende verktøy. Det vil da også raskt bli klart at det å skrive en kompilator for et «ekte» programmeringsspråk som skal oversettes til maskinspråket til en datamaskin vil bli en altfor omfattende oppgave. Vi skal derfor forenkle oppgaven en del ved å lage vårt eget lille programmeringsspråk **AlboC**. Vi skal i det følgende se litt nærmere på dette og andre elementer som inngår i oppgaven.

## 1.3 Om kompilatorer og liknende verktøy

De fleste som starter på kurset INF2100, har neppe full oversikt over hva en kompilator er og hvilken funksjon den har i forbindelse med et programmeringsspråk. Dette vil forhåpentligvis bli mye klarere i løpet av dette kurset, men for å sette scenen skal vi gi en kort forklaring her.

## 1.3 OM KOMPILATORER OG LIKNENDE VERKTØY

---

Grunnen til at man i det hele tatt har komplilatorer, er at det er høyst upraktisk å bygge datamaskiner slik at de direkte utfra sin elektronikk kan utføre et program skrevet i et høynivå programmeringsspråk som for eksempel Java, C, C++ eller Perl. I stedet er datamaskiner bygget slik at de kan utføre et begrenset repertoar av nokså enkle instruksjoner, og det blir derved en overkommelig oppgave å lage elektronikk som kan utføre disse. Til gjengjeld kan datamaskiner raskt utføre lange sekvenser av slike instruksjoner, grovt sett med en hastighet av 1–3 milliarder instruksjoner per sekund.

For å kunne få utført programmer skrevet for eksempel i C, lages det spesielle programmer som kan oversette C-programmet til en tilsvarende sekvens av maskininstruksjoner for en gitt maskin. Det er slike oversettelsesprogrammer som kalles komplilatorer. En komplilator er altså et helt vanlig program som leser data inn og leverer data fra seg. Dataene det leser inn er et tekstlig program (i det programmeringsspråket denne komplilatoren skal oversette fra), og data det leverer fra seg er en sekvens av maskininstruksjoner for den aktuelle maskinen. Disse maskininstruksjonene vil komplilatoren vanligvis legge på en fil i et passelig format med tanke på at de senere kan kopieres inn i en maskin og bli utført.

Det settet med instruksjoner som en datamaskin kan utføre direkte i elektronikken, kalles maskinens **maskinspråk**, og programmer i dette språket kalles *maskinprogrammer* eller *maskinkode*.

En komplilator må også sjekke at det programmet den får inn overholder alle reglene for det aktuelle programmeringsspråket. Om dette ikke er tilfelle, må det gis feilmeldinger, og da lages det som regel heller ikke noe maskinprogram.

For å få begrepet *kompilator* i perspektiv skal vi se litt på et par alternative måter å ordne seg på, og hvordan disse skiller seg fra tradisjonelle komplilatorer.

### 1.3.1 Preprosessorer

I stedet for å komplilere til en sekvens av maskininstruksjoner finnes det også noen komplilatorer som oversetter til et annet programmeringsspråk på samme «nivå». For eksempel kunne man tenke seg å oversette fra Java til C++, for så å la en C++-komplilator oversette det videre til maskinkode. Vi sier da gjerne at denne Java-«komplilatoren» er en **preprosessor** til C++ komplilatoren.

Mest vanlig er det å velge denne løsningen dersom man i utgangspunktet vil bruke et bestemt programmeringsspråk, men ønsker noen spesielle utvidelser; dette kan være på grunn av en bestemt oppgave eller fordi man tror det kan gi språket nye generelle muligheter. En preprosessor behøver da bare ta tak i de spesielle utvidelsene, og oversette disse til konstruksjoner i grunnutgaven av språket.

Et eksempel på et språk der de første komplilatorene ble laget på denne måten, er C++. C++ var i utgangspunktet en utvidelse av språket C, og

utvidelsen besto i å legge til objektorienterte begreper (klasser, subklasser og objekter) hentet fra språket Simula. Denne utvidelsen ble i første omgang implementert ved en preprosessor som oversatte alt til ren C. I dag er imidlertid de fleste kompilatorer for C++ skrevet som selvstendige kompilatorer som oversetter direkte til maskinkode.

En viktig ulempe ved å bruke en preprosessor er at det lett blir tull omkring feilmeldinger og tolkningen av disse. Siden det programmet preprosessoren leverer fra seg likevel skal gjennom en full kompilering etterpå, lar man vanligvis være å gjøre en full programsjekk i preprosessoren. Dermed kan den slippe gjennom feil som i andre omgang resulterer i feilmeldinger fra den avsluttende kompileringen. Problemet blir da at disse vanligvis ikke vil referere til linjenumrene i brukerens opprinnelige program, og de kan også på andre måter virke nokså uforståelige for vanlige brukere.

### 1.3.2 Interpreting

Det er også en annen måte å utføre et program skrevet i et passelig programmeringsspråk på, og den kalles **interpreting**. I stedet for å skrive en kompilator som kan oversette programmer i det aktuelle programmetringsspråket til maskinspråk, skriver man en såkalt **interpreter**. Dette er et program som (i likhet med en kompilator) leser det aktuelle programmet linje for linje, men som i stedet for å produsere maskinkode rett og slett *gjør* det som programmet foreskriver skal gjøres.

Den store forskjellen blir da at en kompilator bare leser (og oversetter) hver linje én gang, mens en interpreter må lese (og utføre) hver linje på nytt hver eneste gang den skal utføres for eksempel i en løkke. Interpreting går derfor generelt en del tregere under utførelsen, men man slipper å gjøre noen kompilering. En del språk er (eller var opprinnelig) siktet spesielt inn på linje-for-linje-interpreting, det gjelder for eksempel Basic. Det finnes imidlertid nå kompilatorer også for disse språkene.

En type språk som nesten alltid blir interpretert, er **kommandospråk** til operativsystemer; ett slikt eksempel er Bash.

Interpreting kan gi en del fordeler med hensyn på fleksibel og gjenbrukbar kode. For å utnytte styrkene i begge teknikkene, er det laget systemer som kombinerer interpreting og kompilering. Noe av koden kompileres helt, mens andre kodebiter oversettes til et mellomnivåspråk som er bedre egnet for interpreting – og som da interpreteres under kjøring. Smalltalk, Perl og Python er eksempler på språk som ofte er implementert slik.

Interpreting kan også gi fordeler med hensyn til portabilitet, og, som vi skal se under, er dette utnyttet i forbindelse med vanlig implementasjon av Java.

### 1.3.3 Kompilering og kjøring av Java-programmer

En av de opprinnelige ideene ved Java var knyttet til datanett ved at et program skulle kunne kompiles på én maskin for så å kunne sendes over nettet til en hvilken som helst annen maskin (for eksempel som en

såkalt *applet*) og bli utført der. For å få til dette definerte man en tenkt datamaskin kalt *Java Virtual Machine* (JVM) og lot kompilatorene produsere maskinkode (gjerne kalt *byte-kode*) for denne maskinen. Det er imidlertid ingen datamaskin som har elektronikk for direkte å utføre slik byte-kode, og maskinen der programmet skal utføres må derfor ha et program som simulerer JVM-maskinen og dens utføring av byte-kode. Vi kan da gjerne si at et slikt simulatingsprogram interpreterer maskinkoden til JVM-maskinen. I dag har for eksempel de fleste nettlesere (Firefox, Opera, Explorer og andre) innebygget en slik JVM-interpreter for å kunne utføre Java-applets når de får disse (ferdig kompilert) over nettet.

Slik interpretering av maskinkode går imidlertid normalt en del saktere enn om man hadde oversatt til «ekte» maskinkode og kjørt den direkte på «hardware». Typisk kan dette for Javas byte-kode gå 1,2 til 2 ganger så sakte. Etter hvert som Java er blitt mer populært har det derfor også blitt behov for systemer som kjører Java-programmer raskere, og den vanligste måten å gjøre dette på er å utstyre JVM-er med såkalt «Just-In-Time» (JIT)-kompilering. Dette vil si at man i stedet for å interpretere byte-koden, oversetter den videre til den aktuelle maskinkoden umiddelbart før programmet startes opp. Dette kan gjøres for hele programmer, eller for eksempel for klasse etter klasse etterhvert som de tas i bruk første gang.

Man kan selvfølgelig også oversette Java-programmer på mer tradisjonell måte direkte fra Java til maskinkode for en eller annen faktisk maskin, og slike kompilatorer finnes og kan gi meget rask kode. Om man bruker en slik kompilator, mister man imidlertid fordelen med at det kompilerte programmet kan kjøres på alle systemer.

## 1.4 Språkene i oppgaven

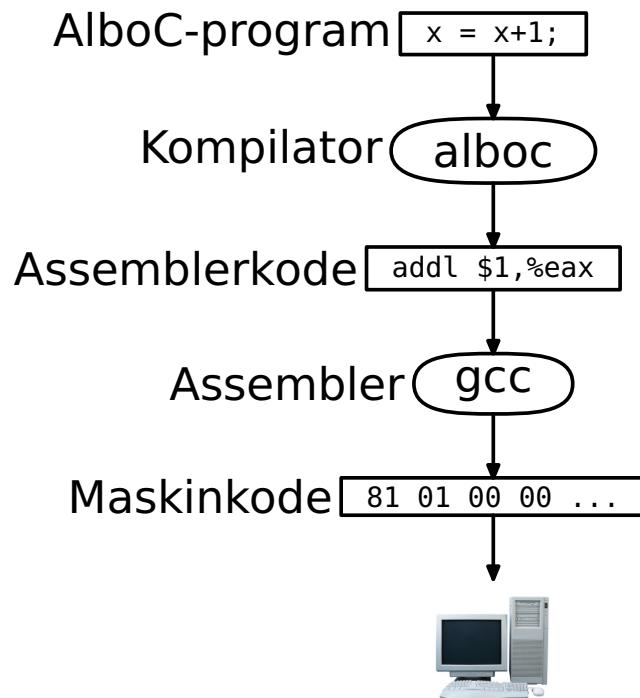
I løpet av dette prosjektet må vi forholde oss til flere språk.

### 1.4.1 Programmeringsspråket AlboC

Det å lage en kompilator for til dømes Java ville fullstendig sprengje kursrammen på ti studiepoeng. I stedet har vi laget et språk spesielt for dette kurset med tanke på at det skal være overkommelig å oversette. Dette språket er en miniversjon av C kalt *AlboC*. Selv om dette språket er enkelt, er det lagt vekt på at man skal kunne uttrykke seg rimelig fritt i det, og at «avstanden» opp til mer realistiske programmeringsspråk ikke skal virke uoverkommelig. Språket AlboC blir beskrevet i detalj i kapittel 2 på side 21.

### 1.4.2 Datamaskinen x86 og dens maskinspråk

En kompilator produserer vanligvis kode for en gitt prosessor og det skal også vår AlboC-kompilator gjøre. Som prosessor er valgt *x86* siden den finnes overalt, for eksempel på Ifis datalaber og i de aller fleste hjemmemaskiner. Dermed kan dere teste koden uansett hvor dere måtte befinne dere.



**Figur 1.1:** Sammenhengen mellom AlboC, kompilator, assembler og en x86-maskin

Emnet INF2100 vil langt fra gi noen full opplæring i denne prosessoren; vi vil kun ta for oss de delene av oppbygningen og instruksjonssettet som er nødvendig for akkurat vårt formål.

### 1.4.3 Assembleren

Når man skal programmere direkte i maskinstruksjoner, er det svært tungt å bruke tallkoder hele tiden, og så godt som alle maskiner har derfor laget en tekstkode som er lettere å huske enn et tall. For eksempel kan man bruke «`addl`» for en instruksjon som legger sammen lange (dvs 32-bits) heltall i stedet for til dømes tallet 129 (=  $81_{16}$ ), som kan være instruksjonens egentlige kode. Man lager så et enkelt program som oversetter sekvenser av slike tekstlige instruksjoner til utførbar maskinkode, og disse oversetterprogrammene kalles tradisjonelt **assemblerer**. Det oppsettet eller formatet man må bruke for å angi et maskinprogram til assembleren, kalles gjerne **assemblerspråket**.

### 1.4.4 Oversikt over de ulike språkene i oppgaven

Det blir i begynnelsen mange programmeringsspråk å holde orden på før man blir kjent med dem og hvordan de forholder seg til hverandre. Det er altså fire språk med i bildet, slik det er vist i figur 1.1:

- 1) **AlboC**, som kompilatoren skal oversette fra.
- 2) **Java**, som AlboC-kompilatoren skal skrives i.

- 3) **x86 assemblerkode** er en tekstlig form for maskininstruksjoner til x86-maskinen.
- 4) **x86s maskinspråk**, som assembleren skal oversette til.

## 1.5 Oppgaven og dens tre deler

Oppgaven skal løses i tre skritt, hvor alle er obligatoriske oppgaver. Som nevnt kan det utover dette komme krav om for eksempel verktøybruk eller levering av skriftlige tilleggsarbeider, men også dette vil i så fall bli annonseret i god tid.

Hele programmet kan grovt regnet bli på fra to til fire tusen Java-linjer, alt avhengig av hvor tett man skriver. Vi gir her en rask oversikt over hva de tre delene vil inneholde, men vi kommer fyldig tilbake til hver av dem på forelesningene og i senere kapitler.

### 1.5.1 Del 0

Første skritt, del 0, består i å få AlboCs **skanner** til å virke. Skanneren er den modulen som fjerner kommentarer fra programmet, og så deler den gjenstående teksten i en veldefinert sekvens av såkalte **symboler** (på engelsk «tokens»). Symbolene er de «ordene» programmet er bygget opp av, så som *navn*, *tall*, *nøkkelord*, '+', '>=' og alle de andre tegnene og tegnkombinasjonene som har en bestemt betydning i AlboC-språket.

Denne «renskårne» sekvensen av symboler vil være det grunnlaget som resten av kompilatoren (del 1) skal arbeide videre med. Mye av programmet til del 0 vil være ferdig laget eller skissert, og dette vil kunne hentes på angitt sted.

### 1.5.2 Del 1

Del 1 vil ta imot den symbolsekvensen som blir produsert av del 0, og det sentrale arbeidet her vil være å sjekke at denne sekvensen har den formen et riktig AlboC-program skal ha (altså, at den følger AlboCs **syntaks**).

Om alt er i orden, skal del 1 bygge opp et **syntakstre**, en **trestruktur** av objekter som direkte representerer det aktuelle AlboC-programmet, altså hvordan det er satt sammen av «expression» inne i «statement» inne i «function body» osv. Denne trestrukturen skal så leveres videre til del 2 som grunnlag for generering av x86-kode.

### 1.5.3 Del 2

I del 2 skal man sjekke variabler og funksjoner mot sine deklarasjoner og så gjøre selve oversettelsen til x86-kode; da tar vi utgangspunkt i den trestrukturen som del 1 produserte for det aktuelle AlboC-programmet. Koden skal legges på en fil og den skal være i såkalt x86 assemblerformat.

I kapittel 4 på side 35 er det angitt hvilke sekvenser av x86-instruksjoner hver enkelt AlboC-konstruksjon skal oversettes til, og det er viktig å merke

seg at disse skjemaene *skal* følges (selv om det i enkelte tilfeller er mulig å produsere lurere x86-kode; dette skal vi eventuelt se på i noen ukeoppgaver).

## 1.6 Programmering av lister, trær etc

Noe av hensikten med INF2100 er at man i størst mulig grad skal få en «hands on»-følelse med alle deler av programmeringen ned gjennom alle nivåer. Det er derfor et krav at gruppene selv programmerer all håndtering av lister og trær og ikke bruker ferdiglagde bibliotekspakker og slikt til det. Med andre ord, det er ikke lov å importere andre Java-klasser enn `java.lang.*` (som alltid er importert automatisk) og `java.io.*`. Det er heller ikke tillatt å benytte seg av Javas **StreamTokenizer** eller andre **-Tokenizer**-klasser.

For de som nettopp har tatt introduksjonskursene, kan dette kanskje være en utfordring, men vi skal bruke noe tid i gruppene til å se på dette, og ut fra eksempler, oppgaver, etc burde det da gå greit.

## 1.7 Krav til samarbeid og gruppetilhørighet

Normalt er det meningen at to personer skal samarbeide om å løse oppgaven. De som samarbeider bør være fra samme øvingsgruppe på kurset. Man bør tidlig begynne å orientere seg for å finne én på gruppen å samarbeide med. Det er også lov å løse oppgaven alene, men dette vil selvfølgelig gi mer arbeid. Om man har en del programmeringserfaring, kan imidlertid dette være et overkommelig alternativ.

Hvis man får samarbeidsproblemer (som at den andre «har meldt seg ut» eller «har tatt all kontroll»), si fra i tide til gruppelærer eller kursledelse, så kan vi se om vi kan hjelpe dere å komme over «krisen». Slik har skjedd før.

## 1.8 Kontroll av innlevert arbeid

For å ha en kontroll på at hvert arbeidslag har programmert og testet ut programmene på egen hånd, og at begge medlemmene har vært med i arbeidet, må studentene være forberedt på at gruppelæreren eller kursledelsen forlanger at studenter som har arbeidet sammen, skal kunne redegjøre for oppgitte deler av den komplilatoren de har skrevet. Med litt støtte og hint skal de for eksempel kunne gjenskape deler av selve programmet på en tavle.

Slik kontroll vil bli foretatt på stikkprøvebasis samt i noen tilfeller der gruppelæreren har sett lite til studentene og dermed ikke har hatt kontroll underveis med studentenes arbeid.

Dessverre har vi tidligere avslørt fusk; derfor ser vi det nødvendig å holde slike overhøringer på slutten av kurset. Dette er altså ingen egentlig eksamen, bare en sjekk på at dere har gjort arbeidet selv. Noe ekstra arbeid for dem som blir innkalt, blir det heller ikke. Når dere har programmert og

## 1.9 DELTA PÅ ØVINGSGRUPPENE

---

testet ut programmet, kan dere kompilatoren deres forlengs, baklengs og med bind for øynene.

Et annet krav er at alle innleverte programmer er vesentlig forskjellig fra alle andre innleveringer. Men om man virkelig gjør jobben selv, får man automatisk et unikt program.

Hvis noen er engstelige for hvor mye de kan samarbeide med andre utenfor sin gruppe, vil vi si:

- Ideer og teknikker kan diskuteres fritt.
- Programkode skal gruppene skrive selv.

Eller sagt på en annen måte: Samarbeid er bra, men kopiering er galt!

Merk at *ingen godkjenning av enkeltdeler er endelig* før den avsluttende runden med slik muntlig kontroll, og denne blir antageligvis holdt en gang rundt begynnelsen av desember.

## 1.9 Delta på øvingsgruppene

Ellers vil vi oppfordre studentene til å være aktive på de ukentlige øvingsgruppene. Oppgavene som blir gjennomgått, er meget relevante for skriving av AlboC-kompilatoren. Om man tar en liten titt på oppgavene før gruppetime, vil man antagelig få svært mye mer ut av gjennomgåelsen.

På gruppa er det helt akseptert å komme med et uartikulert:

«Jeg forstår ikke hva dette har med saken å gjøre!»

Antageligvis føler da flere det på samme måten, så du gjør gruppa en tjeneste. Og om man synes man har en aha-opplevelse, så det fin støtte både for deg selv og andre om du sier:

«Aha, det er altså ... som er poenget! Stemmer det?»

Siden det er mange nye begreper å komme inn i, er det viktig å begynne å jobbe med dem så tidlig som mulig i semesteret. Ved så å ta det fram i hodet og oppfriske det noen ganger, vil det neppe ta lang tid før begrepene begynner å komme på plass. Kompendiet sier ganske mye om hvordan oppgaven skal løses, men alle opplysninger om hver programbit står ikke nødvendigvis samlet på ett sted.

Til sist et råd fra tidligere studenter: *Start i tide!*



## Kapittel 2

# Programmering i språket AlboC

Programmeringsspråket **AlboC** er en miniversjon av C; navnet står for «A little bit of C». Syntaksen er gitt av jernbanediagrammene i figur 2.2 til 2.37 på side 23–29 og bør være lett forståelig for alle som har programmert litt i C. Et eksempel på et AlboC-program er vist i figur 2.1 på neste side.<sup>1</sup>

### 2.1 Kjøring

Inntil dere selv har laget en AlboC-kompilator, kan dere benytte referanse-kompilatoren:

```
$ ~inf2100/alboc easter.alboc
This is the AlboC compiler (version 2014-08-08 on Linux)
Parsing... checking... generating code... OK
Running gcc -m32 -o easter easter.s -L. -L/hom/inf2100 -lalboc
$ ./easter
4 April 2010
24 April 2011
8 April 2012
31 March 2013
20 April 2014
5 April 2015
27 March 2016
16 April 2017
1 April 2018
21 April 2019
12 April 2020
```

#### 2.1.1 Kompilering med C-kompilatoren

Siden AlboC er en nesten ekte undermengde av C, kan man også bruke C-kompilatoren til å lage kjørbar kode. Det eneste man må sørge for, er å ta med AlboC-biblioteket `-lalboc`:<sup>2</sup>

---

<sup>1</sup> Du finner kildekoden til dette programmet og også andre nyttige testprogrammer i mappen `~inf2100/oblig/test/` på alle Ifi-maskiner; mappen er også tilgjengelig fra en vilkårlig nettleser som <http://inf2100.at.ifi.uio.no/oblig/test/>.

<sup>2</sup> Man bør unngå `#`-linjer når man skal benytte C-kompilatoren; de har en annen betydning i C.

```

1  /* Test program 'easter'
2   =====
3   Computes Easter Sunday for the years 2010-2020.
4 */
5
6  int mod (int x, int y)
7  { /* Computes x%y */
8      return x - (x/y*y);
9  }
10
11 int easter (int y)
12 {
13     int a;  int b;  int c;  int d;  int e;  int f;
14     int g;  int h;  int i;  int k;  int l;  int m;
15
16     int month; /* The date of Easter Sunday */
17     int m_name[5];
18     int day;
19
20     int ix;
21
22     a = mod(y,19);
23     b = y / 100;
24     c = mod(y,100);
25     d = b / 4;
26     e = mod(b,4);
27     f = (b+8) / 25;
28     g = (b-f+1) / 3;
29     h = mod(19*a+b-d-g+15,30);
30     i = c / 4;
31     k = mod(c,4);
32     l = mod(32+2*e+2*i-h-k,7);
33     m = (a+11*h+22*l) / 451;
34
35     month = (h+l-(7*m)+114) / 31;
36     day = mod(h+l-(7*m)+114,31) + 1;
37     if (month == 3) {
38         m_name[0] = 'M'; m_name[1] = 'a'; m_name[2] = 'r';
39         m_name[3] = 'c'; m_name[4] = 'h';
40     } else {
41         m_name[0] = 'A'; m_name[1] = 'p'; m_name[2] = 'r';
42         m_name[3] = 'i'; m_name[4] = 'l';
43     }
44
45     /* Print the answer: */
46     putint(day); putchar(' ');
47     for (ix = 0; ix < 5; ix = ix+1) { putchar(m_name[ix]); }
48 }
49
50 int main ()
51 {
52     int y;
53
54     for (y = 2010; y <= 2020; y = y+1) {
55         easter(y); putchar(' ');
56         putint(y); putchar(10);
57     }
58 }
```

**Figur 2.1:** Eksempel på et AlboC-program

```

$ gcc -m32 -o e -x c easter.alboc -L./hom/inf2100 -lalboc
$ ./e
4 April 2010
24 April 2011
8 April 2012
31 March 2013
20 April 2014
5 April 2015
27 March 2016
16 April 2017
1 April 2018
21 April 2019
12 April 2020

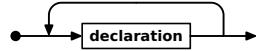
```

## 2.2 AlboC-program

Som vist i figur 2.2 på neste side er et AlboC-program rett og slett en samling funksjoner (kjent som «metoder» i Java). Før, mellom og etter disse

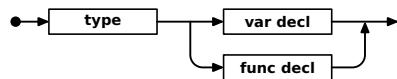
funksjonene kan man deklarere globale variabler.

### program



**Figur 2.2:** Jernbanediagram for {program}

### declaration



**Figur 2.3:** Jernbanediagram for {declaration}

For at vi skal ha et kjørbart program, må det eksistere en int-funksjon med navnet **main**, og programutførelsen starter alltid med å kalle denne funksjonen. Funksjonen **main** kan ikke ha noen parametre.

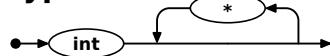
#### 2.2.1 Variabler

Brukeren kan deklarere enten globale variabler eller variabler som er lokale i en funksjon. Alle variabler har angitt en **type** som forteller hva slags verdier de kan lagre. En type i språket vårt kan være int (et heltall), int\* (peker til et heltall), int\*\* (peker til peker til et heltall) osv.

Det er også mulig å deklarere en **array**; arrayer indekseres fra 0 (som i C og Java) og indeksen må alltid være en int. Antallet elementer i en arraydeklarasjon må være en heltallskonstant.

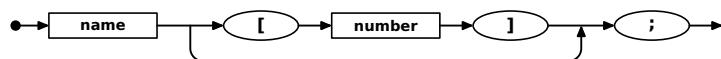
Det er ikke mulig å angi noen **initialverdi** for variabler – de inneholder en ukjent verdi før de tilordnes en verdi av programmet.

### type



**Figur 2.4:** Jernbanediagram for {type}

### var decl



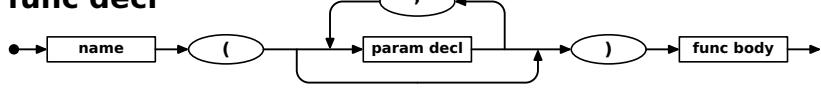
**Figur 2.5:** Jernbanediagram for {var decl}

#### 2.2.2 Funksjoner

Brukeren kan deklarere funksjoner med vilkårlig mange parametre; parametrene kan ikke være arrayer. Parameteroverføringen skjer ved kopiering (som i C og Java).

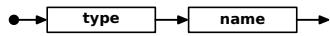
Funksjoner deklarereres som int- eller pekerfunksjoner, dvs at de returnerer en int- eller en pekerverdi. Hvis ingen eksplisitt verdi er angitt (med en return-setning), returneres en tilfeldig verdi.

### **func decl**



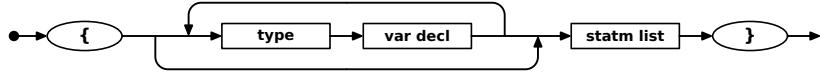
**Figur 2.6:** Jernbanediagram for `<func decl>`

### **param decl**



**Figur 2.7:** Jernbanediagram for `<param decl>`

### **func body**



**Figur 2.8:** Jernbanediagram for `<func body>`

### **statm list**

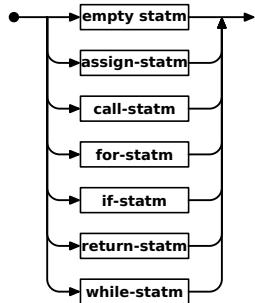


**Figur 2.9:** Jernbanediagram for `<statm list>`

## 2.2.3 Setninger

AlboC kjenner til syv ulike setninger.

### **statement**



**Figur 2.10:** Jernbanediagram for `<statement>`

### 2.2.3.1 Tomme setninger

En tom setning (der det ikke står noe foran semikolonet) er lov i AlboC. Naturlig nok gjør den ingenting.

#### **empty statm**



**Figur 2.11:** Jernbanediagram for `<empty statm>`

### 2.2.3.2 Tilordninger

En tilordning lagrer en verdi i minnet. Avsnitt 6.5.1.4 på side 52 angir hvilke tilordninger som er lov.

### assign-stmt



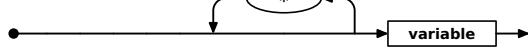
Figur 2.12: Jernbanediagram for {assign-stmt}

### assignment



Figur 2.13: Jernbanediagram for {assignment}

### lhs-variable



Figur 2.14: Jernbanediagram for {lhs-variable}

#### 2.2.3.3 Funksjonskall

Et funksjonskall kan brukes som en egen setning.

### call-stmt



Figur 2.15: Jernbanediagram for {call-stmt}

#### 2.2.3.4 for-setninger

En for-setning er en slags utvidelse av while-setningen: i tillegg til slutt-testen har den en initieringsdel som utføres aller først og en oppdateringsdel som utføres etter hvert gjennomløp av løkken.

### for-stmt



Figur 2.16: Jernbanediagram for {for-stmt}

### for-control

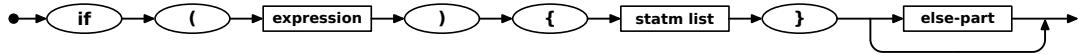


Figur 2.17: Jernbanediagram for {for-control}

#### 2.2.3.5 if-setninger

Disse setningene brukes til å velge om noen setninger skal utføres eller ikke. Selve testen er et uttrykk som beregnes: verdien 0 (enten den er en int eller en peker) angir usann, mens alle andre verdier angir sann.

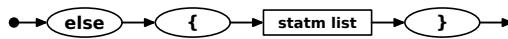
### if-stmt



Figur 2.18: Jernbanediagram for {if-stmt}

if-tester kan utstyres med en else-gren når man vil angi et alternativ.

### else-part



**Figur 2.19:** Jernbanediagram for {else-part}

### 2.2.3.6 return-setninger

En slik setning avslutter utførelsen av en funksjon og angir samtidig returverdien.

### return-stattm



**Figur 2.20:** Jernbanediagram for {return-stattm}

### 2.2.3.7 while-setninger

En while-setning går i løkke inntil testuttrykket (som kan være enten et heltall eller en peker) beregnes til 0.

### while-stattm



**Figur 2.21:** Jernbanediagram for {while-stattm}

## 2.2.4 Uttrykk

Uttrykk i AlboC er en delmengde av de vi kjenner fra C og Java. Definisjonen er delt i {expression}, {term}, {factor}, {primary} og {operand} slik at vi kan håndtere operatorenes **presedens**<sup>3</sup> riktig.

Følgende er verdt å merke seg om uttrykk:

- Symbolet - kan bety enten unær minus (som i  $-a$ ) eller binær minus (som i  $a - 1$ ) avhengig av bruken.
- Det samme gjelder symbolet \*, men her er betydningsforskjellen større: som vanlig operator (i for eksempel  $2 * a$ ) angir det multiplikasjon mens brukt som prefiksoperator (som i  $*p$ ) betyr det å følge pekeren.

### expression



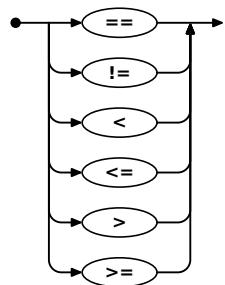
**Figur 2.22:** Jernbanediagram for {expression}

<sup>3</sup> Operatorer har ulik presedens, dvs at noen operatorer binder sterkere enn andre. Når vi skriver for eksempel

$$a + b \times c$$

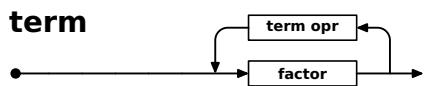
tolkes dette vanligvis som  $a + (b \times c)$  fordi  $\times$  normalt har høyere presedens enn  $+$ , dvs  $\times$  binder sterkere enn  $+$ .

**rel opr**



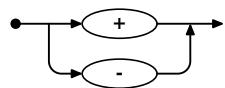
**Figur 2.23:** Jernbanediagram for {rel opr}

**term**



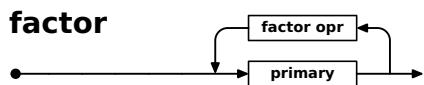
**Figur 2.24:** Jernbanediagram for {term}

**term opr**



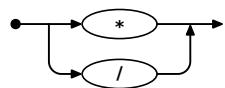
**Figur 2.25:** Jernbanediagram for {term opr}

**factor**



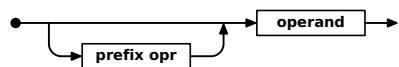
**Figur 2.26:** Jernbanediagram for {factor}

**factor opr**



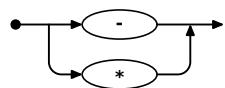
**Figur 2.27:** Jernbanediagram for {factor opr}

**primary**



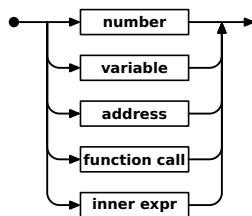
**Figur 2.28:** Jernbanediagram for {primary}

**prefix opr**



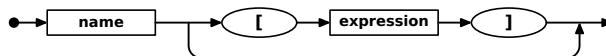
**Figur 2.29:** Jernbanediagram for {prefix opr}

### operand



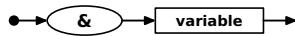
**Figur 2.30:** Jernbanediagram for `{operand}`

### variable



**Figur 2.31:** Jernbanediagram for `{variable}`

### address



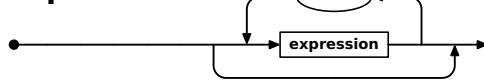
**Figur 2.32:** Jernbanediagram for `{address}`

### function call



**Figur 2.33:** Jernbanediagram for `{function call}`

### expr list



**Figur 2.34:** Jernbanediagram for `{expr list}`

### inner expr



**Figur 2.35:** Jernbanediagram for `{inner expr}`

#### 2.2.4.1 Biblioteket

AlboC kjenner til fem biblioteksfunksjoner som er vist i tabell 2.1 på neste side. Disse kan brukes uten noen spesiell spesifikasjon.

#### 2.2.4.2 Navn

Navn på variabler og funksjoner kan være vilkårlig lange og består av store eller små bokstaver, sifre eller «understrekning» (dvs tegnet `«_»`).

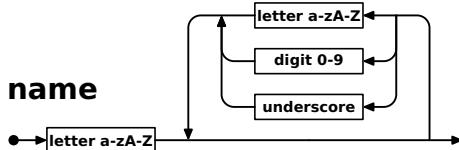
Store og små bokstaver regnes som ulike, så `Per` og `per` er altså to forskjellige navn.

#### 2.2.4.3 Tall og tegn

Heltall brukes som forventet i AlboC. I tillegg er det lov å bruke tegnkonstanter (men altså ikke tekster på mer enn ett tegn). Slike

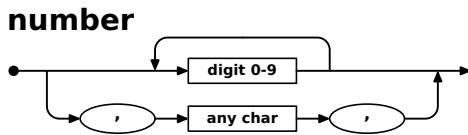
Funksjon	Effekt
int exit (int status)	Avslutter med angitt statusverdi.
int getchar ()	Leser neste tegn fra tastaturet.
int getint ()	Leser neste heltall fra tastaturet.
int putchar (int c)	Skriver et tegn på skjermen.
int putint (int c)	Skriver et heltall på skjermen.

Tabell 2.1: AlboCs biblioteksfunksjoner



Figur 2.36: Jernbanediagram for `<name>`

tegnkonstanter betraktes som tall der verdien er tegnets representasjon i ASCII.



Figur 2.37: Jernbanediagram for `<number>`

## 2.2.5 Andre ting

Det er et par andre ting man bør merke seg ved AlboC:

- Man kan bare benytte variabler og funksjoner deklarert tidligere i programmet.
- Kommentarlinjer har et «#» som aller første tegn; da skal hele linjen ignoreres.
- Kommentarer kan også angis som «/\*...\*/» og kan da strekke seg over flere linjer.



## Kapittel 3

# Datamaskinen x86

Om vi åpner en datamaskin, ser vi at det store hovedkortet er fylt med elektronikk av mange slag; se figur 3.1 på neste side. I denne omgang<sup>1</sup> er vi bare interessert i prosessoren og minnet.

### 3.1 Minnet

Minnet består av tre deler:

**Datadelen** brukes til å lagre globale variabler.

**Stakken** benyttes til parametre, lokale variabler, mellomresultater og diverse systeminformasjon.

**Kodedelen** inneholder programkoden, altså programmet som utføres.

### 3.2 Prosessoren x86

x86-prosessoren er en 32-bits<sup>2</sup> prosessor som inneholder fire viktige deler:

**Logikkenheten** tolker instruksjonene; med andre ord utfører den programkoden. I tabell 3.1 på side 34 er vist de instruksjonene vi vil benytte i prosjektet vårt.

**Regneenheten** kan de fire regneartene for heltall og kan dessuten sammenligne slike verdier.

**Registrene** er spesielle heltallsvariabler som er ekstra tett koblet til regneenheten. Vi skal bruke disse registrene:

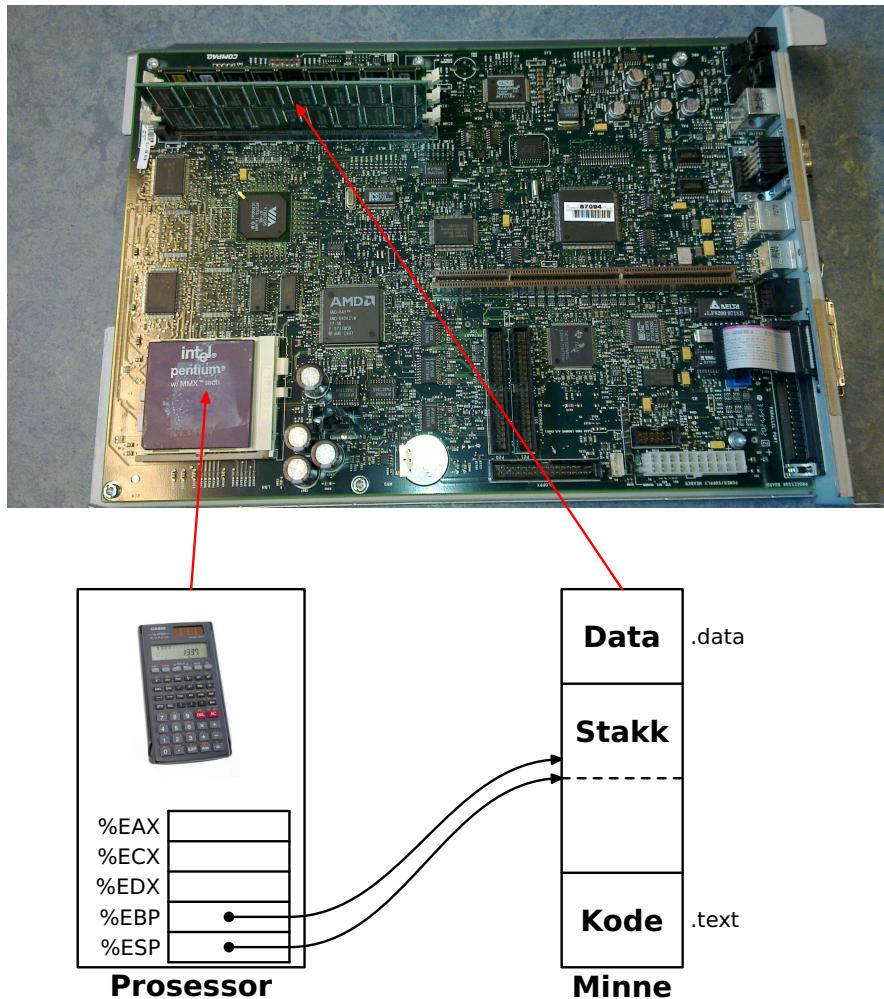
%EAX %ECX %EDX %EBP %ESP

%ESP («extended stack pointer») peker på (dvs inneholder adressen til) toppen av stakken, mens %EBP («extended base pointer») peker på lokale variabler og funksjonsparametre; de andre registrene er stort sett til regning.

---

<sup>1</sup> Dette kapittelet er ingen utfyllende beskrivelse av hvordan en datamaskin er bygget opp – det forteller bare akkurat det vi trenger å vite for å skrive kompilatoren vår.

<sup>2</sup> Dagens prosessorer er oftest av typen x64 som er en 64-bits utvidelse av x86, men de er i stand til å kjøre x86-kode.



**Figur 3.1:** Hovedkortet med prosessor og minne i en datamaskin

### 3.3 Assemblerkode

Assemblerkode er en meget enkel form for kode: instruksjonene skrives én og én på hver linje slik det er vist i figur 3.2.

<u>func:</u>	<u>movl</u>	<u>\$0,%eax</u>	<u># Initier til 0.</u>
Navnelapp	Instruksjon	Parametre	Kommentar

**Figur 3.2:** Instruksjonslinje i assemblerkode

**Navnelapp** («label») gir et navn til instruksjonen.

**Instruksjon** er en av instruksjonene i tabell 3.1 på side 34.

**Parametre** angir data til instruksjonen; antallet avhenger av instruksjonen.  
Vi vil bruke disse parametrene:

**%EAX** er et register.

**\$17** er en tallkonstant.

**minvar** er navnet på en global variabel eller en funksjon.

**8(%ESP)** angir en lokal variabel eller en parameter (se avsnitt 4.1.2 på side 35).

**Kommentarer** ignoreres.

Alle de fire elementene kan være med eller utelates i alle kombinasjoner; man kan for eksempel ha kun en navnelapp på en linje, eller bare en kommentar. Helt blanke linjer er også lov.

#### 3.3.1 Assemblerdirektiver

I tillegg til programkode vil assemblerkode alltid inneholde noen **direktiver** som er en form for beskjeder til assembleren. Vi skal bruke de direktivene som er vist i tabell 3.2 på neste side.

<code>movl</code>	<code>&lt;v1&gt;, &lt;v2&gt;</code>	Flytt <code>&lt;v1&gt;</code> til <code>&lt;v2&gt;</code> .
<code>cdq</code>		Omform 32-bits %EAX til 64-bits %EDX:%EAX.
<code>leal</code>	<code>&lt;v1&gt;, &lt;v2&gt;</code>	Flytt <code>&lt;v1&gt;</code> s adresse til <code>&lt;v2&gt;</code> .
<code>pushl</code>	<code>&lt;v&gt;</code>	Legg <code>&lt;v&gt;</code> på stakken.
<code>popl</code>	<code>&lt;v&gt;</code>	Fjern toppen av stakken og legg verdien i <code>&lt;v&gt;</code> .
<code>negl</code>	<code>&lt;v&gt;</code>	Skift fortegn på <code>&lt;v&gt;</code> .
<code>addl</code>	<code>&lt;v1&gt;, &lt;v2&gt;</code>	Adder <code>&lt;v1&gt;</code> til <code>&lt;v2&gt;</code> .
<code>subl</code>	<code>&lt;v1&gt;, &lt;v2&gt;</code>	Subtraher <code>&lt;v1&gt;</code> fra <code>&lt;v2&gt;</code> .
<code>imull</code>	<code>&lt;v1&gt;, &lt;v2&gt;</code>	Multipliser <code>&lt;v1&gt;</code> med <code>&lt;v2&gt;</code> .
<code>idivl</code>	<code>&lt;v&gt;</code>	Del %EDX:%EAX med <code>&lt;v&gt;</code> ; svar i %EAX.
<code>call</code>	<code>&lt;lab&gt;</code>	Kall funksjonen i <code>&lt;lab&gt;</code> .
<code>enter</code>	<code>\$&lt;n&gt;, \$0</code>	Start en funksjon med <code>&lt;n&gt;</code> byte lokale variabler.
<code>leave</code>		Rydd opp når funksjonen er ferdig.
<code>ret</code>		Returner fra funksjonen.
<code>cmpl</code>	<code>&lt;v1&gt;, &lt;v2&gt;</code>	Sammenligning <code>&lt;v1&gt;</code> og <code>&lt;v2&gt;</code> .
<code>jmp</code>	<code>&lt;lab&gt;</code>	Hopp til <code>&lt;lab&gt;</code> .
<code>je</code>	<code>&lt;lab&gt;</code>	Hopp til <code>&lt;lab&gt;</code> hvis <code>=</code> .
<code>sete</code>	<code>&lt;v&gt;</code>	Sett <code>&lt;v&gt;</code> =1 om <code>=</code> , ellers <code>&lt;v&gt;</code> =0.
<code>setne</code>	<code>&lt;v&gt;</code>	Sett <code>&lt;v&gt;</code> =1 om <code><math>\neq</math></code> , ellers <code>&lt;v&gt;</code> =0.
<code>setl</code>	<code>&lt;v&gt;</code>	Sett <code>&lt;v&gt;</code> =1 om <code>&lt;</code> , ellers <code>&lt;v&gt;</code> =0.
<code>setle</code>	<code>&lt;v&gt;</code>	Sett <code>&lt;v&gt;</code> =1 om <code><math>\leq</math></code> , ellers <code>&lt;v&gt;</code> =0.
<code>setg</code>	<code>&lt;v&gt;</code>	Sett <code>&lt;v&gt;</code> =1 om <code>&gt;</code> , ellers <code>&lt;v&gt;</code> =0.
<code>setge</code>	<code>&lt;v&gt;</code>	Sett <code>&lt;v&gt;</code> =1 om <code><math>\geq</math></code> , ellers <code>&lt;v&gt;</code> =0.

**Tabell 3.1:** x86-instruksjoner brukt i prosjektet. Følgende symboler er brukt i tabellen:

- (v) kan være en konstant («\$17»), et register («%EAX»), en global variabel («Var»), en lokal variabel («-4(%EBP)») eller en parameter(«8(%EBP)»).
- (n) er en heltallskonstant.
- (lab) er en merkelapp som angir en minnelokasjon.

<code>.data</code>	Angi at vi nå skal plassere variabler i data-delen av minnet.
<code>(lab): .fill n,4,0</code>	Sett av $4n$ byte i minnet (til en variabel eller en array).
<code>.globl &lt;lab&gt;</code>	Navnet <code>&lt;lab&gt;</code> skal være kjent utenfor denne filen.
<code>.text</code>	Angi at vi nå skal plassere instruksjoner i kodedelen av minnet.

**Tabell 3.2:** Assemblerdirektiver

# Kapittel 4

# Kodegenerering

## 4.1 Konvensjoner

Når vi skal generere kode, er det en stor fordel å være enige om visse ting, for eksempel registerbruk.

### 4.1.1 Registrer

Vi vil bruke disse registrene:

**%EAX** er det viktigste arbeidsregisteret. Alle uttrykk eller deluttrykk skal produsere et resultat i %EAX.

**%ECX** er et hjelprepregister som brukes ved aritmetiske eller sammenligningsoperatorer eller til indeks ved oppslag i arrayer.

**%EDX** brukes til arrayadresser og som hjelprepregister ved tilordning og divisjon.

**%ESP** peker på toppen av kjørestakken.

**%EBP** peker på den aktuelle funksjonens parametre og lokale variabler.

### 4.1.2 Navn

I utgangspunktet ønsker vi å benytte samme navn i assemblerkoden som brukeren har valgt i sin AlboC-kode, men det er ikke alltid mulig.

**Funksjoner** beholder sitt AlboC-navn.<sup>1</sup> Alle funksjoner må angi hvor de slutter; til det vil vi bruke **.exit\$f** i funksjonen *f*.

**Globale variabler** får også beholdne navnet brukt i AlboC-koden.<sup>1</sup>

**Parametre** trenger ikke navn i assemblerkoden siden de er gitt utfra posisjonen i parameterlisten: Første parameter er 8(%ebp), andre parameter 12(%ebp), tredje parameter 16(%ebp) osv.

---

<sup>1</sup> Når man benytter gcc under Mac eller Windows, må globale navn ha en «\_» foran. Det er ikke nødvendig å implementere det i prosjektet selv om det er gjort i referansekomplilatoren.

**Lokale variabler** trenger heller ikke navn siden de også ligger på stakken.

Nøyaktig hvor de ligger på stakken må kompilatoren vår regne seg frem til; dette avhenger av de andre lokale variablene i samme funksjon.

**Ekstra navn** har vi behov for når assemblerkoden skal hoppe i løkker og annet. De får navn `.L0001`, `.L0002`, osv.

## 4.2 Oversettelse av uttrykk

Hovedregelen når vi skal lage kode for å beregne uttrykk, er at resultatet av alle uttrykk og deluttrykk skal ende opp i `%EAX`.

### 4.2.1 Operander i uttrykk

I tabell 4.1 er vist hvilken kode som må genereres for å hente en verdi  $\langle n \rangle$ , en enkel variabel  $\langle v \rangle$ , et arrayelement  $\langle a \rangle[\langle e \rangle]$  eller et uttrykk i parenteser  $(\langle e \rangle)$  inn i register `%EAX`.

(Kode for funksjonskall er ikke tatt med her – dette er beskrevet i avsnitt 4.4.2 på side 41.)

$\langle n \rangle$	$\Rightarrow$	<code>movl \$n,%eax</code>
$\langle v \rangle$	$\Rightarrow$	<code>movl v,%eax</code>
$\langle a \rangle[\langle e \rangle]$	$\Rightarrow$	<p><math>\langle</math>Beregn <math>\langle e \rangle</math> med svar i <code>%EAX</code><math>\rangle</math></p> <pre>leal a,%edx movl (%edx,%eax,4),%eax</pre>
$(\langle e \rangle)$	$\Rightarrow$	$\langle$ Beregn $\langle e \rangle$ med svar i <code>%EAX</code> $\rangle$

**Tabell 4.1:** Kode for å hente en verdi inn i `%EAX`

Tabell 4.2 på neste side viser tilsvarende hvilken kode som kan brukes til å legge en adresse i register `%EAX`.

#### 4.2.1.1 Adresseoperand

Denne operanden, som er angitt med en «`& w`», skal legge adressen til  $\langle w \rangle$  inn i `%EAX`.  $\langle w \rangle$  kan være en vanlig variabel  $\langle v \rangle$ , et arrayelement  $\langle a \rangle[\langle e \rangle]$  eller en hel array  $\langle a \rangle$ . Koden er vist i tabell 4.3 på neste side.

## 4.2 OVERSETTELSE AV UTTRYKK

$\langle v \rangle$	$\Rightarrow$	leal $\langle v \rangle, %eax$
$\langle a \rangle$	$\Rightarrow$	leal $\langle a \rangle, %eax$
$\langle a \rangle[\langle e \rangle]$	$\Rightarrow$	{Beregner $\langle e \rangle$ med svar i %EAX} leal $\langle a \rangle, %edx$ leal $(%edx, %eax, 4), %eax$

**Tabell 4.2:** Kode for å hente en adresse inn i %EAX

$\& \langle w \rangle$	$\Rightarrow$	{Beregner adressen til $\langle w \rangle$ med svar i %EAX}
------------------------	---------------	---

**Tabell 4.3:** Kode generert av adresseoperand i uttrykk

### 4.2.2 Operatorer i uttrykk

#### 4.2.2.1 Unære operatorer

Tabell 4.4 viser hvordan vi skal oversette de unære operatorene.

$- \langle e \rangle$	$\Rightarrow$	{Beregner $\langle e \rangle$ med svar i %EAX} negl $%eax$
$* \langle e \rangle$	$\Rightarrow$	{Beregner $\langle e \rangle$ med svar i %EAX} movl $(%eax), %eax$

**Tabell 4.4:** Kode generert av unære operatorer i uttrykk

#### 4.2.2.2 Binære operatorer

I tabell 4.5 på neste side er vist hvordan de binære operatorene +, / (som trenger litt annen kode enn de andre regneoperatorene) og == skal oversettes. De øvrige finner du sikkert selv.

$\langle e_1 \rangle + \langle e_2 \rangle$	$\Rightarrow$	(Beregn $\langle e_1 \rangle$ med svar i %EAX) pushl %eax (Beregn $\langle e_2 \rangle$ med svar i %EAX) movl %eax,%ecx popl %eax addl %ecx,%eax
$\langle e_1 \rangle / \langle e_2 \rangle$	$\Rightarrow$	(Beregn $\langle e_1 \rangle$ med svar i %EAX) pushl %eax (Beregn $\langle e_2 \rangle$ med svar i %EAX) movl %eax,%ecx popl %eax cdq idivl %ecx
$\langle e_1 \rangle == \langle e_2 \rangle$	$\Rightarrow$	(Beregn $\langle e_1 \rangle$ med svar i %EAX) pushl %eax (Beregn $\langle e_2 \rangle$ med svar i %EAX) popl %ecx cmpl %eax,%ecx movl \$0,%eax sete %al

**Tabell 4.5:** Kode generert av binære operatorer i uttrykk

## 4.3 Oversettelse av setninger

### 4.3.1 Oversettelse av tomme setninger

Dette er den enkleste setningen å oversette, som vist i tabell 4.6.

;	$\Rightarrow$	
---	---------------	--

**Tabell 4.6:** Kode generert av tom setning

### 4.3.2 Oversettelse av tilordningssetninger

Kodegenerering for slike setninger er vist i tabell 4.7. Husk at  $n$  kan være 0 og at  $\langle w \rangle$  kan være enten en vanlig variabel  $\langle v \rangle$  eller et arrayelement  $\langle a \rangle[\langle e \rangle]$ .

$*^n \langle w \rangle = \langle e \rangle ;$	$\Rightarrow$	(Beregn adressen til $\langle w \rangle$ med svar i %EAX) movl (%eax),%eax gjetas $n$ ganger pushl %eax (Beregn $\langle e \rangle$ med svar i %EAX) popl %edx movl %eax,(%edx)
---	---------------	--

**Tabell 4.7:** Kode generert av tilordning

### 4.3.3 Oversettelse av kallsetninger

En kallsetning er bare et funksjonskall og oversettes på akkurat samme måte; se avsnitt 4.4.2 på side 41.

### 4.3.4 Oversettelse av for-setninger

Denne oversettelsen må du finne frem til selv. Det er klart det skal være en løkke, og det viktigste er å finne ut hvor de tre elementene

- 1) initieringstilordning
- 2) testuttrykk
- 3) oppdateringstilordning

skal stå i forhold til løkken.

### 4.3.5 Oversettelse av if-setninger

Tabell 4.8 viser oversettelse av en if-setning, både uten og med en else-gren.

if ( $\{e\}$ ) $\{\{S\}\}$	$\Rightarrow$	$\langle \text{Beregn } \{e\} \text{ med svar i \%EAX} \rangle$ $\text{cmpl } \$0, \%eax$ $\text{je } \langle \text{lab} \rangle$ $\langle S \rangle$ $\langle \text{lab} \rangle :$
if ( $\{e\}$ ) $\{\{S_1\}\}$ else $\{\{S_2\}\}$	$\Rightarrow$	$\langle \text{Beregn } \{e\} \text{ med svar i \%EAX} \rangle$ $\text{cmpl } \$0, \%eax$ $\text{je } \langle \text{lab}_1 \rangle$ $\langle S_1 \rangle$ $\text{jmp } \langle \text{lab}_2 \rangle$ $\langle \text{lab}_1 \rangle :$ $\langle S_2 \rangle$ $\langle \text{lab}_2 \rangle :$

**Tabell 4.8:** Kode generert av if-setning

### 4.3.6 Oversettelse av return-setninger

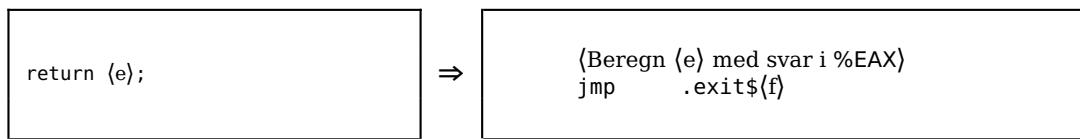
En return-setning innebærer to ting:

- 1) Resultatverdien beregnes.
- 2) Eksekveringen skal hoppe til slutten av funksjonen.

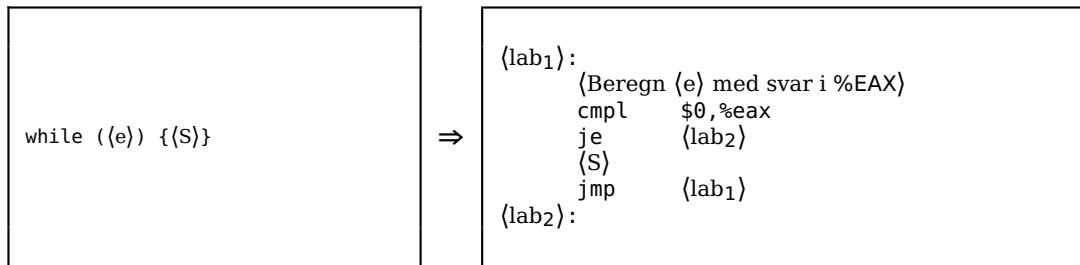
Koden for dette er vist i tabell 4.9 på neste side. (I avsnitt 4.4.2 på side 41 står det hvor slutten er plassert.)

### 4.3.7 Oversettelse av while-setninger

Oversettelse av en while-setning innebærer å lage en løkke og en løkketest; dette er vist i tabell 4.10 på neste side.



**Tabell 4.9:** Kode generert av return-setning



**Tabell 4.10:** Kode generert av while-setning

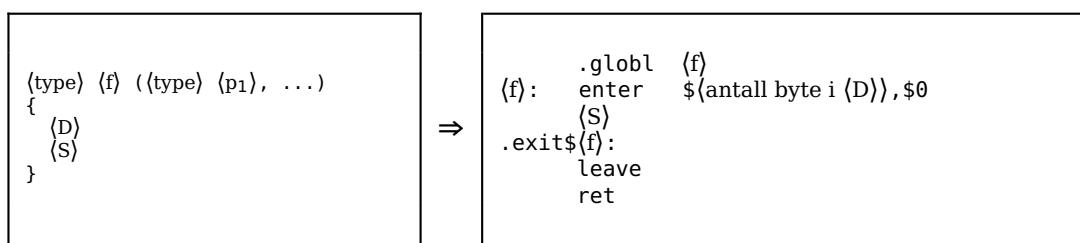
## 4.4 Oversettelse av funksjoner og funksjonskall

Når vi skal oversette funksjoner, må vi se på hvordan den enkelte funksjonsdekklarasjonen oversettes og hva som skjer ved kall på den.

### 4.4.1 Oversettelse av funksjonsdeklarasjoner

Som vist i figur 4.11 legger vi inn litt fast kode i begynnelsen og slutten av funksjonen. Legg også merke til at:

- Vi legger inn en `.globl`-spesifikasjon, så kode i andre filer kan kalle på denne funksjonen.
- Parametrene resulterer ikke i noe kode siden de skal ligge på stakken når funksjonen kalles.
- Instruksjonen `enter` setter av plass til lokale variabler på stakken; for å finne ut hvor mange byte vi skal sette av, må vi summere hvor mange byte hver enkelt lokal variabel tar.
- Vi legger inn en etikett ved uthoppet slik at return-setninger har et sted å hoppe til; se avsnitt 4.3.6 på forrige side. Vi må også bruke `leave`-instruksjonen til å frigjøre plassen vi satte av til lokale variabler før vi hopper tilbake med en `ret`.



**Tabell 4.11:** Kode generert av funksjonsdeklarasjon

#### 4.4.2 Oversettelse av funksjonskall

Slik tabell 4.12 viser, oversettes et funksjonskall til tre kodesekvenser:

- 1) Parametrene legges på stakken (i *omvendt rekkefølge*).
- 2) Funksjonen kalles.
- 3) Parametrene fjernes fra stakken.

I eksemplet har funksjonen to parametre, så 8 byte må fjernes fra stakken etterpå. Det bør være enkelt å generalisere dette til å ha et vilkårlig antall parametre, inkludert 0.

(f)((e <sub>1</sub> ), (e <sub>2</sub> ))	⇒	{Bereg e <sub>2</sub> med svar i %EAX} pushl %eax {Bereg e <sub>1</sub> med svar i %EAX} pushl %eax call f addl \$8,%esp
---	---	---

**Tabell 4.12:** Kode generert av funksjonskall

### 4.5 Deklarasjon av variabler

#### 4.5.1 Deklarasjon av globale variabler

Vi setter av plass til globale variabler med en `.fill`-spesifikasjon. Slike variabler legges i data-segmentet. De får også en `.globl`-spesifikasjon, slik at de blir virkelig globale.

Vanlige enkle variabler er 4 byte lange, så de deklarerdes som vist i tabell 4.13. Til arrayer setter vi av 4 byte til hvert element.

(type) (v);	⇒	(v): .globl (v) .fill 1,4,0
(type) (a)[(n)];	⇒	(a): .globl (a) .fill (n),4,0

**Tabell 4.13:** Kode generert av globale variabeldeklarasjoner

##### 4.5.1.1 Angivelse av minnedel

Vi må plassere variablene i datadelen av minnet mens instruksjonene skal i kodedelen. Derfor må vi holde rede på hvilken del av minnet vi jobber mot for øyeblikket.

- Om vi skal generere en variabeldeklarasjon etter å ha laget instruksjoner, må vi sette inn en

.data

for å skifte til datadelen av minnet.

- Om vi skal skrive ut en instruksjon etter å ha produsert variabel-deklarasjoner, må vi ta med en

.text

for å skifte til kodedelen av minnet.

#### **4.5.2 Deklarasjon av lokale varabler**

Funksjonen sørger selv for å sette av plass til sine lokale varabler på stakken (se tabell 4.11 på side 40).

#### **4.5.3 Deklarasjon av parametre**

Siden parametre legges på stakken ved et funksjonskall, trenger de ingen deklarasjon i den genererte assemblerkoden.

# Kapittel 5

# Implementasjonen

Store programmer (og også middelstore programmer) bør deles i passe store **moduler** når de skal implementeres. Langt fra alle programmeringsspråk tilbyr noen slik mekanisme, men Java gjør det i form av **Java-pakker** angitt med nøkkelordet **package**. Vi skal bruke denne mekanismen, og i tråd med Javas navnetradisjon skal våre pakker hete «no.uio.ifi.alboc.error» og tilsvarende.

Vi skal dele prosjektet vårt i pakkene vist i figur 5.1 på neste side. Siden Java kun tillater klasser i pakkene sine, vil vi alltid legge inn en klasse med samme navn<sup>1</sup> som pakken. Denne klassen inneholder data og metoder som «hører hjemme i» pakken, spesielt de to metodene<sup>2</sup> **init** og **finish** som benyttes for initiering og terminering av modulene. Hver pakke vil altså se ut som vist i figur 5.2 på neste side.

## 5.1 Modulen alboc

Denne modulen inneholder **main**-metoden og er dermed «hovedmodulen». Den vil initiere de andre modulene, tolke kommandoparametrene, starte kompileringen, kjøre assembleren `gcc` og til sist terminere de andre modulene.

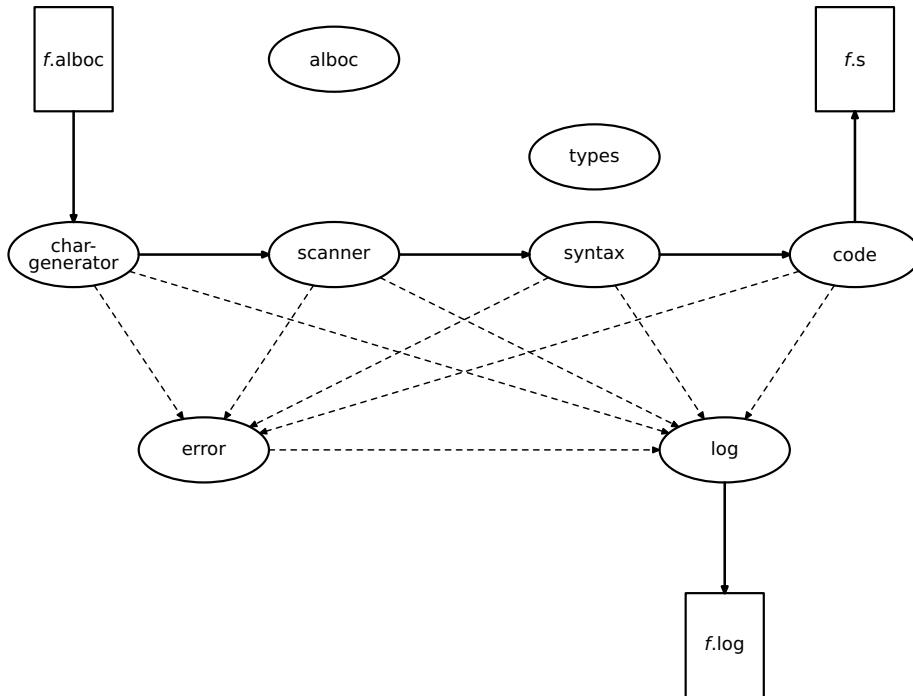
## 5.2 Modulen chargenerator

Denne modulen vil lese kildefilen linje for linje, ignorere #-linjer og sende den resterende kildekoden tegn for tegn videre i to variabler: **curC** (med nåværende tegn) og **nextC** (med neste tegn). Metoden **readNext** vil klargjøre neste tegn.

---

<sup>1</sup> Den eneste forskjellen på navnene er kapitaliseringen – Java-pakker skal helst ha liten forbokstav, mens Java-klasser bør ha stor.

<sup>2</sup> Disse metodene vil være **static** siden vi aldri skal instansiere disse spesialklassene.



**Figur 5.1:** Modulene i kompilatoren

```

package no.uio.ifi.alboc.p;

public class P {
    public static {data-deklasjon};
    private static {data-deklasjon};

    public static void init() {
        :
    }

    public static void finish() {
        :
    }

    :
}
  
```

**Figur 5.2:** Oppsett for de enkelte Java-pakkene

### 5.3 Modulen scanner

Denne modulen vil få tegn fra kildefilen (via chargenerator) og levere fra seg symboler i variablene **curToken** og **nextToken**; disse variablene er av klassen **Token**. **curToken** er det aktuelle symbolet vi skal analysere, mens **nextToken** er det etterfølgende symbolet. Variablene **curLine** og **nextLine** vil inneholde linjenummeret for de tilsvarende symbolene.

Om **curToken** er et **nameToken**, vil **curName** inneholde det aktuelle navnet, og om det er et **numberToken**, vil **curNum** inneholde tallverdien. Det tilsvarende gjelder for **nextName** og **nextNum**.

Metoden **readNext** vil plassere de neste symbolene i **curToken** og **nextToken**. Alle `/*...*/`-kommentarer vil bli ignorert.

Når det ikke er flere symboler igjen på filen, vil **curToken**-variabelen få verdien **eofToken**.<sup>3</sup>

## 5.4 Modulen syntax

Denne modulen tar seg av analysen av AlboC-programmet; slik analyse kalles **parsering** (på engelsk «parsing»). Inndata til analysen er symbolene som scanner-modulen lager og utdata skal være et **parseringstre** (ofte kalt **syntakstre**) som er en representasjon av brukerens program. Programmet `mini.alboc` i figur 6.2 på side 49 har for eksempel et syntakstre som vist i figur 5.3 på neste side.<sup>4</sup>

Modulen inneholder klassen **SyntaxUnit** og diverse subklasser som brukes når parseringstreet skal bygges. Alle implementerer disse metodene:<sup>5</sup>

**parse** vil parsere akkurat denne noden i treeet og eventuelle subtrær.

**printTree** vil skrive ut den delen av programmet som er representert av denne noden og eventuelle subtrær.

**check** vil koble alle navn i denne noden og eventuelle subtrær til sine deklarasjoner og sjekke at de er brukt riktig.

**genCode** vil generere ferdig kode for denne noden og eventuelle subtrær.

## 5.5 Modulen types

Denne modulen inneholder det som trengs for å jobbe med typer. Den inneholder disse klassene (i tillegg til Types):

**Type** er «moderklassen» til alle type-klassene.

**ValueType** representerer de typene som AlboC tillater kopiering av (i vårt tilfelle `int` og pekere).

**ArrayType** er for arrayer.

**PointerType** er for pekere.

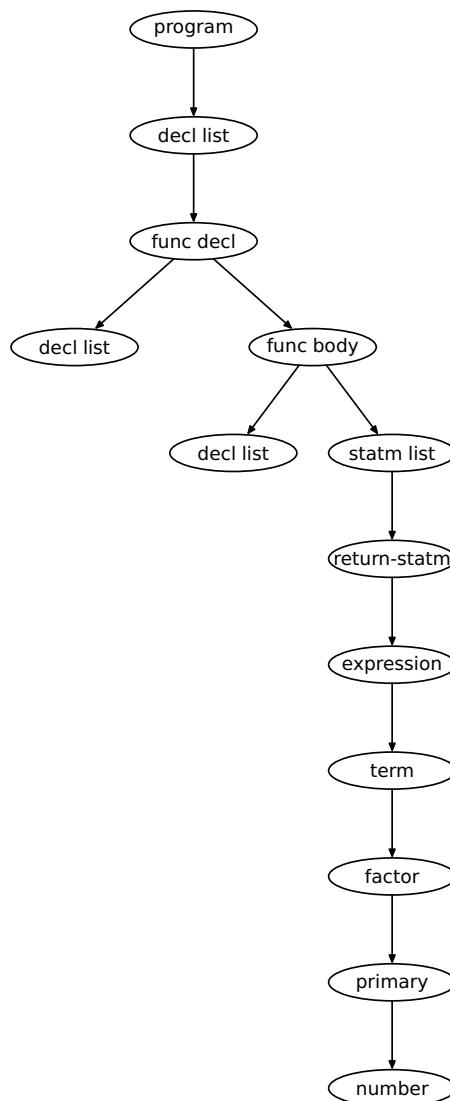
I tillegg inneholder Types variabelen `intType` som er laget av en anonym subklasse av `ValueType`.

---

<sup>3</sup> «Eof» er en vanlig forkortelse for «end of file».

<sup>4</sup> Det er ikke gitt at syntakstre til `mini.alboc` skal se nøyaktig slik ut – det vil være små variasjoner avhengig av hvilke klasser den enkelte definerer i komplilatoren.

<sup>5</sup> Legg merke til at `printTree`, `check` og `checkCode` er **virtuelle** metoder som kan betraktes som ulike implementasjoner av den samme metoden. Dette gjelder ikke `parse` siden alle `parse`-metodene returnerer verdier av ulik klasse.



**Figur 5.3:** Parseringstre for programmet i figur 6.2 på side 49

## 5.6 Modulen code

Denne modulen inneholder nyttige metoder for kodegenereringen.

## 5.7 Modulen error

Denne modulen brukes for feilutskrifter. Den har én sentral metode med navn **error** som skriver en feilmelding på skjermen og i loggfilen før den avbryter kompileringen ved å utløse en **feil** («exception»).

## 5.8 Modulen log

Denne modulen tar seg av logging av informasjon. Den skriver data på filen øyeblinkelig og lukker filen etter hver linje slik at innholdet bevares om kompilatoren krasjer.

# Kapittel 6

# Prosjektet

Som nevnt er AlboC-kompilatoren et større program enn dere sannsynligvis har skrevet før, så prosjektet er delt i tre deler: en minimal introduksjonsdel og to omrent like store restdeler, som vist i figur 6.1 på neste side.

På emnets nettside ligger 2100-oblig.zip som er rammen som *skal* brukes til løsningen. Lag en egen mappe til prosjektet deres og legg ZIP-filen der. Gjør så dette:

```
$ cd mappen  
$ unzip inf2100-oblig.zip  
$ ant
```

Dette vil resultere i en kjørbar fil Alboc.jar som kan kjøres slik

```
$ java -jar Alboc.jar minfil.alboc
```

men vær oppmerksom på at den utleverte koden selvfølgelig ikke vil fungere! Denne er bare en basis du må utvikle til et nyttig program.

Som en hjelp under arbeidet, og for enkelt å sjekke om de ulike delene virker, skal koden kunne håndtere loggutskriftene vist i tabell 6.1 på neste side.

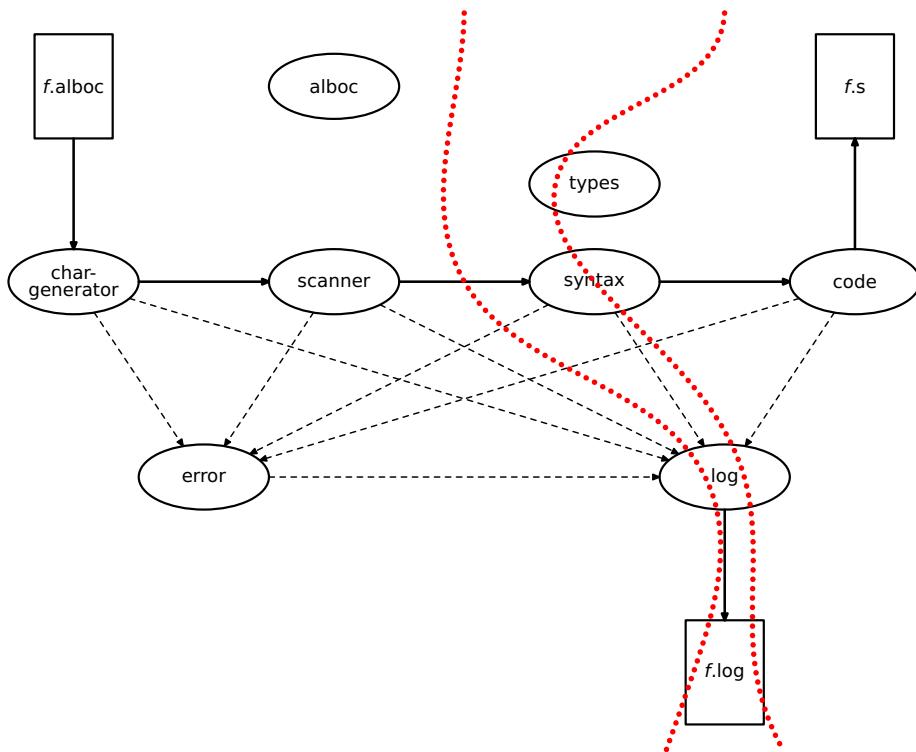
## 6.1 På egen datamaskin

Prosjektet er utviklet på Universitetets Linux-maskiner, men det er også mulig å gjennomføre programmeringen på egen datamaskin, uansett om den kjører Linux, Mac OS X eller Windows. Det er imidlertid ditt ansvar at nødvendige verktøy fungerer skikkelig. Du trenger:

**ant** er en overbygning til Java-kompilatoren; den gjør det enkelt å kompile et system med mange Java-filer. Den kan hentes ned fra <http://ant.apache.org/bindownload.cgi>.

**gas** er assembler. Den lastes gjerne ned sammen med C-kompilatoren **gcc**; se <http://gcc.gnu.org/install/download.html>.

**java** er en Java-interpret (ofte omtalt som «JVM» (Java virtual machine) eller «Java RTE» (Java runtime environment)). Om du installerer javac (se neste punkt), får du alltid med **java**.

**Del 0****Del 1****Del 2****Figur 6.1:** De ulike delene i prosjektet

Opsjon	Del	Hva logges
<code>-logB</code>	Del 2	Hvordan navnene bindes
<code>-logI</code>	Del 1	Utskrift av parseringstreet
<code>-logP</code>	Del 1	Hvilke parseringsmetoder kalles
<code>-logS</code>	Del 0	Hvilke symboler hentes fra skanneren
<code>-logT</code>	Del 2	Typesjekkingen

**Tabell 6.1:** Opsjoner for logging

**javac** er en Java-kompilator; du trenger Java *SE development kit* som kan hentes fra <http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html> eller nyere utgaver.

**Et redigeringsprogram** etter eget valg. Selv foretrekker jeg *emacs* som kan hentes fra <http://www.gnu.org/software/emacs/>, men du kan bruke akkurat hvilket du vil.

## 6.2 Tegnsett

I dag er det spesielt tre tegnkodinger som er i vanlig bruk i Norge:

```

1 # Program 'mini'
2 # -----
3 # A minimal Alboc program!
4
5 int main ()
6 {
7     return 0;
8 }
```

**mini.alboc****Figur 6.2:** Et minimalt Alboc-program mini.alboc

**ISO 8859-1** (også kalt «Latin-1») er et tegnsett der hvert tegn lagres i én byte.

**ISO 8859-15** (også kalt «Latin-9») er en lett modernisert variant av ISO 8859-1.

**UTF-8** er en lagringsform for **Unicode**-kodingen og bruker 1–4 byte til hvert tegn.

Siden dette med tegnsett lett kan gi mange forvirrende feilsituasjoner men ikke er noen viktig del av prosjektet, vil vi i dette kurset bare benytte tegn fra ASCII; disse tegnene er identiske i alle tre tegnkodingene.

## 6.3 Del 0

Del 0 er ment som en introduksjon og dreier seg om å få skanneren til å fungere. For å sjekke dette, kan vi gi opsjonen **-testscanner** (som også slår på logging slik **-logS** gjør):

```
$ java -jar Alboc.jar -testscanner mini.alboc
$ java -jar Alboc.jar -testscanner gcd.alboc
```

der det første testprogrammet er vist i figur 6.2 og det andre i 6.9 på side 54. De resulterende filene **mini.log** og **gcd.log** skal da se ut som vist<sup>1</sup> i henholdsvis figur 6.3 på neste side og figurene 6.10 og 6.11 på side 55 og siden etter.

### Mål for del 0

**Programmet skal utvikles slik at opsjonen -testscanner produserer loggfiler som vist i figurene 6.3 og 6.10–6.11.**

## 6.4 Del 1

Denne delen går ut på å få parseren til å fungere slik at den kan generere syntakstreet. Dette innebærer å skrive alle klassene som tilsvarer metasymbolene og metodene **parse** og **printTree**. Ved å kjøre

```
1 > java -jar Alboc.jar -testparser gcd.alboc
```

<sup>1</sup> Siden loggutskriften kommer fra to kilder (chargenerator og scanner), vil linjene fra disse kunne være blandet på en annen måte enn vist i dette kompendiet. Dette er helt normalt og klart akseptabelt.

## KAPITTEL 6 PROSJEKTET

---

```
1: # Program 'mini'  
2: # -----  
3: # A minimal AlboC program!  
4:  
5: int main ()  
6: Scanner: intToken  
7: Scanner: nameToken main  
8: Scanner: leftParToken  
9: Scanner: rightParToken  
10: 6: {  
11: Scanner: leftCurlToken  
12: 7: return 0;  
13: Scanner: returnToken  
14: Scanner: numberToken 0  
15: Scanner: semicolonToken  
16: 8: }  
17: Scanner: rightCurlToken  
18: Scanner: eofToken
```

---

**Figur 6.3:** Loggfil som demonstrerer hvilke symboler skaneren finner i `mini.alboc`

```
1: # Program 'mini'  
2: # -----  
3: # A minimal AlboC program!  
4:  
5: int main ()  
6: Parser: <program>  
7: Parser: <type>  
8: Parser: </type>  
9: Parser: <func decl>  
10: 6: {  
11: Parser: <func body>  
12: Parser: <statm list>  
13: Parser: <statement>  
14: Parser: <return-statm>  
15: Parser: <expression>  
16: Parser: <term>  
17: Parser: <factor>  
18: Parser: <primary>  
19: Parser: <operand>  
20: Parser: <number>  
21: Parser: 8: }  
22: Parser: </number>  
23: Parser: </operand>  
24: Parser: </primary>  
25: Parser: </factor>  
26: Parser: </term>  
27: Parser: </expression>  
28: Parser: </return-statm>  
29: Parser: </statement>  
30: Parser: </statm list>  
31: Parser: </func body>  
32: Parser: </func decl>  
33: Parser: </program>
```

---

**Figur 6.4:** Loggfil som viser parsering av `mini.alboc`

vil loggfilen vise hvilke **parse**-rutiner som man er innom (opsjonen `-logP`, som settes automatisk av `-testparser`); som kontroll skrives den interne representasjonen av programmet ut (opsjonen `-logI`, som også settes av `-testparser`). Våre vanlige testprogram som tidligere er vist i henholdsvis figur 6.2 på forrige side og figur 6.9 på side 54, vil produsere loggfilene i henholdsvis figur 6.4 og figurene 6.12 til 6.17 på side 57 og etterfølgende.<sup>2</sup>

---

<sup>2</sup> Denne loggutskriften stammer også fra flere kilder, så det er også her helt OK at linjene stokkes om i forhold til det som vises.

```

1 Tree:    int main ()
2 Tree:    {
3 Tree:        return 0;
4 Tree:    }

```

**Figur 6.5:** Loggfil med trerrepresentasjonen av mini.alboc

```
1 Binding: main refers to declaration in line 5
```

**Figur 6.6:** Loggfil med navnebinding for mini.alboc**Mål for del 1**

**Programmet skal implementere parsing og også utskrift av det lagrede programmet; med andre ord skal oppsjonen -testparser gi utskrift som vist i figurene 6.4–6.5 og 6.12–6.18.**

## 6.5 Del 2

Den siste delen er å få sjekkingen og kodegenereringen på plass. Dette kan sjekkes på tre måter:

- I mappen `~inf2100/oblig/test/` (som også er tilgjengelig fra en nettleser som <http://inf2100.at.ifi.uio.no/oblig/test/>) finnes noen AlboC-programmer som bør fungere i den forstand at de ikke gir feilmeldinger, men genererer riktig kode; resultatet av kjøringene skal dessuten gi resultatet vist i `.res`-filene.
- I mappen `~inf2100/oblig/feil/` (som også er tilgjengelig utenfor Ifi som <http://inf2100.at.ifi.uio.no/oblig/feil/>) finnes diverse småprogrammer som alle inneholder en feil. Kompilatoren din bør gi en tilsvarende feilmelding som referansekompiletoren.
- Den genererte assemblerkoden for våre standard testprogram (vist i figur 6.2 og figur 6.9) skal se ut som vist i henholdsvis figur 6.8 på neste side og figurene 6.21 til 6.22 på side 64–65. (Kommentarene på slutten av hver linje kan droppes, eller de kan se ut som du selv ønsker.)

**Mål for del 2**

**Kompilatoren skal foreta navnebindinger og kunne produsere data om dette som vist i figur 6.6 og 6.19; gal navnebruk og typefeil skal gi feilmeldinger. Dessuten skal kompilatoren generere en kodelinje som vist i figur 6.8 og 6.21–6.22. Det er ikke nødvendig å implementere -logT.**

### 6.5.1 Sjekking

Del 2 skal sjekke fire ting, og dette gjøres ved å traversere hele syntakstreet med metoden `check`; noen av testene gjøres på vei utover i treeet og noen på vei tilbake.

## KAPITTEL 6 PROSJEKTET

---

```
1 Checking types: Line 7: return e; in int f(...),  
2 where Type(e) is int
```

---

**Figur 6.7:** Loggfil med typesjekk for mini.alboc

```
1 .globl main  
2 main: enter $0,$0          # Start function main  
3     movl $0,%eax           # 0  
4     jmp .exit$main         # Return-statement  
5 .exit$main:  
6     leave  
7     ret                   # End function main
```

---

**Figur 6.8:** Kodefil laget fra mini.alboc

### 6.5.1.1 Sjekke navn ved deklarasjoner

Det må sjekkes at navn er deklarert riktig. I AlboC er dette enkelt, for det er bare én mulig feil: å deklarere navn flere ganger i samme liste.<sup>3</sup>

### 6.5.1.2 Sjekke navnebruk

Dette innebærer å se på alle navneforekomster og så finne hvilke deklarasjoner som definerer navnet; en referanse til deklarasjonen settes inn i Variable.declRef og i et passende element du selv deklarerer i klassen FunctionCall.

Så må det sjekkes om navnene er brukt riktig, for eksempel sjekke om brukeren har benyttet et variabelnavn for å kalle en funksjon eller et funksjonsnavn for å finne et arrayelement. Dette gjøres ved å definere og kalle på checkWhetherArray og tilsvarende i Declaration-klassen eller utvidelser av denne.

### 6.5.1.3 Bestemme typer

For alle uttrykk og deluttrykk må vi finne hvilken type de har; dette settes inn i Operator.opType og Operand.valType. (Alle deklarasjoner har fått satt sin Declaration.type i del 1.) Tabell 6.3 på neste side angir hvilken type hvert deluttrykk skal få.

### 6.5.1.4 Sjekke typer

For å kunne si hvilke typekrav vi stiller til ulike setninger og uttrykk, kan det være lurt å lage en egen «typematematikk». Her inngår noen spesielle symboler:

$\mathcal{T}_x$  betegner typen til en variabel eller et uttrykk  $x$ .

$\mathcal{A}$  er alle arraytypene.

$\mathcal{P}$  er alle pekertypene.

$\mathcal{V}$  angir alle typer som har verdier; i AlboC er dette `int` og pekere.

$*\mathcal{T}_x$  angir den typen som  $\mathcal{T}_x$  peker på eller inneholder.

---

<sup>3</sup> Et lite hint: Selv om denne testen egentlig skal gjøres gjennom metoden `check`, er det mye enklere å gjøre dette under parseringen i del 1, nærmere bestemt i metoden `DeclList.addDecl`.

	Krav
;	—
v = e;	$\mathcal{T}_v \in \mathcal{V} \wedge (\mathcal{T}_v = \mathcal{T}_e \vee \mathcal{T}_e = \text{int})$
for(...; t; ...)	$\mathcal{T}_t \in \mathcal{V}$
if(t) ...	$\mathcal{T}_t \in \mathcal{V}$
return e; in function t f(...) ...	$\mathcal{T}_e = \mathcal{T}_t \vee \mathcal{T}_e = \text{int}$
while(t) ...	$\mathcal{T}_t \in \mathcal{V}$

**Tabell 6.2:** Typeregler for setninger

	Krav	Resultat
x == y (også !=)	$\mathcal{T}_x \in \mathcal{V} \wedge \mathcal{T}_y \in \mathcal{V} \wedge (\mathcal{T}_x = \mathcal{T}_y \vee \mathcal{T}_x = \text{int} \vee \mathcal{T}_y = \text{int})$	int
x < y (også <=, >, >=)	$\mathcal{T}_x = \text{int} \wedge \mathcal{T}_y = \text{int}$	int
x + y (også -, *, /)	$\mathcal{T}_x = \text{int} \wedge \mathcal{T}_y = \text{int}$	int
- x	$\mathcal{T}_x = \text{int}$	int
* x	$\mathcal{T}_x \in \mathcal{P}$	$^*\mathcal{T}_x$
f(p, ...)	$\mathcal{T}_p = \mathcal{T}_{\text{formell param}} \vee \mathcal{T}_p = \text{int}$	se def av f
& a[e]	$\mathcal{T}_a \in \mathcal{A} \wedge \mathcal{T}_e = \text{int}$	$\text{ptr}(^*\mathcal{T}_a)$
& v	—	$\text{ptr}(\mathcal{T}_v)$
a[e]	$(\mathcal{T}_a \in \mathcal{A} \vee \mathcal{T}_a \in \mathcal{P}) \wedge \mathcal{T}_e = \text{int}$	$^*\mathcal{T}_a$
v	—	$\mathcal{T}_v$
2 (og andre heltall)	—	int

**Tabell 6.3:** Typeregler for uttrykk

$\wedge$  betyr «og».

$\vee$  betyr «eller».

```
1 # Program 'gcd'  
2 # -----  
3 # A program to compute the greatest common divisor.  
4  
5 int LF; /* Line feed */  
6  
7 int gcd (int a, int b)  
8 { /* Computes the gcd of a and b. */  
9  
10    while (a != b) {  
11        if (a < b) {  
12            b = b-a;  
13        } else {  
14            a = a-b;  
15        }  
16    }  
17    return a;  
18 }  
19  
20 int main ()  
21 {  
22    int v1; int v2;  
23  
24    LF = 10; ;  
25    putchar('?'); putchar(' ');  
26    v1 = getInt(); v2 = getInt();  
27    putInt(gcd(v1,v2)); putchar(LF);  
28    exit(0);  
29 }
```

gcd.alboc

**Figur 6.9:** Et litt større AlboC-program gcd.alboc

---

```

1: # Program 'gcd'
2: # -----
3: # A program to compute the greatest common divisor.
4:
5: 5: int LF; /* Line feed */
6: Scanner: intToken
7: Scanner: nameToken LF
8: Scanner: semicolonToken
9:   6:
10:    7: int gcd (int a, int b)
11: Scanner: intToken
12: Scanner: nameToken gcd
13: Scanner: leftParToken
14: Scanner: intToken
15: Scanner: nameToken a
16: Scanner: commaToken
17: Scanner: intToken
18: Scanner: nameToken b
19: Scanner: rightParToken
20:   8: { /* Computes the gcd of a and b. */
21: Scanner: leftCurlToken
22:   9:
23:   10: while (a != b) {
24: Scanner: whileToken
25: Scanner: leftParToken
26: Scanner: nameToken a
27: Scanner: notEqualToken
28: Scanner: nameToken b
29: Scanner: rightParToken
30: Scanner: leftCurlToken
31:   11: if (a < b) {
32: Scanner: ifToken
33: Scanner: leftParToken
34: Scanner: nameToken a
35: Scanner: lessToken
36: Scanner: nameToken b
37: Scanner: rightParToken
38: Scanner: leftCurlToken
39:   12: b = b-a;
40: Scanner: nameToken b
41: Scanner: assignToken
42: Scanner: nameToken b
43: Scanner: subtractToken
44: Scanner: nameToken a
45: Scanner: semicolonToken
46:   13: } else {
47: Scanner: rightCurlToken
48: Scanner: elseToken
49: Scanner: leftCurlToken
50:   14: a = a-b;
51: Scanner: nameToken a
52: Scanner: assignToken
53: Scanner: nameToken a
54: Scanner: subtractToken
55: Scanner: nameToken b
56: Scanner: semicolonToken
57:   15: }
58: Scanner: rightCurlToken
59:   16: }
60: Scanner: rightCurlToken
61:   17: return a;
62: Scanner: returnToken
63: Scanner: nameToken a
64: Scanner: semicolonToken
65:   18: }
66: Scanner: rightCurlToken
67:   19:
68:   20: int main ()
69: Scanner: intToken
70: Scanner: nameToken main
71: Scanner: leftParToken
72: Scanner: rightParToken
73:   21: {
74: Scanner: leftCurlToken
75:   22: int v1; int v2;

```

---

**Figur 6.10:** Loggfil som demonstrerer hvilke symboler skanneren finner i gcd.alboc (del 1)

```

76 Scanner: intToken
77 Scanner: nameToken v1
78 Scanner: semicolonToken
79 Scanner: intToken
80 Scanner: nameToken v2
81 Scanner: semicolonToken
82     23:
83     24:   LF = 10; ;
84 Scanner: nameToken LF
85 Scanner: assignToken
86 Scanner: numberToken 10
87 Scanner: semicolonToken
88 Scanner: semicolonToken
89     25: putchar('?'); putchar(' ');
90 Scanner: nameToken putchar
91 Scanner: leftParToken
92 Scanner: numberToken 63
93 Scanner: rightParToken
94 Scanner: semicolonToken
95 Scanner: nameToken putchar
96 Scanner: leftParToken
97 Scanner: numberToken 32
98 Scanner: rightParToken
99 Scanner: semicolonToken
100    26: v1 = getInt();   v2 = getInt();
101 Scanner: nameToken v1
102 Scanner: assignToken
103 Scanner: nameToken getInt
104 Scanner: leftParToken
105 Scanner: rightParToken
106 Scanner: semicolonToken
107 Scanner: nameToken v2
108 Scanner: assignToken
109 Scanner: nameToken getInt
110 Scanner: leftParToken
111 Scanner: rightParToken
112 Scanner: semicolonToken
113     27: putInt(gcd(v1,v2));  putchar(LF);
114 Scanner: nameToken putInt
115 Scanner: leftParToken
116 Scanner: nameToken gcd
117 Scanner: leftParToken
118 Scanner: nameToken v1
119 Scanner: commaToken
120 Scanner: nameToken v2
121 Scanner: rightParToken
122 Scanner: rightParToken
123 Scanner: semicolonToken
124 Scanner: nameToken putchar
125 Scanner: leftParToken
126 Scanner: nameToken LF
127 Scanner: rightParToken
128 Scanner: semicolonToken
129     28: exit(0);
130 Scanner: nameToken exit
131 Scanner: leftParToken
132 Scanner: numberToken 0
133 Scanner: rightParToken
134 Scanner: semicolonToken
135     29: }
136 Scanner: rightCurlToken
137 Scanner: eofToken

```

---

**Figur 6.11:** Loggfil som demonstrerer hvilke symboler skanneren finner i gcd.alboc (del 2)

---

```

1   1: # Program 'gcd'
2   2: # -----
3   3: # A program to compute the greatest common divisor.
4   4:
5   5: int LF; /* Line feed */
6 Parser:  <program>
7 Parser:  <type>
8 Parser:  </type>
9 Parser:  <var decl>
10  6:
11  7: int gcd (int a, int b)
12 Parser:  <var decl>
13 Parser:  <type>
14 Parser:  </type>
15 Parser:  <func decl>
16 Parser:  <type>
17 Parser:  </type>
18 Parser:  <param decl>
19 Parser:  </param decl>
20 Parser:  <type>
21 Parser:  </type>
22 Parser:  <param decl>
23  8: { /* Computes the gcd of a and b. */
24 Parser:  </param decl>
25  9:
26 10:   while (a != b) {
27 Parser:  <func body>
28 Parser:  <stmt list>
29 Parser:  <statement>
30 Parser:  <while-statm>
31 Parser:  <expression>
32 Parser:  <term>
33 Parser:  <factor>
34 Parser:  <primary>
35 Parser:  <operand>
36 Parser:  <variable>
37 Parser:  </variable>
38 Parser:  </operand>
39 Parser:  </primary>
40 Parser:  </factor>
41 Parser:  </term>
42 Parser:  <rel opr>
43 Parser:  </rel opr>
44 Parser:  <term>
45 Parser:  <factor>
46 Parser:  <primary>
47 Parser:  <operand>
48 Parser:  <variable>
49 Parser:  </variable>
50 Parser:  </operand>
51 Parser:  </primary>
52 Parser:  </factor>
53 Parser:  </term>
54 Parser:  </expression>
55 11:   if (a < b) {
56 Parser:  <stmt list>
57 Parser:  <statement>
58 Parser:  <if-statm>
59 Parser:  <expression>
60 Parser:  <term>
61 Parser:  <factor>
62 Parser:  <primary>
63 Parser:  <operand>
64 Parser:  <variable>
65 Parser:  </variable>
66 Parser:  </operand>
67 Parser:  </primary>
68 Parser:  </factor>
69 Parser:  </term>
70 Parser:  <rel opr>
71 Parser:  </rel opr>
72 Parser:  <term>
73 Parser:  <factor>
74 Parser:  <primary>
75 Parser:  <operand>

```

---

**Figur 6.12:** Logfil som viser parsering av gcd.alboc (del 1)

## KAPITTEL 6 PROSJEKTET

---

```
76 Parser:          <variable>
77 Parser:          </variable>
78 Parser:          </operand>
79 Parser:          </primary>
80 Parser:          </factor>
81 Parser:          </term>
82 Parser:          </expression>
83   12: b = b-a;
84 Parser:          <statm list>
85 Parser:          <statement>
86 Parser:          <assign-statm>
87 Parser:          <assignment>
88 Parser:          <lhs-variable>
89 Parser:          <variable>
90 Parser:          </variable>
91 Parser:          </lhs-variable>
92 Parser:          <expression>
93 Parser:          <term>
94 Parser:          <factor>
95 Parser:          <primary>
96 Parser:          <operand>
97 Parser:          <variable>
98 Parser:          </variable>
99 Parser:          </operand>
100 Parser:          </primary>
101 Parser:          </factor>
102 Parser:          <term opr>
103 Parser:          </term opr>
104 Parser:          <factor>
105 Parser:          <primary>
106 Parser:          <operand>
107 Parser:          <variable>
108   13: } else {
109 Parser:          </variable>
110 Parser:          </operand>
111 Parser:          </primary>
112 Parser:          </factor>
113 Parser:          </term>
114 Parser:          </expression>
115 Parser:          </assignment>
116 Parser:          </assign statm>
117 Parser:          </statement>
118 Parser:          </statm list>
119 Parser:          <else-part>
120   14: a = a-b;
121 Parser:          <statm list>
122 Parser:          <statement>
123 Parser:          <assign-statm>
124 Parser:          <assignment>
125 Parser:          <lhs-variable>
126 Parser:          <variable>
127 Parser:          </variable>
128 Parser:          </lhs-variable>
129 Parser:          <expression>
130 Parser:          <term>
131 Parser:          <factor>
132 Parser:          <primary>
133 Parser:          <operand>
134 Parser:          <variable>
135 Parser:          </variable>
136 Parser:          </operand>
137 Parser:          </primary>
138 Parser:          </factor>
139 Parser:          <term opr>
140 Parser:          </term opr>
141 Parser:          <factor>
142 Parser:          <primary>
143 Parser:          <operand>
144 Parser:          <variable>
145   15: }
146 Parser:          </variable>
147 Parser:          </operand>
148 Parser:          </primary>
149 Parser:          </factor>
150 Parser:          </term>
```

---

**Figur 6.13:** Loggfil som viser parsing av gcd.alboc (del 2)

---

```

151 Parser:           </expression>
152 Parser:           </assignment>
153   16:   }
154 Parser:           </assign statm>
155 Parser:           </statement>
156 Parser:           </statm list>
157   17:   return a;
158 Parser:           </else-part>
159 Parser:           </if-statm>
160 Parser:           </statement>
161 Parser:           </statm list>
162 Parser:           </while-statm>
163 Parser:           </statement>
164 Parser:           <return-statm>
165 Parser:           <expression>
166 Parser:           <term>
167 Parser:           <factor>
168 Parser:           <primary>
169 Parser:           <operand>
170 Parser:           <variable>
171 Parser:           18:   }
172 Parser:           </variable>
173 Parser:           </operand>
174 Parser:           </primary>
175 Parser:           </factor>
176 Parser:           </term>
177 Parser:           </expression>
178 Parser:           19:
179   20: int main ()
180 Parser:           </return-statm>
181 Parser:           </statement>
182 Parser:           </statm list>
183 Parser:           </func body>
184 Parser:           </func decl>
185 Parser:           <type>
186 Parser:           </type>
187 Parser:           </type>
188 Parser:           <func decl>
189   21: {
190   22:   int v1; int v2;
191 Parser:           <func body>
192 Parser:           <type>
193 Parser:           </type>
194 Parser:           <var decl>
195 Parser:           </var decl>
196 Parser:           <type>
197 Parser:           </type>
198 Parser:           <var decl>
199   23:
200   24:   LF = 10; ;
201 Parser:           </var decl>
202 Parser:           <statm list>
203 Parser:           <statement>
204 Parser:           <assign-statm>
205 Parser:           <assignment>
206 Parser:           <lhs-variable>
207 Parser:           <variable>
208 Parser:           </variable>
209 Parser:           </lhs-variable>
210 Parser:           <expression>
211 Parser:           <term>
212 Parser:           <factor>
213 Parser:           <primary>
214 Parser:           <operand>
215 Parser:           <number>
216 Parser:           </number>
217 Parser:           </operand>
218 Parser:           <primary>
219 Parser:           </factor>
220 Parser:           </term>
221 Parser:           </expression>
222 Parser:           </assignment>
223   25:   putchar(' ');
224 Parser:           </assign statm>
225 Parser:           </statement>

```

---

**Figur 6.14:** Loggfil som viser parsering av gcd.alboc (del 3)

```

226 Parser:      <statement>
227 Parser:          <empty statm>
228 Parser:          </empty statm>
229 Parser:      </statement>
230 Parser:      <statement>
231 Parser:          <call-statm>
232 Parser:              <function call>
233 Parser:                  <expr list>
234 Parser:                      <expression>
235 Parser:                          <term>
236 Parser:                              <factor>
237 Parser:                                  <primary>
238 Parser:                                      <operand>
239 Parser:                                          <number>
240 Parser:                                              </number>
241 Parser:          </operand>
242 Parser:              </primary>
243 Parser:          </factor>
244 Parser:      </term>
245 Parser:          </expression>
246 Parser:      </expr list>
247 Parser:          </function call>
248 Parser:      </call-statm>
249 Parser:      </statement>
250 Parser:      <statement>
251 Parser:          <call-statm>
252 Parser:              <function call>
253 Parser:                  <expr list>
254 Parser:                      <expression>
255 Parser:                          <term>
256 Parser:                              <factor>
257 Parser:                                  <primary>
258 Parser:                                      <operand>
259 Parser:                                          <number>
260 Parser:                                              </number>
261 Parser:          </operand>
262 Parser:              </primary>
263 Parser:          </factor>
264 Parser:      </term>
265 Parser:          </expression>
266 Parser:      </expr list>
267 26:    v1 = gint();    v2 = gint();
268 Parser:          </function call>
269 Parser:      </call-statm>
270 Parser:      </statement>
271 Parser:      <statement>
272 Parser:          <assign-statm>
273 Parser:              <assignment>
274 Parser:                  <lhs-variable>
275 Parser:                      <variable>
276 Parser:                          </variable>
277 Parser:                  </lhs-variable>
278 Parser:          <expression>
279 Parser:              <term>
280 Parser:                  <factor>
281 Parser:                      <primary>
282 Parser:                          <operand>
283 Parser:                              <function call>
284 Parser:                                  <expr list>
285 Parser:                                      </expr list>
286 Parser:          </function call>
287 Parser:          </operand>
288 Parser:          </primary>
289 Parser:          </factor>
290 Parser:      </term>
291 Parser:          </expression>
292 Parser:          </assignment>
293 Parser:      </assign statm>
294 Parser:      </statement>
295 Parser:      <statement>
296 Parser:          <assign-statm>
297 Parser:              <assignment>
298 Parser:                  <lhs-variable>
299 Parser:                      <variable>
300 Parser:                          </variable>

```

---

**Figur 6.15:** Loggfil som viser parsing av gcd.alboc (del 4)

---

```

301 Parser:          </lhs-variable>
302 Parser:          <expression>
303 Parser:          <term>
304 Parser:          <factor>
305 Parser:          <primary>
306 Parser:          <operand>
307 Parser:          <function call>
308 Parser:          <expr list>
309 Parser:          </expr list>
310     27: putint(gcd(v1,v2)); putchar(LF);
311 Parser:          </function call>
312 Parser:          </operand>
313 Parser:          </primary>
314 Parser:          </factor>
315 Parser:          </term>
316 Parser:          </expression>
317 Parser:          </assignment>
318 Parser:          </assign statm>
319 Parser:          </statement>
320 Parser:          <call-statm>
321 Parser:          <function call>
322 Parser:          <expr list>
323 Parser:          <expression>
324 Parser:          <term>
325 Parser:          <factor>
326 Parser:          <primary>
327 Parser:          <operand>
328 Parser:          <function call>
329 Parser:          <expr list>
330 Parser:          <expression>
331 Parser:          <term>
332 Parser:          <factor>
333 Parser:          <primary>
334 Parser:          <operand>
335 Parser:          <variable>
336 Parser:          </variable>
337 Parser:          </primary>
338 Parser:          </factor>
339 Parser:          </term>
340 Parser:          </expression>
341 Parser:          <expression>
342 Parser:          <term>
343 Parser:          <factor>
344 Parser:          <primary>
345 Parser:          <operand>
346 Parser:          <variable>
347 Parser:          </variable>
348 Parser:          </primary>
349 Parser:          <operand>
350 Parser:          </variable>
351 Parser:          </primary>
352 Parser:          </factor>
353 Parser:          </term>
354 Parser:          </expression>
355 Parser:          <expr list>
356 Parser:          </function call>
357 Parser:          </operand>
358 Parser:          </primary>
359 Parser:          </factor>
360 Parser:          </term>
361 Parser:          </expression>
362 Parser:          <expr list>
363 Parser:          </function call>
364 Parser:          </call-statm>
365 Parser:          </statement>
366 Parser:          <statement>
367 Parser:          <call-statm>
368 Parser:          <function call>
369 Parser:          <expr list>
370 Parser:          <expression>
371 Parser:          <term>
372 Parser:          <factor>
373 Parser:          <primary>
374 Parser:          <operand>
375 Parser:          <variable>

```

---

**Figur 6.16:** Loggfil som viser parsering av gcd.alboc (del 5)

## KAPITTEL 6 PROSJEKTET

---

```
376 Parser:           </variable>
377 Parser:           </operand>
378 Parser:           </primary>
379 Parser:           </factor>
380 Parser:           </term>
381 Parser:           </expression>
382 Parser:           </expr list>
383   28: exit(0);
384 Parser:           </function call>
385 Parser:           </call-stm>
386 Parser:           </statement>
387 Parser:           <statement>
388 Parser:           <call-stm>
389 Parser:           <function call>
390 Parser:           <expr list>
391 Parser:           <expression>
392 Parser:           <term>
393 Parser:           <factor>
394 Parser:           <primary>
395 Parser:           <operand>
396 Parser:           <number>
397 Parser:           </number>
398 Parser:           </operand>
399 Parser:           </primary>
400 Parser:           </factor>
401 Parser:           </term>
402 Parser:           </expression>
403 Parser:           </expr list>
404   29: }
405 Parser:           </function call>
406 Parser:           </call-stm>
407 Parser:           </statement>
408 Parser:           </stmt list>
409 Parser:           </func body>
410 Parser:           </func decl>
411 Parser:           </program>
```

---

**Figur 6.17:** Loggfil som viser parsing av gcd.alboc (del 6)

```
1 Tree: int LF;
2 Tree:
3 Tree: int gcd (int a, int b)
4 Tree: {
5 Tree:     while (a != b) {
6 Tree:         if (a < b) {
7 Tree:             b = b - a;
8 Tree:         } else {
9 Tree:             a = a - b;
10 Tree:         }
11 Tree:     }
12 Tree:     return a;
13 Tree: }
```

---

```
14 Tree:
15 Tree: int main ()
16 Tree: {
17 Tree:     int v1;
18 Tree:     int v2;
19 Tree:
20 Tree:     LF = 10;
21 Tree:     ;
22 Tree:     putchar(63);
23 Tree:     putchar(32);
24 Tree:     v1 = getInt();
25 Tree:     v2 = getInt();
26 Tree:     putInt(gcd(v1,v2));
27 Tree:     putchar(LF);
28 Tree:     exit(0);
29 Tree: }
```

---

**Figur 6.18:** Loggfil med trrepräsentasjonen av gcd.alboc

```

1 Binding: Line 10: a refers to declaration in line 7
2 Binding: Line 10: b refers to declaration in line 7
3 Binding: Line 11: a refers to declaration in line 7
4 Binding: Line 11: b refers to declaration in line 7
5 Binding: Line 12: b refers to declaration in line 7
6 Binding: Line 12: b refers to declaration in line 7
7 Binding: Line 12: a refers to declaration in line 7
8 Binding: Line 14: a refers to declaration in line 7
9 Binding: Line 14: a refers to declaration in line 7
10 Binding: Line 14: b refers to declaration in line 7
11 Binding: Line 17: a refers to declaration in line 7
12 Binding: Line 24: LF refers to declaration in line 5
13 Binding: Line 25: putchar refers to declaration in the library
14 Binding: Line 25: putchar refers to declaration in the library
15 Binding: Line 26: v1 refers to declaration in line 22
16 Binding: Line 26: getInt refers to declaration in the library
17 Binding: Line 26: v2 refers to declaration in line 22
18 Binding: Line 26: getint refers to declaration in the library
19 Binding: Line 27: putint refers to declaration in the library
20 Binding: Line 27: gcd refers to declaration in line 7
21 Binding: Line 27: v1 refers to declaration in line 22
22 Binding: Line 27: v2 refers to declaration in line 22
23 Binding: Line 27: putchar refers to declaration in the library
24 Binding: Line 27: LF refers to declaration in line 5
25 Binding: Line 28: exit refers to declaration in the library
26 Binding: main refers to declaration in line 20

```

---

**Figur 6.19:** Loggfil med navnebinding for gcd.alboc

```

1 Checking types: Line 10: x notEqualToken y,
2   where Type(x) is int and Type(y) is int
3 Checking types: Line 11: x lessToken y,
4   where Type(x) is int and Type(y) is int
5 Checking types: Line 12: x subtractToken y,
6   where Type(x) is int and Type(y) is int
7 Checking types: Line 12: v = e,
8   where Type(v) is int and Type(e) is int
9 Checking types: Line 14: x subtractToken y,
10  where Type(x) is int and Type(y) is int
11 Checking types: Line 14: v = e,
12  where Type(v) is int and Type(e) is int
13 Checking types: Line 11: if (t) ...,
14  where Type(t) is int
15 Checking types: Line 10: while (t) ...,
16  where Type(t) is int
17 Checking types: Line 24: v = e,
18  where Type(v) is int and Type(e) is int
19 Checking types: Line 25: Parameter #1 in call on putchar,
20  where Type(actual) is int and Type(formal) is int
21 Checking types: Line 25: Parameter #1 in call on putchar,
22  where Type(actual) is int and Type(formal) is int
23 Checking types: Line 26: v = e,
24  where Type(v) is int and Type(e) is int
25 Checking types: Line 26: v = e,
26  where Type(v) is int and Type(e) is int
27 Checking types: Line 27: Parameter #1 in call on gcd,
28  where Type(actual) is int and Type(formal) is int
29 Checking types: Line 27: Parameter #2 in call on gcd,
30  where Type(actual) is int and Type(formal) is int
31 Checking types: Line 27: Parameter #1 in call on putint,
32  where Type(actual) is int and Type(formal) is int
33 Checking types: Line 27: Parameter #1 in call on putchar,
34  where Type(actual) is int and Type(formal) is int
35 Checking types: Line 28: Parameter #1 in call on exit,
36  where Type(actual) is int and Type(formal) is int
37 Checking types: Line 17: return e; in int f(...),
38  where Type(e) is int

```

---

**Figur 6.20:** Loggfil med typesjekking for gcd.alboc

## KAPITTEL 6 PROSJEKTET

---

```

1      .data
2      .globl  LF
3  LF:   .fill   1,4,0          # int LF
4      .text
5      .globl  gcd
6  gcd:  enter  $0,$0          # Start function gcd
7  .L0001:    movl  8(%ebp),%eax
8      pushl  %eax
9      movl  12(%ebp),%eax      # b
10     popl  %ecx
11     cmpl  %eax,%ecx
12     movl  $0,%eax
13     setne %al
14     cmpl  $0,%eax
15     je    .L0002
16
17     movl  8(%ebp),%eax      # Start if-statement
18     pushl  %eax
19     movl  12(%ebp),%eax      # b
20     popl  %ecx
21     cmpl  %eax,%ecx
22     movl  $0,%eax
23     setl  %al
24     cmpl  $0,%eax
25     je    .L0004
26
27     leal  12(%ebp),%eax      # b
28     pushl  %eax
29     movl  12(%ebp),%eax      # b
30     pushl  %eax
31     movl  8(%ebp),%eax      # a
32     movl  %eax,%ecx
33     popl  %eax
34     subl  %ecx,%eax        # Compute -
35     popl  %edx
36     movl  %eax,(%edx)
37     jmp   .L0003
38
39     leal  8(%ebp),%eax      # else-part
40     pushl  %eax
41     movl  8(%ebp),%eax      # a
42     pushl  %eax
43     movl  12(%ebp),%eax      # b
44     movl  %eax,%ecx
45     popl  %eax
46     subl  %ecx,%eax        # Compute -
47     popl  %edx
48     movl  %eax,(%edx)
49
50     .L0003:    jmp   .L0001
51
52     .L0002:    movl  8(%ebp),%eax
53     jmp   .exit$gcd
54
55     .exit$gcd:
56     leave
57     ret
58
59     .globl  main
60     main:  enter  $8,$0          # Start function main
61     leal  LF,%eax
62     pushl  %eax
63     movl  $10,%eax
64     popl  %edx
65     movl  %eax,(%edx)
66     movl  $63,%eax
67     pushl  %eax
68     call   putchar
69     addl  $4,%esp
70     movl  $32,%eax
71     pushl  %eax
72     call   putchar
73     addl  $4,%esp
74     leal  -4(%ebp),%eax
75     pushl  %eax
76     call   getint
77     popl  %edx

```

---

**Figur 6.21:** Kodefil produsert fra gcd.alboc (del 1)

---

```
76    movl %eax,(%edx)      # =
77    leal -8(%ebp),%eax    # v2
78    pushl %eax
79    call getint           # Call getint
80    popl %edx
81    movl %eax,(%edx)      # =
82    movl -8(%ebp),%eax    # v2
83    pushl %eax            # Push parameter #2
84    movl -4(%ebp),%eax    # v1
85    pushl %eax            # Push parameter #1
86    call gcd               # Call gcd
87    addl $8,%esp           # Remove parameters
88    pushl %eax            # Push parameter #1
89    call putint             # Call putint
90    addl $4,%esp           # Remove parameters
91    movl LF,%eax           # LF
92    pushl %eax            # Push parameter #1
93    call putchar             # Call putchar
94    addl $4,%esp           # Remove parameters
95    movl $0,%eax           # 0
96    pushl %eax            # Push parameter #1
97    call exit                # Call exit
98    addl $4,%esp           # Remove parameters
99 .exit$main:
100   leave
101   ret                     # End function main
```

---

**Figur 6.22:** Kodefil produsert fra gcd.alboc (del 2)



# Kapittel 7

# Koding

## 7.1 Suns anbefalte Java-stil

Datafirmaet Sun, som utviklet Java, har også tanker om hvordan Java-koden bør se ut. Dette er uttrykt i et lite skriv på 24 sider som kan hentes fra <http://java.sun.com/docs/codeconv/CodeConventions.pdf>. Her er hovedpunktene.

### 7.1.1 Klasser

Hver klasse bør ligge i sin egen kildefil; unntatt er private klasser som «tilhører» en vanlig klasse.

Klasse-filer bør inneholde følgende (i denne rekkefølgen):

- 1) En kommentar med de aller viktigste opplysningene om filen:

```
/*
 * Klassens navn
 *
 * Versjonsinformasjon
 *
 * Copyrightangivelse
 */
```

- 2) Alle import-spesifikasjonene.
- 3) JavaDoc-kommentar for klassen. (JavaDoc er beskrevet i avsnitt 8.1 på side 71.)
- 4) Selve klassen.

### 7.1.2 Variabler

Variabler bør deklarereres én og én på hver linje:

```
int level;
int size;
```

De bør komme først i {}-blokken (dvs før alle setningene), men lokale for-indekser er helt OK:

```
for (int i = 1; i <= 10; ++i) {
    ...
}
```

```
do {  
    setninger;  
} while (uttrykk);  
  
for (init; betingelse; oppdatering) {  
    setninger;  
}  
  
if (uttrykk) {  
    setninger;  
}  
  
if (uttrykk) {  
    setninger;  
} else {  
    setninger;  
}  
  
if (uttrykk) {  
    setninger;  
} else if (uttrykk) {  
    setninger;  
} else if (uttrykk) {  
    setninger;  
}  
  
return uttrykk;  
  
switch (uttrykk) {  
case xxx    setninger;  
    break;  
  
case xxx    setninger;  
    break;  
  
default     setninger;  
    break;  
}  
  
try {  
    setninger;  
} catch (ExceptionClass e) {  
    setninger;  
}  
  
while (uttrykk) {  
    setninger;  
}
```

**Figur 7.1:** Suns forslag til hvordan setninger bør skrives

Om man kan initialisere variablene samtidig med deklarasjonen, er det en fordel.

### 7.1.3 Setninger

Enkle setninger bør stå én og én på hver linje:

```
i = 1;  
j = 2;
```

De ulike sammensatte setningene skal se ut slik figur 7.1 viser. De skal alltid ha {} rundt innmaten, og innmaten skal indenteres 4 posisjoner.

### 7.1.4 Navn

Navn bør velges slik det er angitt i tabell 7.1 på neste side.

Type navn	Kapitalisering	Hva slags ord	Eksempel
Klasser	XxxxxXxxx	Substantiv som beskriver objektene	IfStatement
Metoder	xxxxxXxxx	Verb som angir hva metoden gjør	readToken
Variabler	xxxxxxXxxx	Korte substantiver; «bruk-og-kast-variabler» kan være på én bokstav	curToken, i
Konstanter	XXXX_XX	Substantiv	MAX_MEMORY

**Tabell 7.1:** Suns forslag til navnevalg i Java-programmer

### 7.1.5 Utseende

#### 7.1.5.1 Linjelengde og linjedeling

Linjene bør ikke være mer enn 80 tegn lange, og kommentarer ikke lengre enn 70 tegn.

En linje som er for lang, bør deles

- etter et komma eller
- før en operator (som + eller &&).

Linjedelen etter delingspunktet bør indenteres likt med starten av uttrykket som ble delt.

#### 7.1.5.2 Blanke linjer

Sett inn doble blanke linjer

- mellom klasser.

Sett inn enkle blanke linjer

- mellom metoder,
- mellom variabeldeklarasjonene og første setning i metoder eller
- mellom ulike deler av en metode.

#### 7.1.5.3 Mellomrom

Sett inn mellomrom

- etter kommaer i parameterlister,
- rundt binære operatorer:  

```
if (x < a + 1) {
```

(men ikke etter unære operatorer: -a)
- ved typekonvertering:  

```
(int) x
```



# Kapittel 8

# Dokumentasjon

## 8.1 JavaDoc

Sun har også laget et opplegg for dokumentasjon av programmer. Hovedtankene er

- 1) Brukeren skriver kommentarer i hver Java-pakke, -klasse og -metode i henhold til visse regler.
- 2) Et eget program javadoc leser kodelinene og bygger opp et helt nett av HTML-filer med dokumentasjonen.

Et typisk eksempel på JavaDoc-dokumentasjon er den som beskriver Javas enorme bibliotek: <http://java.sun.com/javase/7/docs/api/>.

### 8.1.1 Hvordan skrive JavaDoc-kommentarer

Det er ikke vanskelig å skrive JavaDoc-kommentarer. Her er en kort innføring til hvordan det skal gjøres; den fulle beskrivelsen finnes på nettsiden <http://java.sun.com/j2se/javado/writingdoccomments/>.

En JavaDoc-kommentar for en klasse ser slik ut:

```
/**  
 * Én setning som kort beskriver klassen  
 * Mer forklaring  
 * :  
 * @author navn  
 * @author navn  
 * @version dato  
 */
```

Legg spesielt merke til den doble stjernen på første linje – det er den som angir at dette er en JavaDoc-kommentar og ikke bare en vanlig kommentar.

JavaDoc-kommentarer for metoder følger nesten samme oppsettet:

```
/**  
 * Én setning som kort beskriver metoden  
 * Ytterligere kommentarer  
 * :  
 * @param navn1 Kort beskrivelse av parameteren  
 * @param navn2 Kort beskrivelse av parameteren
```

```
* @return Kort beskrivelse av returverdien
* @see      navn3
*/
```

Her er det viktig at den første setningen kort og presist forteller hva metoden gjør. Denne setningen vil bli brukt i metodeoversikten.

Ellers er verdt å merke seg at kommentaren skrives i HTML-kode, så man kan bruke konstruksjoner som `<i>...</i>` eller `<table>...</table>` om man ønsker det.

### 8.1.2 Eksempel

I figur 8.1 kan vi se en Java-metode med dokumentasjon.

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

**Figur 8.1:** Java-kode med JavaDoc-kommentarer

## 8.2 «Lesbar programmering»

Lesbar programmering («literate programming») er oppfunnet av Donald Knuth, forfatteren av *The art of computer programming* og opphavsmannen til *T<sub>E</sub>X*. Hovedtanken er at programmer først og fremst skal skrives slik at mennesker kan lese dem; datamaskiner klarer å «forstå» alt så lenge programmet er korrekt. Dette innebærer følgende:

- Programkoden og dokumentasjonen skrives som en enhet.
- Programmet deles opp i passende små navngitte enheter som legges inn i dokumentasjonen. Slike enheter kan referere til andre enheter.
- Programmet skrives i den rekkefølgen som er enklest for leseren å forstå.
- Dokumentasjonen skrives i et dokumentasjonsspråk (som *L<sub>A</sub>T<sub>E</sub>X*) og kan benytte alle tilgjengelige typografiske hjelpe midler som figurer, matematiske formler, fotnoter, kapittelinndeling, fontskifte og annet.

- Det kan automatisk lages oversikter og klasser, funksjoner og variabler: hvor de deklarereres og hvor de brukes.

Ut ifra kildekoden («web-koden») kan man så lage

- 1) et dokument som kan skrives ut og
- 2) en kompilerbar kildekode.

### 8.2.1 Et eksempel

Som eksempel skal vi bruke en implementasjon av boblesortering. Fremgangsmåten er som følger:

- 1) Skriv kildefilen `bubble.w0` (vist i figur 8.2 og 8.3). Dette gjøres med en vanlig tekstbehandler som for eksempel Emacs.

- 2) Bruk programmet `weave01` til å lage det ferdige dokumentet som er vist i figur 8.4–8.7:

```
$ weave0 -l c -e -o bubble.tex bubble.w0  
$ ltx2pdf bubble.tex
```

- 3) Bruk `tangle0` til å lage et kjørbart program:

```
$ tangle0 -o bubble.c bubble.w0  
$ gcc -c bubble.c
```

---

<sup>1</sup> Dette eksemplet bruker Dags versjon av lesbar programmering kalt `web0`; for mer informasjon, se <http://dag.at.ifi.uio.no/public/doc/web0.pdf>.

**bubble.w0 del 1**

```
\documentclass[12pt,a4paper]{webzero}
\usepackage[latin1]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{amssymb,mathpazo,textcomp}

\title{Bubble sort}
\author{Dag Langmyhr\\ Department of Informatics\\
University of Oslo\\ [5pt] \texttt{dag@ifi.uio.no} }

\begin{document}
\maketitle

\noindent This short article describes \emph{bubble sort}, which quite probably is the easiest sorting method to understand and implement. Although far from being the most efficient one, it is useful as an example when teaching sorting algorithms.

Let us write a function \texttt{bubble} in C which sorts an array \texttt{a} with \texttt{n} elements. In other words, the array \texttt{a} should satisfy the following condition when \texttt{bubble} exits:
\[
\forall i, j \in \mathbb{N}: 0 \leq i < j < n
\Rightarrow a[i] \leq a[j]
\]

<<bubble sort>>=
void bubble(int a[], int n)
{
    <<local variables>>
    <<use bubble sort>>
}
@
Bubble sorting is done by making several passes through the array, each time letting the larger elements "bubble" up. This is repeated until the array is completely sorted.

<<use bubble sort>>=
do {
    <<perform bubbling>>
} while (<<not sorted>>);
@
```

**Figur 8.2:** «Lesbar programmering» – kildefilen `bubble.w0 del 1`

**bubble.w0 del 2**

Each pass through the array consists of looking at every pair of adjacent elements;<sup>We could, on the average, double the execution speed of \texttt{bubble} by reducing the range of the \texttt{for}-loop by 1 each time.</sup> Since a simple implementation is the main issue, however, this improvement was omitted.) if the two are in the wrong sorting order, they are swapped:

```
<<perform bubbling>>
<<initialize>>
for (i=0; i<n-1; ++i)
    if (a[i]>a[i+1]) { <<swap a[i] and a[i+1]>> }
@
```

The \texttt{for}-loop needs an index variable \texttt{i}:

```
<<local var...>>
int i;
@
```

Swapping two array elements is done in the standard way using an auxiliary variable \texttt{temp}. We also increment a swap counter named \texttt{n\\_swaps}.

```
<<swap ...>>
temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
++n_swaps;
@
```

The variables \texttt{temp} and \texttt{n\\_swaps} must also be declared:

```
<<local var...>>
int temp, n_swaps;
@
```

The variable \texttt{n\\_swaps} counts the number of swaps performed during one ‘‘bubbling’’ pass. It must be initialized prior to each pass.

```
<<initialize>>
n_swaps = 0;
@
```

If no swaps were made during the ‘‘bubbling’’ pass, the array is sorted.

```
<<not sorted>>
n_swaps > 0
@
```

```
\wzvarindex \wzmetaindex
\end{document}
```

**Figur 8.3:** «Lesbar programmering» – kildefilen bubble.w0 del 2

## Bubble sort

Dag Langmyhr  
 Department of Informatics  
 University of Oslo  
 dag@ifi.uio.no

July 4, 2014

This short article describes *bubble sort*, which quite probably is the easiest sorting method to understand and implement. Although far from being the most efficient one, it is useful as an example when teaching sorting algorithms.

Let us write a function `bubble` in C which sorts an array `a` with `n` elements. In other words, the array `a` should satisfy the following condition when `bubble` exits:

$$\forall i, j \in \mathbb{N} : 0 \leq i < j < n \Rightarrow a[i] \leq a[j]$$

```
#1  ⟨bubble sort⟩ ≡
1   void bubble(int a[], int n)
2   {
3     ⟨local variables #4(p.1)⟩
4
5     ⟨use bubble sort #2(p.1)⟩
6   }
⟨This code is not used.⟩
```

Bubble sorting is done by making several passes through the array, each time letting the larger elements “bubble” up. This is repeated until the array is completely sorted.

```
#2  ⟨use bubble sort⟩ ≡
7   do {
8     ⟨perform bubbling #3(p.1)⟩
9   } while (⟨not sorted #7(p.2)⟩);
⟨This code is used in #1 (p.1).⟩
```

Each pass through the array consists of looking at every pair of adjacent elements;<sup>1</sup> if the two are in the wrong sorting order, they are swapped:

```
#3  ⟨perform bubbling⟩ ≡
10  ⟨initialize #6(p.2)⟩
11  for (i=0; i<n-1; ++i)
12    if (a[i]>a[i+1]) { ⟨swap a[i] and a[i+1] #5(p.2)⟩ }
⟨This code is used in #2 (p.1).⟩
```

The `for`-loop needs an index variable `i`:

```
#4  ⟨local variables⟩ ≡
13  int i;
⟨This code is extended in #4 (p.2). It is used in #1 (p.1).⟩
```

<sup>1</sup>We could, on the average, double the execution speed of `bubble` by reducing the range of the `for`-loop by 1 each time. Since a simple implementation is the main issue, however, this improvement was omitted.

**Figur 8.4:** «Lesbar programmering» – utskrift side 1

Swapping two array elements is done in the standard way using an auxiliary variable `temp`. We also increment a swap counter named `n_swaps`.

```
#5  ⟨swap a[i] and a[i+1]⟩ ≡  
14   temp = a[i];  a[i] = a[i+1];  a[i+1] = temp;  
15   ++n_swaps;  
(This code is used in #3 (p.1).)
```

The variables `temp` and `n_swaps` must also be declared:

```
#4a  ⟨local variables #4(p.1)⟩ +≡  
16   int temp, n_swaps;
```

The variable `n_swaps` counts the number of swaps performed during one “bubbling” pass. It must be initialized prior to each pass.

```
#6  ⟨initialize⟩ ≡  
17   n_swaps = 0;  
(This code is used in #3 (p.1).)  
  
If no swaps were made during the “bubbling” pass, the array is sorted.  
  
#7  ⟨not sorted⟩ ≡  
18   n_swaps > 0  
(This code is used in #2 (p.1).)
```

**Figur 8.5:** «Lesbar programmering» – utskrift side 2

**Variables**

**A**

a .....1, 12, 14

**I**

i ..... 11, 12, 13, 14

**N**

n .....1, 11

n\_swaps ..... 15, 16, 17, 18

**T**

temp ..... 14, 16

VARIABLES

page 3

**Figur 8.6:** «Lesbar programmering» – utskrift side 3

### Meta symbols

<i>(bubble sort #1)</i> .....	page	1 *
<i>(initialize #6)</i> .....	page	2
<i>(local variables #4)</i> .....	page	1
<i>(not sorted #7)</i> .....	page	2
<i>(perform bubbling #3)</i> .....	page	1
<i>(swap <math>a[i]</math> and <math>a[i+1]</math> #5)</i> .....	page	2
<i>(use bubble sort #2)</i> .....	page	1

(Symbols marked with \* are not used.)

META SYMBOLS

page 4

**Figur 8.7:** «Lesbar programmering» – utskrift side 4



# Register

.L0001, 36  
.exit\$f, 35  
  
ALBOC, 21  
alboc, 43  
ant, 47  
Array, 23  
Assembler, 16  
Assemblerspråk, 16  
  
chargenerator, 43  
check, 45  
code, 46  
curC, 43  
curLine, 44  
curName, 44  
curNum, 44  
curToken, 44, 45  
  
emacs, 48  
eofToken, 45  
error, 46  
Exception, 46  
  
Feil, 46  
finish, 43  
  
gas, 47  
gcc, 47  
genCode, 45  
  
Høynivå programmeringsspråk, 11  
  
init, 43  
Initialverdi, 23  
Interpreter, 14  
  
java, 47  
Java-pakker, 43  
javac, 47, 48  
JavaDoc, 71  
  
Kommandospråk, 14  
Kompilator, 11  
  
Linux, 47  
log, 46  
  
Mac OS X, 47  
main, 23, 43  
  
Maskinspråk, 11  
  
nameToken, 44  
nextC, 43  
nextLine, 44  
nextName, 44  
nextNum, 44  
nextToken, 44, 45  
numberToken, 44  
  
package, 43  
parse, 45, 50  
Parsering, 45  
Parseringstre, 45  
Preprocessor, 13  
Presedens, 26  
printTree, 45  
Programmeringsstil, 67  
  
readNext, 43, 45  
  
scanner, 44  
Skanner, 17  
StreamTokenizer, 18  
Symboler, 17  
Syntaks, 17  
Syntakstre, 17, 45  
syntax, 45  
SyntaxUnit, 45  
  
Token, 44  
Tokenizer, 18  
Tokens, 17  
Type, 23  
types, 45  
  
Unicode, 49  
  
Virtuelle, 45  
  
Windows, 47

