

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

C_b og kompilatoren hans

Kompendium for INF2100

Stein Krogdahl, Dag Langmyhr
Høsten 2013



Innhold

Innhold	3
Figurer	7
Tabeller	9
Forord	11
I Innledning	13
1.1 Hva er emnet INF2100?	13
1.2 Hvorfor oppgaven er å lage en kompilator	14
1.3 Litt om kompilatorer og liknende verktøy	14
1.3.1 Preprosessorer	15
1.3.2 Interpreting	16
1.3.3 Kompilering og kjøring av Java-programmer	16
1.4 Språkene i oppgaven	17
1.4.1 Programmeringsspråket C _b	17
1.4.2 Datamaskinen x86 og dens maskinspråk	17
1.4.3 Assembleren	17
1.4.4 Oversikt over de ulike språkene i oppgaven	18
1.5 Oppgaven og dens tre deler	18
1.5.1 Del 0	19
1.5.2 Del 1	19
1.5.3 Del 2	19
1.6 Programmering av lister, trær etc	19
1.7 Krav til samarbeid og gruppetilhørighet	20
1.8 Kontroll av innlevert arbeid	20
1.9 Delta på øvingsgruppene	21
2 Programmeringsspråket C_b	23
2.1 Kjøring	23
2.1.1 Kompilering med C-kompilatoren	23
2.2 C _b -program	25
2.2.1 Variabler	25
2.2.2 Funksjoner	25
2.2.3 Setninger	26
2.2.4 Uttrykk	28
2.2.5 Andre ting	31
2.3 Forskjeller til C	31
3 Datamaskinen x86	33
3.1 Minnet	33
3.2 Prosessoren x86	33
3.3 Assemblerkode	34
3.3.1 Assemblerdirektiver	36

INNHOLD

4 Kodegenerering	37
4.1 Konvensjoner	37
4.1.1 Heltallsregistre	37
4.1.2 Flyttallsregistre	37
4.1.3 Navn	37
4.2 Oversettelse av uttrykk	38
4.2.1 Operander i uttrykk	38
4.2.2 Operatorer i uttrykk	38
4.3 Oversettelse av setninger	39
4.3.1 Oversettelse av tomme setninger	39
4.3.2 Oversettelse av tilordninger	39
4.3.3 Oversettelse av kallsetninger	39
4.3.4 Oversettelse av for-setninger	40
4.3.5 Oversettelse av if-setninger	40
4.3.6 Oversettelse av return-setninger	40
4.3.7 Oversettelse av while-setninger	41
4.4 Oversettelse av funksjoner og funksjonskall	41
4.4.1 Oversettelse av funksjonsdeklarasjoner	42
4.4.2 Oversettelse av funksjonskall	43
4.5 Deklarasjon av variabler	44
4.5.1 Deklarasjon av globale variabler	44
4.5.2 Deklarasjon av lokale variabler	45
4.5.3 Deklarasjon av parametre	45
5 Implementasjonen	47
5.1 Modulen Cflat	47
5.2 Modulen CharGenerator	47
5.3 Modulen Scanner	47
5.4 Modulen Syntax	49
5.5 Modulen Types	49
5.6 Modulen Code	50
5.7 Modulen Error	51
5.8 Modulen Log	51
6 Prosjektet	53
6.1 På egen datamaskin	53
6.2 Tegnsett	54
6.3 Del 0	55
6.4 Del 1	55
6.5 Del 2	56
6.5.1 Sjekking	56
7 Koding	69
7.1 Suns anbefalte Java-stil	69
7.1.1 Klasser	69
7.1.2 Variabler	69
7.1.3 Setninger	70
7.1.4 Navn	70
7.1.5 Utseende	70

INNHOLD

8 Dokumentasjon	73
8.1 JavaDoc	73
8.1.1 Hvordan skrive JavaDoc-kommentarer	73
8.1.2 Eksempel	74
8.2 «Lesbar programmering»	74
8.2.1 Et eksempel	75
Register	83

Figurer

1.1 Sammenhengen mellom C _b , kompilator, assembler og en x86-maskin	18
2.1 Eksempel på et C _b -program	24
2.2 Jernbanediagram for <program>	25
2.3 Jernbanediagram for <var decl>	25
2.4 Jernbanediagram for <type>	25
2.5 Jernbanediagram for <func decl>	25
2.6 Jernbanediagram for <param decl>	26
2.7 Jernbanediagram for <func body>	26
2.8 Jernbanediagram for <statm list>	26
2.9 Jernbanediagram for <statement>	26
2.10 Jernbanediagram for <empty statm>	26
2.11 Jernbanediagram for <assign-statm>	27
2.12 Jernbanediagram for <assignment>	27
2.13 Jernbanediagram for <call-statm>	27
2.14 Jernbanediagram for <for-statm>	27
2.15 Jernbanediagram for <for-control>	27
2.16 Jernbanediagram for <if-statm>	27
2.17 Jernbanediagram for <else-part>	27
2.18 Jernbanediagram for <return-statm>	28
2.19 Jernbanediagram for <while-statm>	28
2.20 Jernbanediagram for <expression>	28
2.21 Jernbanediagram for <rel opr>	28
2.22 Jernbanediagram for <term>	28
2.23 Jernbanediagram for <term opr>	29
2.24 Jernbanediagram for <factor>	29
2.25 Jernbanediagram for <factor opr>	29
2.26 Jernbanediagram for <operand>	29
2.27 Jernbanediagram for <variable>	29
2.28 Jernbanediagram for <function call>	29
2.29 Jernbanediagram for <expr list>	29
2.30 Lovlige og ulovlige typekombinasjoner for operander og tilordninger	30
2.31 Jernbanediagram for <name>	30
2.32 Jernbanediagram for <number>	31
3.1 Hovedkortet i en datamaskin	34
3.2 Instruksjonslinje i assemblerkode	34
5.1 Modulene i kompilatoren	48
5.2 Oppsett for de enkelte Java-pakkene	48
5.3 Et parseringstre	50
6.1 De ulike delene i prosjektet	54
6.2 Et minimalt C _b -program mini.cflat	58

FIGURER

6.3	Skanning av <code>mini.cls</code>	58
6.4	Parsering av <code>mini.cflat</code>	58
6.5	Utskrift av treet til <code>mini.cflat</code>	58
6.6	Navnebindinger i <code>mini.cflat</code>	59
6.7	Generert kodelinje for <code>mini.cflat</code>	59
6.8	Et litt større C _b -program <code>gcd.less</code>	59
6.9	Skanning av <code>gcd.cflat</code> (del 1)	60
6.10	Skanning av <code>gcd.cflat</code> (del 2)	61
6.11	Parsering av <code>gcd.cflat</code> (del 1)	62
6.12	Parsering av <code>gcd.cflat</code> (del 2)	63
6.13	Parsering av <code>gcd.cflat</code> (del 3)	64
6.14	Parsering av <code>gcd.cflat</code> (del 4)	65
6.15	Parsering av <code>gcd.cflat</code> (del 5)	66
6.16	Utskrift av treet til <code>gcd.cflat</code>	66
6.17	Generert kodelinje for <code>gcd.cflat</code> (del 1)	67
6.18	Generert kodelinje for <code>gcd.cflat</code> (del 2)	68
6.19	Navnebindinger i <code>gcd.cflat</code>	68
7.1	Suns forslag til hvordan setninger bør skrives	71
8.1	Java-kode med JavaDoc-kommentarer	74
8.2	«Lesbar programmering» — kildefilen <code>bubble.w0</code> del 1	76
8.3	«Lesbar programmering» — kildefilen <code>bubble.w0</code> del 2	77
8.4	«Lesbar programmering» — utskrift side 1	78
8.5	«Lesbar programmering» — utskrift side 2	79
8.6	«Lesbar programmering» — utskrift side 3	80
8.7	«Lesbar programmering» — utskrift side 4	81

Tabeller

2.1	CØs biblioteksfunksjoner	30
2.2	Tegnsettet ISO 8859-1	32
3.1	x86-instruksjoner brukt i prosjektet	35
3.2	Assemblerdirektiver	36
4.1	Kode generert av operand i uttrykk	39
4.2	Kode generert av heltallsoperatorene + og /	40
4.3	Kode generert av flyttallsoperatoren +	40
4.4	Kode generert av operatoren ==	41
4.5	Instruksjoner til de ulike sammenligningsoperatorene	41
4.6	Kode generert av tom setning	41
4.7	Kode generert av tilordning	42
4.8	Kode generert av funksjonskall som setning	42
4.9	Kode generert av if-setning	43
4.10	Kode generert av return-setning	43
4.11	Kode generert av while-setning	43
4.12	Kode generert av funksjonsdeklarasjon	44
4.13	Kode generert av funksjonskall	44
4.14	Kode generert av globale variabeldeklarasjoner	45
6.1	Opsjoner for logging	54
7.1	Suns forslag til navnevalg i Java-programmer	70

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 renoveret 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 kan 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: *C_b* (uttales «c flat»).

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 *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, Ryhor Sivuda, Christian Tryti, Jørgen Vigdal* og *Olga Voronkova* som har påpekt skrivefeil i tidligere utgaver. Om flere studenter finner feil, vil de også få navnet sitt på trykk.

Blindern, 27. juni 2013
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 I

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 **høynivå 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, og ikke minst senere gjøre endringer i, slike programmer, 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 oppgaven er å 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 omtrent 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 oppsplittning 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 C#. Vi skal i det følgende se litt nærmere på dette og andre elementer som inngår i oppgaven.

1.3 Litt 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.

Grunnen til at man i det hele tatt har kompilatorer, 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 1000–3000 millioner 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 komplilatorer for C++ skrevet som selvstendige komplilatorer 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 preprosessen 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 programmeringsspråket 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; et 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 kodebitar oversettes til et mellomnivåspråk som er bedre egnet for interpreting – og som da interpereteres 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 kompileres 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 interperetering 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 interperitere 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 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 C_b

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 C_b. 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 C_b blir beskrevet i detalj i kapittel 2 på side 23.

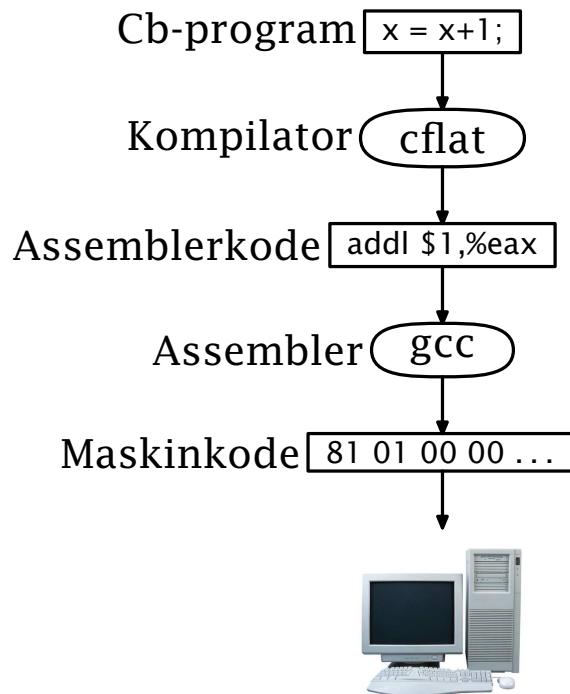
1.4.2 Datamaskinen x86 og dens maskinspråk

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

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 (= 81₁₆),



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

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 **assemblerere**. 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) Java, som C_b-kompilatoren skal skrives i.
- 2) C_b, som kompilatoren skal oversette fra.
- 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 Cb's **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 Cb-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 Cb-program skal ha (altså, at den følger Cb's **syntaks**).

Om alt er i orden, skal del 1 bygge opp et **syntakstre**, en **trestruktur** av objekter som direkte representerer det aktuelle Cb-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 Cb-programmet. Koden skal legges på en fil og den skal være i såkalt x86 assemblerformat.

I kapittel 4 på side 37 er det angitt hvilke sekvenser av x86-instruksjoner hver enkelt Cb-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.*` 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.

I.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.

I.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 komplataoren 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 testet ut programmet, kan dere komplataoren 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 stort sett meget relevante for skriving av C_b-kompilatoren. Om man tar en liten titt på oppgavene før gruppetimene, 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

Programmeringsspråket C_b

Programmeringsspråket C_b er en miniversjon av C; navnet uttales «si flætt» og er et ordspill på C#. Syntaksen er gitt av jernbanediagrammene i figur 2.2 til 2.32 på side 25–31 og bør være lett forståelig for alle som har programmert litt i C. Et eksempel på et C_b-program er vist i figur 2.1 på neste side.¹

2.1 Kjøring

Inntil dere selv har laget en C_b-kompilator, kan dere benytte referansekompiletoren:

```
$ cflat easter.cflat
This is the Cb compiler (version 2013-06-24 on Linux)
Parsing... checking... generating code... OK
Running gcc -m32 -o easter easter.s -L. -L/local/share/inf2100 -lcflat
$ ./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 C_b 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 C_b-biblioteket -lcflat:²

¹ 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/>.

² Man bør unngå #-linjer når man skal benytte C-kompilatoren.

```

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 C_b-program

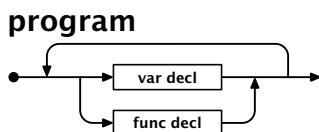
```

$ gcc -m32 -o e -x c easter.cflat -L. -L/local/share/inf2100 -lcflat
$ ./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 C_b-program

Som vist i figur 2.2 er et C_b-program rett og slett en samling funksjoner (kjent som «metoder» i Java). Før, mellom og etter disse funksjonene kan man deklarere globale variabler.



Figur 2.2: Jernbanediagram for <program>

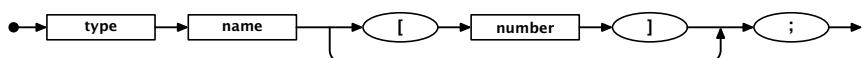
For at vi skal ha et kjørbart program, må det eksistere en int-funksjon med navnet **main**, og programutførelsen starter alltid med å kalle denne funksjonen. Funksjonen **main** kan ikke ha noen parametere.

2.2.1 Variabler

Brukeren kan deklarere enten globale variabler eller variabler som er lokale i en funksjon. Det finnes to datatyper: **int** og **double**. Det er også mulig å deklarere en **vektor** («array») av disse to datatypene; vektorer indekseres fra 0 (som i C og Java) og indeksen må alltid være en **int**. Antallet elementer i en vektordeklarasjon må være en heltallskonstant.

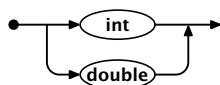
Det er ikke mulig å angi noen **initialverdi** for variabler – de inneholder en ukjent verdi før de tilordnes en verdi av programmet.

var decl



Figur 2.3: Jernbanediagram for <var decl>

type

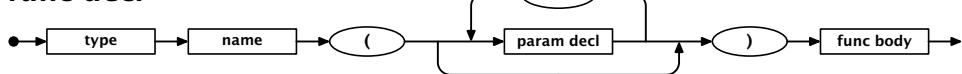


Figur 2.4: Jernbanediagram for <type>

2.2.2 Funksjoner

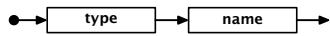
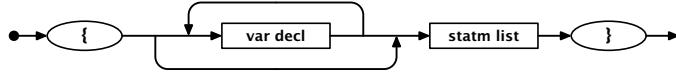
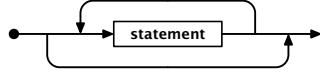
Brukeren kan deklarere funksjoner med vilkårlig mange parametere; parametrene kan ikke være vektorer. Parameteroverføringen skjer ved kopiering (som i C og Java).

func decl

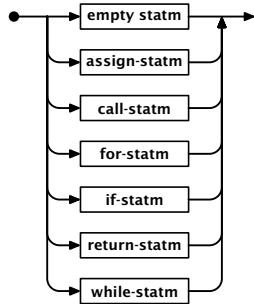


Figur 2.5: Jernbanediagram for <func decl>

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

param decl**Figur 2.6:** Jernbanediagram for <param decl>**func body****Figur 2.7:** Jernbanediagram for <func body>**statm list****Figur 2.8:** Jernbanediagram for <statm list>**2.2.3 Setninger**

C_b kjenner til syv ulike setninger.

statement**Figur 2.9:** Jernbanediagram for <statement>**2.2.3.1 Tomme setninger**

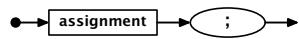
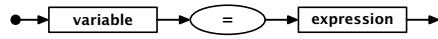
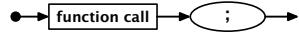
En tom setning (der det ikke står noe foran semikolonet) er lov i C_b. Naturlig nok gjør den ingenting.

empty statm**Figur 2.10:** Jernbanediagram for <empty statm>**2.2.3.2 Tilordninger**

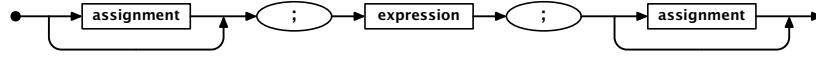
Det er tillatt å tilordne verdier til vanlige enkle variabler eller vektorelementer. Det er lov å tilordne et heltall til et flyttall eller omvendt; i så fall blir verdien konvertert.

2.2.3.3 Funksjonskall

Et funksjonskall kan brukes som en egen setning.

assign-stmt**Figur 2.11:** Jernbanediagram for <assign-stmt>**assignment****Figur 2.12:** Jernbanediagram for <assignment>**call-stmt****Figur 2.13:** Jernbanediagram for <call-stmt>**2.2.3.4 for-setninger**

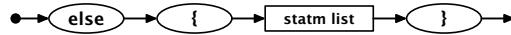
En for-setning er en slags utvidelse av while-setningen: i tillegg til slutt-testen kan den ha en initieringsdel som utføres aller først og en oppdateringsdel som utføres etter hvert gjennomløp av løkken.

for-stmt**Figur 2.14:** Jernbanediagram for <for-stmt>**for-control****Figur 2.15:** 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 double) angir *usann*, mens alle andre verdier angir *sann*.

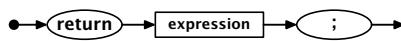
if-stmt**Figur 2.16:** Jernbanediagram for <if-stmt>

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

else-part**Figur 2.17:** Jernbanediagram for <else-part>**2.2.3.6 return-setninger**

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

return-statm



Figur 2.18: Jernbanediagram for <return-statm>

2.2.3.7 while-setninger

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

while-statm



Figur 2.19: Jernbanediagram for <while-statm>

2.2.4 Uttrykk

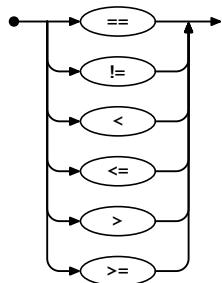
Uttrykk i C_b er en delmengde av de vi kjenner fra C og Java. Definisjonen er delt i <expression>, <term>, <factor> og <operand> slik at vi kan håndtere operatorenes **presedens**³ riktig. C_b krever ellers at begge operandene er av samme type. Figur 2.30 på side 30 viser hva som er lovlig.

expression



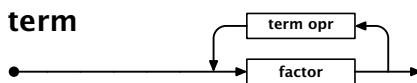
Figur 2.20: Jernbanediagram for <expression>

rel opr



Figur 2.21: Jernbanediagram for <rel opr>

term



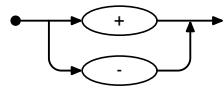
Figur 2.22: Jernbanediagram for <term>

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

$$a + b \times c$$

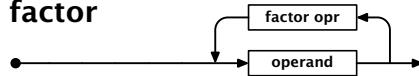
tolkes dette vanligvis som $a + (b \times c)$ fordi \times normalt har høyere presedens enn $+$, dvs \times binder sterkere enn $+$.

term opr



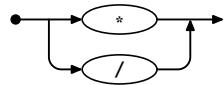
Figur 2.23: Jernbanediagram for <term opr>

factor



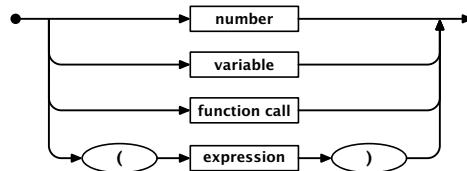
Figur 2.24: Jernbanediagram for <factor>

factor opr



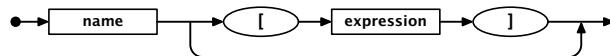
Figur 2.25: Jernbanediagram for <factor opr>

operand



Figur 2.26: Jernbanediagram for <operand>

variable



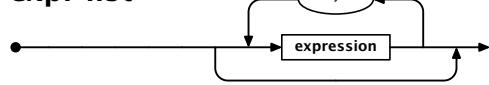
Figur 2.27: Jernbanediagram for <variable>

function call



Figur 2.28: Jernbanediagram for <function call>

expr list



Figur 2.29: Jernbanediagram for <expr list>

```

int iv;    double fv;

iv + iv      OK          iv = iv      OK
iv + fv      Ulovlig!    iv = fv      OK
fv + iv      Ulovlig!    fv = iv      OK
fv + fv      OK          fv = fv      OK

```

Figur 2.30: Lovlige og ulovlige typekombinasjoner for operander og tilordninger

2.2.4.1 Biblioteket

C_b kjenner til syv biblioteksfunksjoner som er vist i tabell 2.1. Disse kan brukes uten noen spesiell spesifikasjon. Funksjonene getchar og putchar forutsetter at tastaturet og skjermen benytter ISO 8859-1; se avsnitt 6.2 på side 54.

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

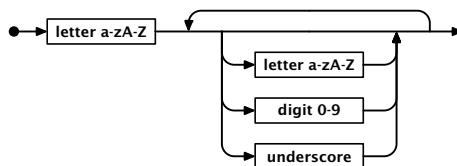
Tabell 2.1: C_bs biblioteksfunksjoner

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.

name

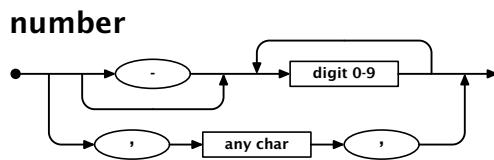


Figur 2.31: Jernbanediagram for <name>

2.2.4.3 Tall og tegn

Heltall brukes som forventet i C_b.⁴ I tillegg er det lov å bruke tegnkonstanter (men altså ikke tekster på mer enn ett tegn). Slike tegnkonstanter betraktes som tall der verdien er tegnets representasjon i ISO 8859-1; se tabell 2.2 på side 32.

⁴ Det er én markant forskjell mellom C og C_b når det gjelder heltall: heltall i C_b kan ha fortegn. Dette betyr at uttrykket a-1 tolkes som de to symbolene [a] og [-1] og dette vil gi en feilmelding om manglende operator. Uttrykket blir korrekt om vi skriver et mellomrom mellom - og 1.

**Figur 2.32:** Jernbanediagram for <number>

Det finnes ingen flyttallskonstanter i C_b. Om man for eksempel ønsker å angi π , må man deklarere:

```
double pi; double mill;
pi = 3141593; mill = 1000000; pi = pi/mill;
```

2.2.5 Andre ting

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

- Som i C kan man 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.

2.3 Forskjeller til C

C_b er en forenklet C, så det er mange elementer i C som mangler. Her er noen av dem, sortert etter viktighet (slik vi ser det):

- Det er bare to datatyper: `int` og `double`. Det er altså ingen `void` og heller ingen `struct`-er, `union`-er eller `typedef`-er.
- C_b er ganske restriktiv når det gjelder typer: begge operander til +, == etc må være av samme type.
- Funksjoner kan ikke ha vektorer som parametre.
- Det er ingen `do-while`- eller `switch`-setninger.
- Det er ingen preprosessordirektiver («#xxx-linjer»).⁵
- Funksjonsbiblioteket er minimalt sammenlignet med C.

⁵ # -linjer er lov i C_b, men betraktes kun som kommentarer.

ISO 8859-1

0	000	32	040	64	@	100	96	'	140	128	200	160	240	192	À	300	224	à	340		
	00		20		40		60		80		A0		C0		E0						
1	001	33	!	041	65	A	101	97	a	61	81	161	i	241	193	Á	301	225	á	341	
	01		21		41		41		82		A1		C1		E1						
2	002	34	"	042	66	B	102	98	b	62	142	130	202	162	242	194	Â	302	226	â	342
	02		22		42		42		83		A2		C2		E2						
3	003	35	#	043	67	C	103	99	c	63	143	131	203	163	243	195	Ã	303	227	ã	343
	03		23		43		43		84		A3		C3		E3						
4	004	36	\$	044	68	D	104	100	d	64	144	132	204	164	244	196	Ä	304	228	ä	344
	04		24		44		44		84		A4		C4		E4						
5	005	37	%	045	69	E	105	101	e	65	145	133	205	165	245	197	Å	305	229	å	345
	05		25		45		45		85		A5		C5		E5						
6	006	38	&	046	70	F	106	102	f	66	146	134	206	166	246	198	Æ	306	230	æ	346
	06		26		46		46		86		A6		C6		E6						
7	007	39	,	047	71	G	107	103	g	67	147	135	207	167	247	199	§	307	231	ç	347
	07		27		47		47		87		A7		C7		E7						
8	010	40	(050	72	H	110	104	h	68	150	136	210	168	250	200	È	310	232	è	350
	08		28		48		48		88		A8		C8		E8						
9	011	41)	051	73	I	111	105	i	69	151	137	211	169	251	201	É	311	233	é	351
	09		29		49		49		89		A9		C9		E9						
10	012	42	*	052	74	J	112	106	j	6A	152	138	212	170	252	202	Ê	312	234	ê	352
	0A		2A		4A		4A		AA		AA		CA		EA						
11	013	43	+	053	75	K	113	107	k	6B	153	139	213	171	253	203	Ë	313	235	ë	353
	0B		2B		4B		4B		BB		AB		CB		EB						
12	014	44	,	054	76	L	114	108	l	6C	154	140	214	172	254	204	Ì	314	236	ì	354
	0C		2C		4C		4C		8C		AC		CC		EC						
13	015	45	-	055	77	M	115	109	m	6D	155	141	215	173	255	205	Í	315	237	í	355
	0D		2D		4D		4D		BD		AD		CD		ED						
14	016	46	.	056	78	N	116	110	n	6E	156	142	216	174	256	206	Î	316	238	î	356
	0E		2E		4E		4E		BE		AE		CE		EE						
15	017	47	/	057	79	O	117	111	o	6F	157	143	217	175	257	207	Ï	317	239	ï	357
	0F		2F		4F		4F		BF		AF		CF		EF						
16	020	48	0	060	80	P	120	112	p	6G	160	144	220	176	260	208	Ð	320	240	ð	360
	10		30		50		50		70		70		80		F0						
17	021	49	1	061	81	Q	121	113	q	71	161	145	221	177	261	209	Ñ	321	241	ñ	361
	11		31		51		51		91		B1		D1		F1						
18	022	50	2	062	82	R	122	114	r	72	162	146	222	178	262	210	Ò	322	242	ò	362
	12		32		52		52		92		B2		D2		F2						
19	023	51	3	063	83	S	123	115	s	73	163	147	223	179	263	211	Ó	323	243	ó	363
	13		33		53		53		93		B3		D3		F3						
20	024	52	4	064	84	T	124	116	t	74	164	148	224	180	264	212	Ô	324	244	ô	364
	14		34		54		54		94		B4		D4		F4						
21	025	53	5	065	85	U	125	117	u	75	165	149	225	181	265	213	Ñ	325	245	ñ	365
	15		35		55		55		95		B5		D5		F5						
22	026	54	6	066	86	V	126	118	v	76	166	150	226	182	266	214	Ö	326	246	ö	366
	16		36		56		56		96		B6		D6		F6						
23	027	55	7	067	87	W	127	119	w	77	167	151	227	183	267	215	×	327	247	÷	367
	17		37		57		57		97		B7		D7		F7						
24	030	56	8	070	88	X	130	120	x	78	170	152	230	184	270	216	Ø	330	248	ø	370
	18		38		58		58		98		B8		D8		F8						
25	031	57	9	071	89	Y	131	121	y	79	171	153	231	185	271	217	Ù	331	249	ù	371
	19		39		59		59		99		B9		D9		F9						
26	032	58	:	072	90	Z	132	122	z	7A	172	154	232	186	272	218	Ú	332	250	ú	372
	1A		3A		5A		5A		9A		BA		DA		FA						
27	033	59	;	073	91	[133	123	{	7B	173	155	233	187	273	219	Û	333	251	û	373
	1B		3B		5B		5B		9B		BB		DB		FB						
28	034	60	<	074	92	\	134	124		7C	174	156	234	188	274	220	Ü	334	252	ü	374
	1C		3C		5C		5C		9C		BC		DC		FC						
29	035	61	=	075	93]	135	125	}	7D	175	157	235	189	275	221	Ý	335	253	ý	375
	1D		3D		5D		5D		9D		BD		DD		FD						
30	036	62	>	076	94	^	136	126	~	7E	176	158	236	190	276	222	Þ	336	254	þ	376
	1E		3E		5E		5E		9E		BE		DE		FE						
31	037	63	?	077	95	—	137	127	—	7F	177	159	237	191	277	223	ß	337	255	ÿ	377
	1F		3F		5F		5F		9F		BF		DF		FF						

© April 1995, DFL, Hi/UiO

Tabell 2.2: Tegnsettet ISO 8859-1

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 omgang¹ 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-bits² prosessor som inneholder fire viktige deler:

Logikkenheten tolker instruksjonene; med andre ord utfører den programkoden. I tabell 3.1 på side 35 er vist de instruksjonene vi vil benytte i prosjektet vårt.

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

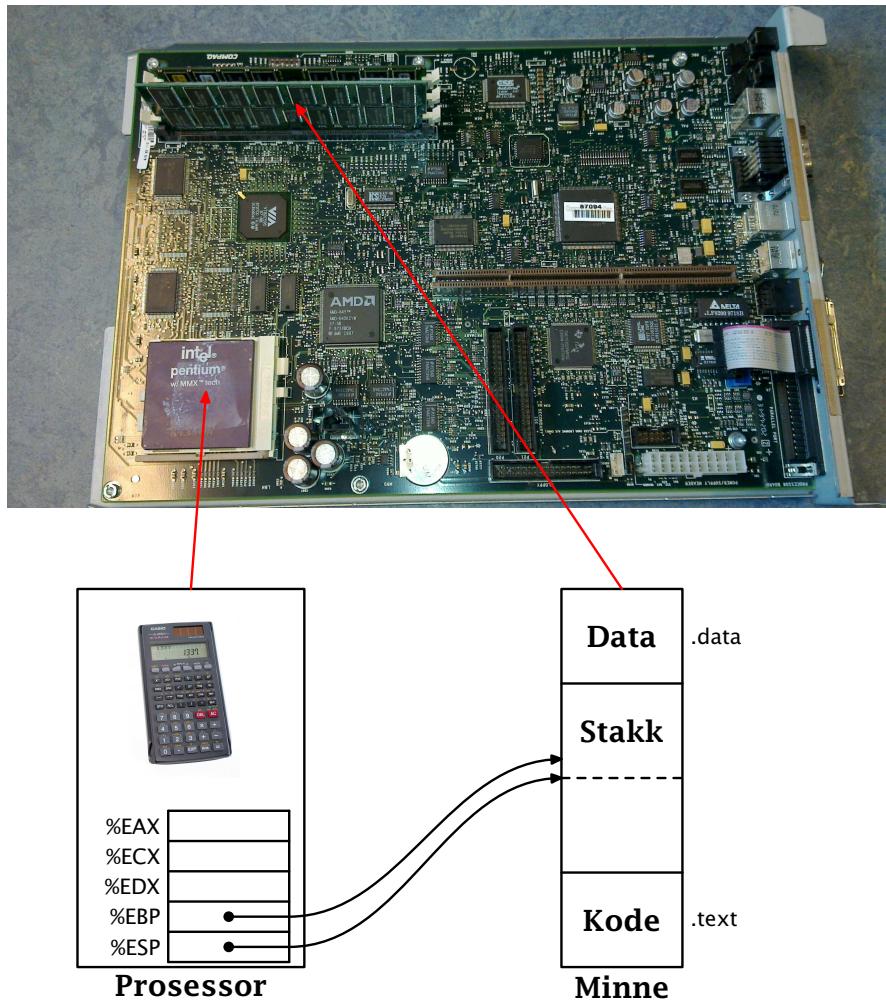
Registrene er spesielle heltallsvariabler som er ekstra tett koblet til regneenheten. Vi skal bruke disse registrene:

%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.

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

² 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

Flyttallsenheten er en egen enhet som kan regne med flyttall. Den inneholder 8 registre organisert som en stakk.

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>mov</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 neste side.

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

<code>movl</code>	<code><iv₁>, <iv₂></code>	Flytt <code><iv₁></code> til <code><iv₂></code> .
<code>cdq</code>		Omform 32-bits <code>%EAX</code> til 64-bits <code>%EDX:%EAX</code> .
<code>leal</code>	<code><iv₁>, <iv₂></code>	Flytt <code><iv₁></code> s <i>adresse</i> til <code><iv₂></code> .
<code>pushl</code>	<code><iv></code>	Legg <code><iv></code> på stakken.
<code>popl</code>	<code><iv></code>	Fjern toppen av stakken og legg verdien i <code><iv></code> .
<code>addl</code>	<code><iv₁>, <iv₂></code>	Adder <code><iv₁></code> til <code><iv₂></code> .
<code>subl</code>	<code><iv₁>, <iv₂></code>	Subtraher <code><iv₁></code> fra <code><iv₂></code> .
<code>imull</code>	<code><iv₁>, <iv₂></code>	Multipliser <code><iv₁></code> med <code><iv₂></code> .
<code>idivl</code>	<code><iv></code>	Del <code>%EDX:%EAX</code> med <code><iv></code> ; svar i <code>%EAX</code> .
<code>call</code>	<code><lab></code>	Kall funksjonen i <code><lab></code> .
<code>enter</code>	<code>\$<n>, \$0</code>	Start en funksjon med <code><n></code> byte lokale variabler.
<code>leave</code>		Rydd opp når funksjonen er ferdig.
<code>ret</code>		Returner fra funksjonen.
<code>cmpl</code>	<code><iv₁>, <iv₂></code>	Sammenligning <code><iv₁></code> og <code><iv₂></code> .
<code>jmp</code>	<code><lab></code>	Hopp til <code><lab></code> .
<code>je</code>	<code><lab></code>	Hopp til <code><lab></code> hvis <code>=</code> .
<code>sete</code>	<code><iv></code>	Sett <code><iv>=1</code> om <code>=</code> , ellers <code><iv>=0</code> .
<code>setne</code>	<code><iv></code>	Sett <code><iv>=1</code> om <code>≠</code> , ellers <code><iv>=0</code> .
<code>setl</code>	<code><iv></code>	Sett <code><iv>=1</code> om <code><</code> , ellers <code><iv>=0</code> .
<code>setle</code>	<code><iv></code>	Sett <code><iv>=1</code> om <code>≤</code> , ellers <code><iv>=0</code> .
<code>setg</code>	<code><iv></code>	Sett <code><iv>=1</code> om <code>></code> , ellers <code><iv>=0</code> .
<code>setge</code>	<code><iv></code>	Sett <code><iv>=1</code> om <code>≥</code> , ellers <code><iv>=0</code> .
<code>faddp</code>		Adder to flyttall.
<code>fdivp</code>		Divider ett flyttall på et annet.
<code>fildl</code>	<code><iv></code>	Legg heltall i <code><iv></code> på stakken som flyttall.
<code>fistpl</code>	<code><iv></code>	Lagre flyttall som <code>int</code> i <code><iv></code> .
<code>fldl</code>	<code><fv></code>	Legg flyttall på stakken.
<code>fldz</code>		Legg 0,0 på stakken.
<code>fmulp</code>		Multipliser to flyttall.
<code>fstpl</code>	<code><fv></code>	Lagre flyttall som <code>double</code> i <code><fv></code> .
<code>fstsps</code>	<code><fv></code>	Lagre flyttall som <code>float</code> i <code><fv></code> .
<code>fsubp</code>		Subtraher ett flyttall fra et annet.

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

- `<iv>` kan være en konstant (`«$17»`),
et register (`«%EAX»`),
en global variabel (`«Var»`),
en lokal variabel (`«-4(%EBP)»`) eller
en parameter(`«8(%EBP)»`).
- `<fv>` er en flyttallsvariabel.
- `<lab>` er en merkelapp som angir en minnelokasjon.

%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 4.1.3 på neste side).

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 kommentarer. 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.

	.data	Angi at vi nå skal plassere variabler i datadelen av minnet.
(lab):	.fill n	Sett av <i>n</i> byte i minnet (til en variabel eller en vektor).
	.globl <lab>	Navnet <i>lab</i> skal være kjent utenfor denne filen.
	.text	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 Heltallsregistere

Vi vil bruke disse heltallsregistrene:

%EAX er det viktigste arbeidsregisteret for heltall. Alle heltallsuttrykk eller -deluttrykk skal produsere et resultat i %EAX.

%ECX er et hjelpperegister som brukes ved aritmetiske eller sammenligningsoperatorer med heltall eller til indeks ved oppslag i vektorer.

%EDX brukes til vektoradresser og som hjelpperegister ved heltallsdivisjon.

%ESP peker på toppen av kjørestakkene.

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

4.1.2 Flyttallsregistere

Flyttall legges alltid på toppen av flyttallsstakken og hentes også derfra. Vi vil bare bruke de to øverste posisjonene.

For konvertering mellom heltall og flyttall trenger vi dessuten en egen minnelokasjon. Denne genereres automatisk i starten av alle kodelfiler og heter **.tmp**.

4.1.3 Navn

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

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

Globale variabler får også beholde navnet brukt i C-b-koden.¹

Parametre trenger ikke navn i assemblerkoden siden de er gitt utfra posisjonen i parameterlisten: For heltall (dvs av typen `int`) er første parameter `8(%ebp)`, andre parameter `12(%ebp)`, tredje parameter `16(%ebp)` osv. For flyttallsparametre (dvs av typen `double`) øker tallet med 8 i stedet.

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` om det er en `int`-verdi eller på toppen av flyttallsstakken om det er en `double`-verdi.

4.2.1 Operander i uttrykk

I tabell 4.1 på neste side er vist hvilken kode som må genereres for å hente en heltallsverdi $\langle n \rangle$, en enkel `int`-variabel $\langle iv \rangle$, en enkel `double`-variabel $\langle fv \rangle$, et vektorelement $\langle ia \rangle[\langle ie \rangle]$ eller $\langle fa \rangle[\langle ie \rangle]$ eller et uttrykk i parenteser $(\langle ie \rangle)$ eller $(\langle fe \rangle)$ inn i register `%EAX` eller på flyttallsstakken.

(Kode for funksjonskall er ikke tatt med her – den er beskrevet i avsnitt 4.4.2 på side 43.)

4.2.2 Operatorer i uttrykk

I tabell 4.2 på side 40 er vist hvordan man oversetter en heltallsaddisjon; subtraksjon gjøres på samme måte (med en `subl`-instruksjon) og også multiplikasjon (med en `imull`-instruksjon). Divisjon av heltall krever et litt annet opplegg, som vist i tabellen.

Naturlig nok oversettes flyttallsoperasjonene litt anderledes; dette er vist i tabell 4.3 på side 40. De andre tre operasjonene bruker henholdsvis `fsubp`, `fmulp` og `fdivp`.

I tabell 4.4 på side 41 er vist hvordan man oversetter en likhetstest (`==`) mellom to heltall og mellom to flyttall; de ulike testene bruker operasjonene vist i tabell 4.5 på side 41.

¹ 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 referansekompiletoren.

$\langle n \rangle$	\Rightarrow	<code>movl \$n,%eax</code>
$\langle iv \rangle$	\Rightarrow	<code>movl iv,%eax</code>
$\langle fv \rangle$	\Rightarrow	<code>fild fv</code>
$\langle ia \rangle [\langle ie \rangle]$	\Rightarrow	<code>Beregn ie med svar i %EAX</code> <code>leal ia,%edx</code> <code>movl (%edx,%eax,4),%eax</code>
$\langle fa \rangle [\langle ie \rangle]$	\Rightarrow	<code>Beregn ie med svar i %EAX</code> <code>leal fa,%edx</code> <code>fild (%edx,%eax,8)</code>
$(\langle ie \rangle)$	\Rightarrow	<code>Beregn ie med svar i %EAX</code>
$(\langle fe \rangle)$	\Rightarrow	<code>Beregn fe med svar på flyttallsstakken</code>

Tabell 4.1: Kode generert av operand i uttrykk

4.3 Oversettelse av setninger

4.3.1 Oversettelse av tomme setninger

Som vist i tabell 4.6 på side 41 er det meget enkelt å oversette tomme setninger.

4.3.2 Oversettelse av tilordninger

Det er i utgangspunktet enkelt å lage kode for tilordninger, men det er 8 varianter å ta hensyn til: variabelen på venstresiden kan være `int` eller `double` og den kan være en vanlig enkel variabel eller et vektorelement, mens høyresiden kan være et `int`- eller et `double`-uttrykk. I tabell 4.7 på side 42 er det vist fire av variantene – du må selv finne ut hvilken kode som skal lages for de andre.

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 43. Merk at om funksjonen er en `double`-funksjon, må vi rydde opp på flyttallsstakken og fjerne verdien på toppen; se tabell 4.8 på side 42.

$\langle ie_1 \rangle + \langle ie_2 \rangle$	\Rightarrow	<p>(Beregn $\langle ie_1 \rangle$ med svar i %EAX) pushl %eax (Beregn $\langle ie_2 \rangle$ med svar i %EAX) movl %eax,%ecx popl %eax addl %ecx,%eax</p>
$\langle ie_1 \rangle / \langle ie_2 \rangle$	\Rightarrow	<p>(Beregn $\langle ie_1 \rangle$ med svar i %EAX) pushl %eax (Beregn $\langle ie_2 \rangle$ med svar i %EAX) movl %eax,%ecx popl %eax cdq idivl %ecx</p>

Tabell 4.2: Kode generert av heltallsoperatorene + og /

$\langle fe_1 \rangle + \langle fe_2 \rangle$	\Rightarrow	<p>(Beregn $\langle fe_1 \rangle$ med svar på flyttallsstakken) subl \$8,%esp fstppl (%esp) (Beregn $\langle fe_2 \rangle$ med svar på flyttallsstakken) fildl (%esp) addl \$8,%esp faddp</p>
---	---------------	--

Tabell 4.3: Kode generert av flyttallsoperatoren +

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 4.9 på side 43 viser oversettelse av en if-setning der testen er en **int**-verdi og en der den er en **double**.² I tillegg er det vist en if-setning med en else-gren, men bare med **int**-test; **double**-varianten skjønner du sikker selv hvordan skal se ut.

4.3.6 Oversettelse av return-setninger

En return-setning innebærer to ting:

² En **double** brukt som testverdi er ikke særlig vanlig, men det er lov i C, så da må vi ta hensyn til det.

$\langle ie_1 \rangle == \langle ie_2 \rangle$	\Rightarrow	(Beregn $\langle ie_1 \rangle$ med svar i %EAX) pushl %eax (Beregn $\langle ie_2 \rangle$ med svar i %EAX) popl %ecx cmpl %eax,%ecx movl \$0,%eax sete %al
$\langle fe_1 \rangle == \langle fe_2 \rangle$	\Rightarrow	(Beregn $\langle fe_1 \rangle$ med svar på flyttallsstakken) subl \$8,%esp fstpl (%esp) (Beregn $\langle fe_2 \rangle$ med svar på flyttallsstakken) fldl (%esp) addl \$8,%esp fsubp fstps .tmp cmpl \$0,.tmp movl \$0,%eax sete %al

Tabell 4.4: Kode generert av operatoren ==

$==$	sete
$!=$	setne
$<$	setl
\leq	setle
$>$	setg
\geq	setge

Tabell 4.5: Instruksjoner til de ulike sammenligningsoperatorene

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

Tabell 4.6: Kode generert av tom setning

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

Koden for dette er vist i tabell 4.10 på side 43. (I avsnitt 4.4.2 på side 43 står det hvor slutten er plassert.) Dette gjelder uansett om resultatverdien er en `int` eller en `double`.

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 4.11 på side 43 når testen er en `int`; husk at den også kan være en `double`.

4.4 Oversettelse av funksjoner og funksjonskall

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

$\langle iv \rangle = \langle ie \rangle;$	\Rightarrow	(Beregn $\langle ie \rangle$ med svar i %EAX) <code>movl %eax, \langle iv \rangle</code>
$\langle fv \rangle = \langle fe \rangle;$	\Rightarrow	(Beregn $\langle fe \rangle$ med svar på flyttallsstakken) <code>fstp1 \langle fv \rangle</code>
$\langle fa \rangle [\langle ie_1 \rangle] = \langle ie_2 \rangle;$	\Rightarrow	(Beregn $\langle ie_1 \rangle$ med svar i %EAX) <code>pushl %eax</code> (Beregn $\langle ie_2 \rangle$ med svar i %EAX) <code>leal \langle fa \rangle, %edx</code> <code>popl %ecx</code> <code>movl %eax, .tmp</code> <code>fildl .tmp</code> <code>fstp1 (%edx, %ecx, 8)</code>
$\langle ia \rangle [\langle ie \rangle] = \langle fe \rangle;$	\Rightarrow	(Beregn $\langle ie \rangle$ med svar i %EAX) <code>pushl %eax</code> (Beregn $\langle fe \rangle$ med svar på flyttallsstakken) <code>leal \langle ia \rangle, %edx</code> <code>popl %ecx</code> <code>fistpl (%edx, %ecx, 4)</code>

Tabell 4.7: Kode generert av tilordning

$\langle intf \rangle (\dots);$	\Rightarrow	$\langle \text{Kall på } \langle intf \rangle \rangle$
$\langle df \rangle (\dots);$	\Rightarrow	$\langle \text{Kall på } \langle df \rangle \rangle$ <code>fstsps .tmp</code>

Tabell 4.8: Kode generert av funksjonskall som setning

4.4.1 Oversettelse av funksjonsdeklarasjoner

Som vist i figur 4.12 på side 44 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 side 40. Vi må også bruke `leave`-instruksjonen til å frigjøre plassen vi satte av til lokale variabler før vi hopper tilbake med en `ret`.

4.4 OVERSETTELSE AV FUNKSJONER OG FUNKSJONSKALL

<code>if (<ie>) {<S>}</code>	⇒	(Beregner <ie> med svar i %EAX) <code>cmpl \$0,%eax</code> <code>je <lab></code> <code><S></code> <code><lab>:</code>
<code>if (<fe>) {<S>}</code>	⇒	(Beregner <fe> med svar på flyttallsstakken) <code>fstsps .tmp</code> <code>cmpl \$0,.tmp</code> <code>je <lab></code> <code><S></code> <code><lab>:</code>
<code>if (<ie>) {<S₁>} else {<S₂>}</code>	⇒	(Beregner <ie> med svar i %EAX) <code>cmpl \$0,%eax</code> <code>je <lab₁></code> <code><S₁></code> <code>jmp <lab₂></code> <code><lab₁>:</code> <code><S₂></code> <code><lab₂>:</code>

Tabell 4.9: Kode generert av if-setning

<code>return (e);</code>	⇒	(Beregner <e>) <code>jmp .exit\$f</code>
--------------------------	---	---

Tabell 4.10: Kode generert av return-setning

<code>while (<ie>) {<S>}</code>	⇒	<code><lab₁>:</code> (Beregner <ie> med svar i %EAX) <code>cmpl \$0,%eax</code> <code>je <lab₂></code> <code><S></code> <code>jmp <lab₁></code> <code><lab₂>:</code>
---	---	--

Tabell 4.11: Kode generert av while-setning

- Hvis funksjonen er en `double`-funksjon, må vi rett før sluttetiketten legge inn en instruksjon som legger en verdi (f eks 0,0) på flyttallsstakken. Dette er for å sikre at det alltid finnes en returverdi selv om brukeren har droppet retur-setningen.

4.4.2 Oversettelse av funksjonskall

Slik tabell 4.13 på neste side viser, oversettes et funksjonskall til tre kodesekvenser:

<pre>int <intf> (int <ip₁>, ...) { <D> <S> }</pre>	⇒	<pre>.globl <intf> <intf>: enter \$<antall byte i <D>>,\$0 <S> .exit\$<intf>: leave ret</pre>
<pre>double <df> (int <ip₁>, ...) { <D> <S> }</pre>	⇒	<pre>.globl <df> <df>: enter \$<antall byte i <D>>,\$0 <S> fldz .exit\$<df>: leave ret</pre>

Tabell 4.12: Kode generert av funksjonsdeklarasjon

- 1) Parametrene legges på stakken (i *omvendt rekkefølge*). `int`-parametre lagres i 4 byte mens `double`-parametre trenger 8.
- 2) Funksjonen kalles.
- 3) Parametrene fjernes fra stakken.

I eksemplet har funksjonen to parametre, en `int` og en `double`, så 12 byte må fjernes fra stakken etterpå. Det bør være enkelt å generalisere dette til å ha et vilkårlig antall parametre, inkludert 0.

<pre>(f)(<ie>, <fe>)</pre>	⇒	<pre>(Beregner <fe> med svar på flyttallsstakken) subl \$8,%esp fstpl (%esp) (Beregner <ie> med svar i %EAX) pushl %eax call <f> addl \$12,%esp</pre>
--	---	--

Tabell 4.13: 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 om de er av typen `int` og 8 byte om de er `double`, så de deklarereres som vist i tabell 4.14 på neste side. Til vektorer setter vi av 4 eller 8 byte til hvert element.

<code>int <iv>;</code>	\Rightarrow	<code>(iv): .globl <iv></code> <code>.fill 4</code>
<code>double <fa>[<n>];</code>	\Rightarrow	<code><fa>: .globl <fa></code> <code>.fill <8n></code>

Tabell 4.14: 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 variabeldeklarasjoner, må vi ta med en

`.text`

for å skifte til kodedelen av minnet.

4.5.2 Deklarasjon av lokale variabler

Funksjonen sørger selv for å sette av plass til sine lokale variabler på stakken (se tabell 4.12 på forrige side).

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.uioifi.cflat.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 navn¹ som pakken. Denne klassen inneholder data og metoder som «hører hjemme i» pakken, spesielt de to metodene² **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 Cflat

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

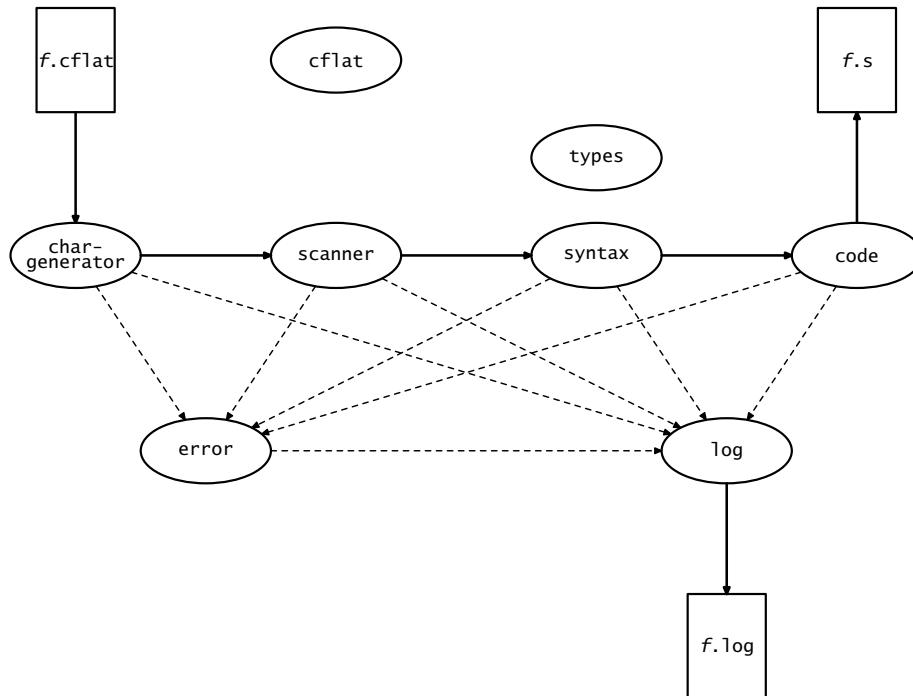
Denne modulen vil lese kildefilen linje for linje, ignorere #-linjer og sende den resterende kildekoden tegn for tegn videre i to variabler: **curC** (med nåværende tegn) og **nextC** (med neste tegn). Metoden **readNext** vil klargjøre neste tegn.

5.3 Modulen Scanner

Denne modulen vil få tegn fra kildefilen (via `CharGenerator`) og levere fra seg symboler i variablene **curToken**, **nextToken** og **nextNextToken**;

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

² Disse metodene vil være `static` siden vi aldri skal instansiere disse spesialklassene.



Figur 5.1: Modulene i komplataoren

```

package no.uio_ifi_cflat.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

disse variablene er av klassen **Token**. **curToken** er det aktuelle symbolet vi skal analysere, mens **nextToken** og **nextNextToken** er de to etterfølgende symbolene. Variablene **curLine**, **nextLine** og **nextNextLine** 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** og for **nextNextName** og **nextNextNum**.

Metoden **readNext** vil plassere de neste symbolene i **curToken**, **nextToken** og **nextNextToken**. Alle /*...*/-kommentarer vil bli oversett.

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

5.4 Modulen Syntax

Denne modulen tar seg av analysen av C_b-programmet; slik analyse kalles **parsing** (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.cflat** i figur 6.2 på side 58 har for eksempel et syntakstre som vist i figur 5.3 på neste side.⁴

Modulen inneholder klassen **SyntaxUnit** og diverse subklasser som brukes når parsingstreet skal bygges. Disse implementerer ulike utgaver av disse metodene:⁵

parse vil parsere akkurat denne noden i treet 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

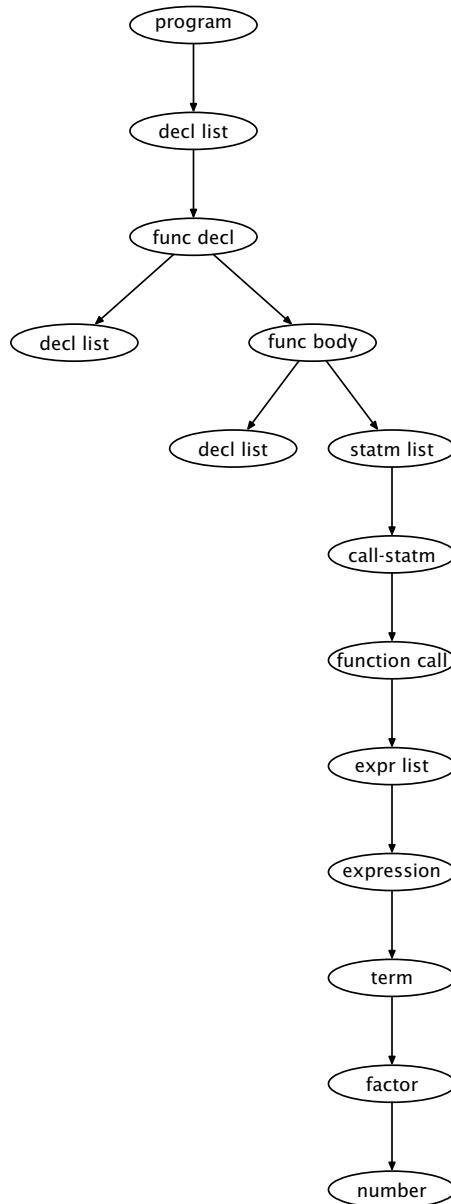
Denne modulen inneholder det som trengs for å jobbe med typer. Den inneholder disse klassene (i tillegg til **Types**):

Type er «moderklassen» til alle type-klassene.

³ «Eof» er en vanlig forkortelse for «end of file».

⁴ Det er ikke gitt at syntakstreet til **mini.cflat** skal se nøyaktig slik ut – det vil være små variasjoner avhengig av hvilke klasser man definerer i kompilatoren.

⁵ 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: Et parseringstre

BasicType representerer de grunnleggende typene (i vårt tilfelle `double` og `int`).

ArrayType er for vektorer.

I tillegg inneholder Types de to variablene `doubleType` og `intType` som representerer akkurat de to typene; begge er laget av anonyme subklasser av **BasicType**.

5.6 Modulen Code

Denne modulen tar seg av 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 øyeblikkelig og lukker filen etter hver linje slik at innholdet bevares om kompilatoren krasjer. Modulen har disse nyttige metodene:

noteError noterer en feilmelding, men bare om loggfilen er i bruk.

enterParser brukes under parseringen til å vise at en ny **parse**-metode kalles, men bare om **doLogParser** er satt.

leaveParser brukes tilsvarende når en parseringsmetode forlates, men også her bare om **doLogParser** er satt.

noteSourceLine legger inn en kildekodelinje i loggen for å vise hvor langt lesingen er kommet; dette skjer kun når **doLogParser** eller **doLogScanner** er satt.

noteToken benyttes av skanneren til å logge hvilke symboler den finner; loggingen skjer bare når **doLogScanner** er satt.

wTree og **wTreeLn** brukes til skrive ut parсерstreet; de virker som `System.out.print` og `System.out.println`.

indentTree brukes når utskriften av parсерstreet skal indenteres et {}-nivå.

outdentTree benyttes når et slikt nivå skal avsluttes.

noteBinding angir hvilken navnebinding som ble gjort, men bare om **noteBinding** er satt.

Kapittel 6

Prosjektet

Som nevnt er C_b-kompilatoren et større program enn dere sannsynligvis har skrevet før, så prosjektet er delt i tre deler: en minimal introduksjonsdel og to omtrent 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  
$ cd inf2100  
$ ant
```

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

```
$ java -jar Cflat.jar minfil.cflat
```

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

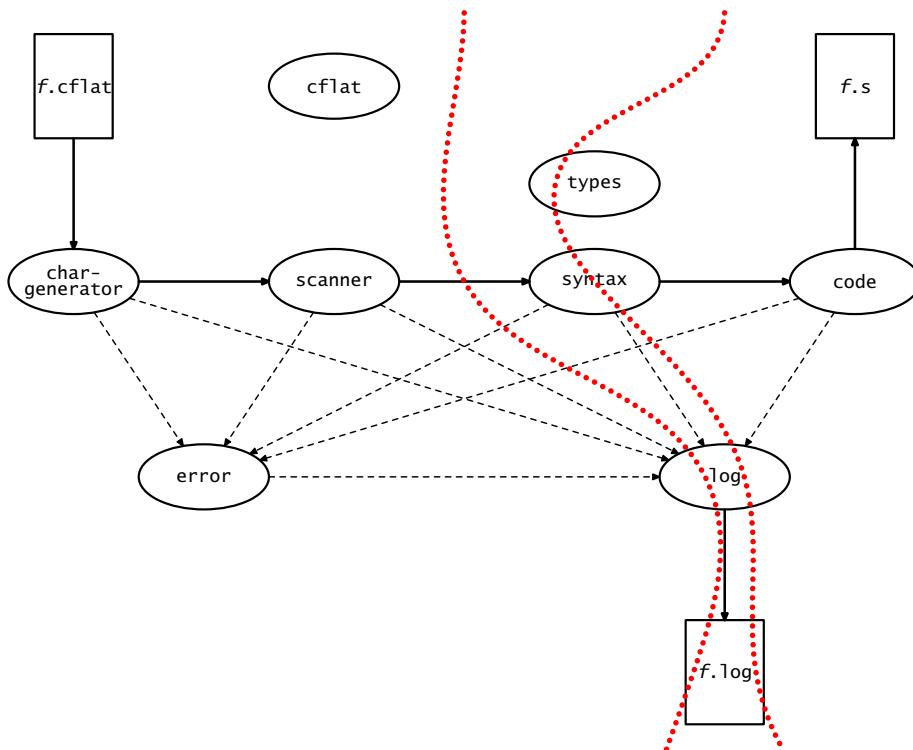
Prosjektet er utviklet på Universitetets Linux-maskiner, men det er også mulig å gjennomføre programmeringen på egen datamaskin, uansett om den kjører Linux, Mac OS X eller Windows. Det er imidlertid ditt ansvar at nødvendige verktøy fungerer skikkelig. Du trenger:

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>-logP</code>	Del 1	Hvilke parseringsmetoder kalles
<code>-logS</code>	Del 0	Hvilke symboler hentes fra skanneren
<code>-logT</code>	Del 1	Utskrift av parsingstreet

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>.

Et redigingsprogram 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 minst tre tegnkoderingar som er i vanlig bruk i Norge:

Iso 8859-1 (også kalt «Latin-1») er et tegnsett der hvert tegn lagres i én byte; se oversikten over tegnene i tabell 2.2 på side 32. Vår kompilator

forutsetter at C_b-programmene er lagret i denne kodingen, så sorg for å sette dette i oppsettet for den editoren du bruker.¹

Iso 8859-15 (også kalt «Latin-9») er en lett modernisert variant av ISO 8859-1. Til vår bruk i dette prosjektet kan vi betrakte de to som like.

UTF-8 er en lagringsform for **Unicode**-kodingen og bruker 1–4 byte til hvert tegn. Selve kompilatoren (dvs Java-programmet) kan godt lagres i UTF-8, men C_b-programmene må lagres som Iso 8859-1 (se over).

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 Cflat.jar -testscanner mini.cflat
$ java -jar Cflat.jar -testscanner gcd.cflat
```

der det første testprogrammet er vist i figur 6.2 på side 58 og det andre i 6.8 på side 59. De resulterende filene **mini.log** og **gcd.log** skal da se ut som vist² i henholdsvis figur 6.3 på side 58 og figurene 6.9 og 6.10 på side 60 og siden etter.

Mål for del 0

Programmet skal utvikles slik at opsjonen **-testscanner** produserer loggfiler som vist i figurene 6.3, 6.9 og 6.10.

6.4 Del I

Denne delen går ut på å få parseren til å fungere. Dette innebærer å skrive alle klassene som tilsvarer metasymbolene. Ved å kjøre

```
> java -jar Cflat.jar -testparser gcd.cflat
```

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 **-logT**, som også settes av **-testparser**). Våre vanlige testprogram vist i henholdsvis figur 6.2 på side 58 og figur 6.8 på side 59 vil produsere loggfiler i henholdsvis figur 6.4 på side 58 og figurene 6.11 til 6.15 på side 62 og etterfølgende.³

¹ Siden de såkalte ASCII-tegnene (dvs venstre halvpart av tabell 2.2) lagres likt i alle de aktuelle tegnsettene, er det ikke nødvendig å bry seg med dette om man bare bruker dem, dvs unngår særnorske bokstaver som ÅØÅ og lignende.

² 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.

³ Denne loggutskriften stammer også fra flere kilder, så det er også her helt OK at linjene stokkes om i forhold til det som vises.

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 6.11–6.16.

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 C_b-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 6.8) skal se ut som vist i henholdsvis figur 6.7 på side 59 og figurene 6.17 til 6.18 på side 67–68. (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 producere data om dette som vist i figur 6.6 og 6.19; gal navnebruk og typefeil skal gi feilmeldinger. Dessuten skal kompilatoren generere en kodelfil som vist i figur 6.7 og 6.17–6.18.

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 treet og noen på vei tilbake.

6.5.1.1 Sjekke navn ved deklarasjoner

Det må sjekkes at navn er deklarert riktig. I C_b er dette enkelt, for det er bare én mulig feil: å deklarere navn flere ganger i samme liste.⁴

⁴ 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`.

6.5.1.2 Sjekke navnebruk

Dette innebærer å se på alle navnforekomster 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 vektorelement. 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.) Typefastsettelse starter i bladnodene i syntakstreet og går oppover:

- Heltallskonstanter er alltid av typen `int`.
- Variabler er av den typen de ble deklarert som.

Basert på dette kan vi fastsette hvilken type de ulike operasjonene har:

- Funksjonskall er av typen funksjonen er definert som.
- Aritmetiske operatorer ('+', '*' etc) har samme type som alle operandene (og alle disse må ha samme type, se avsnitt 6.5.1.4).
- Sammenligningsoperatorer ('==' , '<=' etc) gir alltid en `int`-verdi.

6.5.1.4 Sjekke typer

Følgende typeregler må sjekkes:

- En vektorindeks må alltid være en `int`.
- Begge operandene til en gitt operator (enten det er '+', '/', '!=', '<=' eller en av de andre) må ha samme type.
- I et funksjonskall må alle parametrene ha den typen som er angitt i funksjonsdeklarasjonen.
- Uttrykket i en return-setning må ha den typen som funksjonsdeklarasjonen tilsier.

Legg merke at vi ikke trenger å sjekke typer ved tilordning; her er det lov å ha `int` og `double` på begge sider av likhetstegnet i alle kombinasjoner.

```

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

mini.cflat

Figur 6.2: Et minimalt C_b-program mini.cflat

```

1 1: # Program 'mini'
2 2: #
3 3: # A minimal Cb program!
4 4:
5 5: int main ()
6 Scanner: intToken
7 Scanner: nameToken main
8 Scanner: leftParToken
9 Scanner: rightParToken
10 10: {
11 Scanner: leftCurlToken
12 12:     putchar('x');
13 Scanner: nameToken putchar
14 Scanner: leftParToken
15 Scanner: numberToken 120
16 Scanner: rightParToken
17 Scanner: semicolonToken
18 18: }
19 Scanner: rightCurlToken
20 Scanner: eofToken
```

Figur 6.3: Loggfil som demonstrerer hvilke symboler skanneren finner i mini.cflat

```

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

Figur 6.4: Loggfil som viser parsering av mini.cflat

```

1 Tree: int main ()
2 Tree: {
3 Tree:     putchar(120);
4 Tree: }
```

Figur 6.5: Loggfil med trerrepresentasjonen av mini.cflat

```

1 Binding: Line 7: putchar refers to declaration in the library
2 Binding: main refers to declaration in line 5

```

Figur 6.6: Logfil med navnebindinger foretatt for mini.cflat

```

1      .data
2 .tmp:   .fill    4          # Temporary storage
3      .text
4      .globl  main
5 main:   enter   $0,$0        # Start function main
6      movl    $120,%eax       # 120
7      pushl    %eax          # Push parameter #1
8      call    putchar         # Call putchar
9      addl    $4,%esp         # Remove parameters
10     .exit$main:
11     leave
12     ret                  # End function main

```

Figur 6.7: Kodefil laget fra mini.cflat

gcd.cflat

```

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     while (a != b) {
10         if (a < b) {
11             b = b-a;
12         } else {
13             a = a-b;
14         }
15     }
16     return a;
17 }
18
19 int main ()
20 {
21     int v1;  int v2;
22
23     LF = 10; ;
24     putchar('?' );  putchar(' ');
25     v1 = getInt();  v2 = getInt();
26     putInt(gcd(v1,v2));  putchar(LF);
27     exit(0);
28 }

```

Figur 6.8: Et litt større C_b-program gcd.less

```

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 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 6.9: Loggfil som demonstrerer hvilke symboler skanneren finner i gcd. Tess (del I)

```

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 6.10: Logfil som demonstrerer hvilke symboler skanneren finner i gcd.cflat (del 2)

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

Figur 6.11: Loggfil som viser parsering av gcd.cflat (del 1)

```

76 Parser:          <factor>
77 Parser:          <operand>
78 Parser:          <variable>
79 Parser:          </variable>
80 Parser:          </factor>
81 Parser:          </term operator>
82 Parser:          <term operator>
83   13: } else {
84 Parser:          </term operator>
85 Parser:          <factor>
86 Parser:          <operand>
87 Parser:          <variable>
88 Parser:          </variable>
89 Parser:          </factor>
90 Parser:          </term>
91 Parser:          </expression>
92 Parser:          </assignment>
93 Parser:          </assign statm>
94 Parser:          </statement>
95 Parser:          </statm list>
96 Parser:          <else-part>
97   14: a = a-b;
98 Parser:          <statm list>
99 Parser:          <statement>
100 Parser:          <assign-statm>
101 Parser:          <assignment>
102 Parser:          <variable>
103 Parser:          </variable>
104 Parser:          <expression>
105 Parser:          <term>
106 Parser:          <factor>
107 Parser:          <operand>
108 Parser:          <variable>
109 Parser:          </variable>
110 Parser:          </factor>
111 Parser:          </term operator>
112 Parser:          <term operator>
113 Parser:          <factor>
114   15: }
115 Parser:          </term operator>
116 Parser:          <factor>
117 Parser:          <operand>
118 Parser:          <variable>
119   16: }
120 Parser:          </variable>
121 Parser:          </operand>
122 Parser:          </factor>
123 Parser:          </term>
124 Parser:          </expression>
125 Parser:          </assignment>
126   17: return a;
127 Parser:          </assign statm>
128 Parser:          </statement>
129 Parser:          </statm list>
130 Parser:          </else-part>
131 Parser:          </if-statm>
132 Parser:          </statement>
133 Parser:          </statm list>
134 Parser:          </while-statm>
135 Parser:          </statement>
136 Parser:          <return-statm>
137 Parser:          <expression>
138   18: }
139 Parser:          <term>
140 Parser:          <factor>
141 Parser:          <operand>
142 Parser:          <variable>
143 Parser:          </variable>
144   19:
145   20: int main ()
146 Parser:          </variable>
147 Parser:          </operand>
148 Parser:          </factor>
149 Parser:          </term>
150 Parser:          </expression>

```

Figur 6.12: Logfil som viser parsing av gcd.cflat (del 2)

```

151 Parser:          </return-statm>
152 Parser:          </statement>
153 Parser:          </statm list>
154 Parser:          </func body>
155 Parser:          </func decl>
156 Parser:          <func decl>
157     21: {
158         22:   int v1; int v2;
159     Parser:          <func body>
160     Parser:          <var decl>
161     Parser:          </var decl>
162     Parser:          <var decl>
163     23:
164         24:   LF = 10; ;
165     Parser:          </var decl>
166     Parser:          <statm list>
167     Parser:          <statement>
168     Parser:          <assign-statm>
169     Parser:          <assignment>
170     Parser:          <variable>
171     Parser:          </variable>
172     Parser:          <expression>
173     Parser:          <term>
174     Parser:          <factor>
175     Parser:          <operand>
176     Parser:          <number>
177     25:   putchar('?');  putchar(' ');
178     Parser:          </number>
179     Parser:          </operand>
180     Parser:          </factor>
181     Parser:          </term>
182     Parser:          </expression>
183     Parser:          </assignment>
184     Parser:          </assign statm>
185     Parser:          </statement>
186     Parser:          <statement>
187     Parser:          <empty statm>
188     Parser:          </empty statm>
189     Parser:          </statement>
190     Parser:          <statement>
191     Parser:          <call-statm>
192     Parser:          <function call>
193     Parser:          <expr list>
194     Parser:          <expression>
195     Parser:          <term>
196     Parser:          <factor>
197     Parser:          <operand>
198     Parser:          <number>
199     Parser:          </number>
200     Parser:          </operand>
201     Parser:          </factor>
202     Parser:          </term>
203     Parser:          </expression>
204     Parser:          </expr list>
205     Parser:          </function call>
206     Parser:          </call-statm>
207     Parser:          </statement>
208     Parser:          <statement>
209     Parser:          <call-statm>
210     Parser:          <function call>
211     Parser:          <expr list>
212     Parser:          <expression>
213     Parser:          <term>
214     Parser:          <factor>
215     Parser:          <operand>
216     Parser:          <number>
217     26:   v1 = getInt();   v2 = getInt();
218     Parser:          </number>
219     Parser:          </operand>
220     Parser:          </factor>
221     Parser:          </term>
222     Parser:          </expression>
223     Parser:          </expr list>
224     Parser:          </function call>
225     Parser:          </call-statm>

```

Figur 6.13: Logfil som viser parsering av gcd.cflat (del 3)

```

226 Parser:      </statement>
227 Parser:      <statement>
228 Parser:          <assign-stm>
229 Parser:              <assignment>
230 Parser:                  <variable>
231 Parser:                      </variable>
232 Parser:                  <expression>
233 Parser:                      <term>
234 Parser:                          <factor>
235 Parser:                              <operand>
236 Parser:                                  <function call>
237 Parser:                                      <expr list>
238 Parser:                                          </expr list>
239 Parser:                                              </function call>
240 Parser:                                          </operand>
241 Parser:                                      </factor>
242 Parser:                                  </term>
243 Parser:                                      <expression>
244 Parser:                                          </assignment>
245 Parser:                                              </assign statm>
246 Parser:                                          </statement>
247 Parser:      <statement>
248 Parser:          <assign-stm>
249 Parser:              <assignment>
250 Parser:                  <variable>
251 Parser:                      </variable>
252 Parser:                  <expression>
253 Parser:                      <term>
254 Parser:                          <factor>
255 Parser:                              <operand>
256 Parser:                                  <function call>
257     27: putint(gcd(v1,v2)); putchar(LF);
258 Parser:                                  <expr list>
259 Parser:                                      </expr list>
260 Parser:                                              </function call>
261 Parser:                                          </operand>
262 Parser:                                      </factor>
263 Parser:                                  </term>
264 Parser:                                      <expression>
265 Parser:                                          </assignment>
266 Parser:                                              </assign statm>
267 Parser:      </statement>
268 Parser:      <statement>
269 Parser:          <call-stm>
270 Parser:              <function call>
271 Parser:                  <expr list>
272 Parser:                      <expression>
273 Parser:                          <term>
274 Parser:                              <factor>
275 Parser:                                  <operand>
276 Parser:                                      <function call>
277 Parser:                                          <expr list>
278 Parser:                                              <expression>
279 Parser:                      <term>
280 Parser:                          <factor>
281 Parser:                              <operand>
282 Parser:                                  <variable>
283 Parser:                                      </variable>
284 Parser:                                  <operand>
285 Parser:                          </factor>
286 Parser:                      </term>
287 Parser:                      <expression>
288 Parser:                      <expression>
289 Parser:                          <term>
290 Parser:                              <factor>
291 Parser:                                  <operand>
292 Parser:                                      <variable>
293 Parser:                                  </variable>
294 Parser:                                  <operand>
295 Parser:                          </factor>
296 Parser:                      </term>
297 Parser:                      <expression>
298 Parser:                          </expr list>
299 Parser:                                              </function call>
300 Parser:                                          </operand>

```

Figur 6.14: Logfil som viser parsering av gcd.cflat (del 4)

```

301 Parser:           </factor>
302 Parser:           </term>
303 Parser:           </expression>
304 Parser:           </expr list>
305 Parser:           </function call>
306 Parser:           </call-stm>
307 Parser:           </statement>
308 Parser:           <statement>
309 Parser:           <call-stm>
310 Parser:           <function call>
311 Parser:           <expr list>
312 Parser:           <expression>
313 Parser:           <term>
314 Parser:           <factor>
315 Parser:           <operand>
316 Parser:           <variable>
317     28:   exit(0);
318 Parser:           </variable>
319 Parser:           </operand>
320 Parser:           </factor>
321 Parser:           </term>
322 Parser:           </expression>
323 Parser:           </expr list>
324 Parser:           </function call>
325 Parser:           </call-stm>
326 Parser:           </statement>
327 Parser:           <statement>
328 Parser:           <call-stm>
329 Parser:           <function call>
330 Parser:           <expr list>
331 Parser:           <expression>
332 Parser:           <term>
333 Parser:           <factor>
334 Parser:           <operand>
335 Parser:           <number>
336     29: }
337 Parser:           </number>
338 Parser:           </operand>
339 Parser:           </factor>
340 Parser:           </term>
341 Parser:           </expression>
342 Parser:           </expr list>
343 Parser:           </function call>
344 Parser:           </call-stm>
345 Parser:           </statement>
346 Parser:           </stmt list>
347 Parser:           </func body>
348 Parser:           </func decl>
349 Parser:           </program>

```

Figur 6.15: Logfil som viser parsering av gcd.cflat (del 5)

```

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 6.16: Logfil med trerepresentasjonen av gcd.cflat

```

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

```

Figur 6.17: Kodefil produsert fra gcd.cflat (del 1)

```

76      addl    $4,%esp          # Remove parameters
77      movl    LF,%eax          # LF
78      pushl    %eax            # Push parameter #1
79      call    putchar           # Call putchar
80      addl    $4,%esp          # Remove parameters
81      movl    $0,%eax          # 0
82      pushl    %eax            # Push parameter #1
83      call    exit              # Call exit
84      addl    $4,%esp          # Remove parameters
85 .exit$main:
86      leave
87      ret                     # End function main

```

Figur 6.18: Kodefil produsert fra gcd.cflat (del 2)

```

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: Logfil med navnebindinger foretatt for gcd.cflat

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 73.)
- 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`-indeks er helt OK:

Type navn	Kapitalisering	Hva slags ord	Eksempel
Klasser	XxxxxXxxxx	Substantiv som beskriver objektene	IfStatement
Metoder	xxxxxXxxxx	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

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

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 på neste side 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.

7.1.5 Utseende

7.1.5.1 Linjelengde og linjedeling

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

```
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

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/javadoc/writingdoccomments/>.

En JavaDoc-kommentarer 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 \TeX . 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 \LaTeX) og kan benytte alle tilgjengelige typografiske hjelpeemidler 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 bøblesortering. 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
```

¹ Dette eksemplet bruker Dags versjon av lesbar programmering kalt `web0`; for mer informasjon, se [/local/opt/web0/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; \footnote{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. 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

June 24, 2013

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;¹ 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).⟩
```

¹We 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 $\langle \text{swap } a[i] \text{ and } a[i+1] \rangle \equiv$
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 $\langle \text{local variables } \#4 \text{ (p.1)} \rangle + \equiv$
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 $\langle \text{initialize} \rangle \equiv$
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 $\langle \text{not sorted} \rangle \equiv$
18 `n_swaps > 0`
(This code is used in #2 (p.1).)

File: *bubble.w0*

page 2

Figur 8.5: «Lesbar programmering» — utskrift side 2

Variables

A

a1, 12, 14

I

i 11, 12, 13, 14

N

n1, 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 $a[i]$ and $a[i+1]$ #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, 38
.exit\$f, 37
.tmp, 37
ant, 53
Array, 25
Assembler, 18
Assemblerspråk, 18
Cb, 23
Cflat, 47
CharGenerator, 47
check, 49
Code, 50
curC, 47
curLine, 49
curName, 49
curNum, 49
curToken, 47, 49
doLogParser, 51
doLogScanner, 51
double, 25
emacs, 54
enterParser, 51
eofToken, 49
Error, 51
error, 51
Exception, 51
Feil, 51
finish, 47
gas, 53
gcc, 53
genCode, 49
Høynivå programmeringsspråk, 13
indentTree, 51
init, 47
Initialverdi, 25
int, 25
Interpreter, 16
java, 53
Java-pakker, 47
javac, 53, 54
JavaDoc, 73
Kommandospråk, 16
Kompilator, 13
leaveParser, 51
Linux, 53
Log, 51
Mac OS X, 53
main, 25, 47
Maskinspråk, 13
nameToken, 49
nextC, 47
nextLine, 49
nextName, 49
nextNextLine, 49
nextNextName, 49
nextNextNum, 49
nextNextToken, 47, 49
nextNum, 49
nextToken, 47, 49
noteBinding, 51
noteError, 51
noteSourceLine, 51
noteToken, 51
numberToken, 49
outdentTree, 51
package, 47
parse, 49, 51, 55
Parsing, 49
Parseringstre, 49
Preprocessor, 15
Presedens, 28
printTree, 49
Programmeringsstil, 69
readNext, 47, 49
Scanner, 47
Skanner, 19
StreamTokenizer, 19
Symboler, 19
Syntaks, 19
Syntakstre, 19, 49
Syntax, 49
SyntaxUnit, 49
Token, 49
Tokenizer, 19
Tokens, 19

Types, 49
Unicode, 55
Vektor, 25
Virtuelle, 49
Windows, 53
wTree, 51
wTreeLn, 51