UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

En kompilator for Pascal

Kompendium for INF2100

Stein Krogdahl, Dag Langmyhr

Høsten 2015

Innhold

Figurer

Tabeller

Forord

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

Det opprinnelige kurset ble utviklet av *Stein Krogdahl* rundt 1980 og dreide seg om å skrive en kompilator som oversatte det Simula-lignende språket *Minila* til kode for en tenkt datamaskin *Flink*; implementasjonsspråket var Simula. I 1999 gikk man over til å bruke Java som implementasjonsspråk, og i 2007 ble kurset fullstendig renovert av *Dag Langmyhr*: Minila ble erstattet av en minimal variant av C kalt *RusC* og datamaskinen Flink ble avløst av en annen ikkeeksisterende maskin kalt *Rask*. I 2010 ble det besluttet å lage ekte kode for Intel-prosessoren x86 slik at den genererte koden kunne kjøres direkte på en datamaskin. Dette medførte så store endringer i språket RusC at det fikk et nytt navn: *C<* (uttales «c less»). Ønsker om en utvidelse førte i 2012 til at det ble innført datatyper (`int` og `double`) og språket fikk igjen et nytt navn: *Cb* (uttales «c flat»). Tilbakemelding fra studentene avslørte at de syntes det ble veldig mye fikling å lage kode for `double`, så i 2014 ble språket endret enda en gang. Under navnet *AlboC* hadde det nå pekere i stedet for flyt-tall.

Nå er det blitt 2015, og hele opplegget har gått gjennom en revisjon. Det gjelder også språket som skal kompileres: nå er det *Pascal2100* som utgjør mesteparten av gode, gamle *Pascal*.

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

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

Blindern, 19. august 2015
Stein Krogdahl *Dag Langmyhr*

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

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

— Forfatterne

Kapittel 1

Innledning

1.1 Hva er emnet INF2100?

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

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

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

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

I dette kompendiet beskrives stort sett bare selve programmerings-oppgaven 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 nettsider.

1.2 Hvorfor lage en kompilator?

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

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

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

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

Ut fra disse kriteriene synes ett felt å peke seg ut som spesielt fristende, nemlig det å skrive en kompilator, altså et program som oppfører seg 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 uoversiktlig oppgave. Noe av poenget med kurset er å demonstrere at med en hensiktsmessig oppsplitting av programmet i deler som hver tar ansvaret for en avgrenset del av oppgaven, så kan både de enkelte deler og den helheten de danner, bli høyst medgjørlig. Det er denne erfaringen, og forståelsen av hvordan slik oppdeling kan gjøres på et reelt eksempel, som er det viktigste studentene skal få med seg fra dette kurset.

Vi skal i neste avsnitt se litt mer på hva en kompilator er og hvordan den står i forhold til liknende verktøy. Det vil da også raskt bli klart at det å skrive en kompilator for et «ekte» programmeringsspråk som skal oversettes til maskinspråket til en datamaskin, vil bli en altfor omfattende oppgave. Vi skal derfor forenkle oppgaven en del ved å lage vårt eget lille programmeringsspråk **Pascal2100**. Vi skal i det følgende se litt nærmere på dette og andre elementer som inngår i oppgaven.

1.3 Om kompilatorer og liknende verktøy

Mange 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 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 1–3 milliarder instruksjoner per sekund.

For å kunne få utført programmer skrevet for eksempel i C, lages det spesielle programmer som kan *oversette* C-programmet til en tilsvarende sekvens av maskininstruksjoner for en gitt maskin. Det er slike oversettelsesprogrammer som kalles kompilatorer. En kompilator 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 kompilatoren skal oversette fra), og data det leverer fra seg er en sekvens av maskininstruksjoner for den aktuelle maskinen. Disse maskininstruksjonene vil kompilatoren 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 kompilator 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 kompilatorer.

1.3.1 Preprosessorer

I stedet for å kompilere til en sekvens av maskininstruksjoner finnes det også noen kompilatorer 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++-kompilator oversette det videre til maskinkode. Vi sier da gjerne at denne Java-«kompilatoren» er en **preprosessor** til C++-kompilatoren.

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

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

1.3.2 Interpretering

Det er også en annen måte å utføre et program skrevet i et passelig programmeringsspråk på, og den kalles **interpretering**. 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. Interpretering 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-interpretering, det gjelder for eksempel Basic. Det finnes imidlertid nå kompilatorer også for disse språkene.

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

Interpretering 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 interpretering og kompilering. Noe av koden kompileres helt, mens andre kodebiter oversettes til et mellomnivåspråk som er bedre egnet for interpretering – og som da interpreteres under kjøring. Smalltalk, Perl og Python er eksempler på språk som ofte er implementert slik.

Interpretering 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 simuleringsprogram interpreterer maskinkoden til JVM-maskinen. I dag har for eksempel de fleste nettlesere (Firefox, Opera,

Chrome og andre) innebygget en slik JVM-interpreter for å kunne utføre Java-applets når de får disse (ferdig kompilert) over nettet.

Slik interpretering av maskinkode går imidlertid normalt en del saktere enn om man hadde oversatt til «ekte» maskinkode og kjørt den direkte på «hardware». Typisk kan dette for Javas byte-kode gå 2 til 10 ganger så sakte. Etter hvert som Java er blitt mer populært har det derfor også blitt behov for systemer som kjører Java-programmer raskere, og den vanligste måten å gjøre dette på er å utstyre JVM-er med såkalt «Just-In-Time» (JIT)-kompilering. Dette vil si at man i stedet for å interpretere byte-koden, oversetter den videre til den aktuelle maskinkoden umiddelbart før programmet startes opp. Dette kan gjøres for hele programmer, eller for eksempel for klasse etter klasse etterhvert som de tas i bruk første gang.

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

1.4 Språkene i oppgaven

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

1.4.1 Programmeringsspråket Pascal2100

Det å lage en kompilator for til dømes Java ville sprenge 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 Pascal kalt *Pascal2100*. 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 Pascal2100 blir beskrevet i detalj i kapittel 2 på side 21.

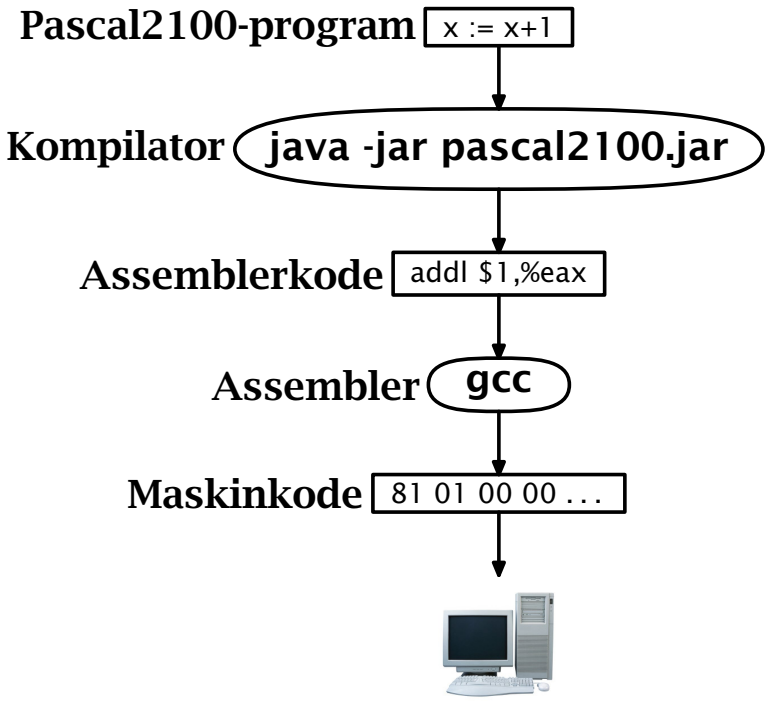
1.4.2 Prosessoren x86 og dens maskinspråk

En kompilator produserer vanligvis kode for en gitt prosessor og det skal også vår Pascal2100-kompilator gjøre. Som prosessor er valgt *x86* siden den finnes overalt, for eksempel på Ifis datalaber og i de aller fleste hjemmemaskiner. Dermed kan dere teste koden uansett hvor dere måtte befinne dere.

Emnet INF2100 vil langtfra 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 maskininstruksjoner, 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.



Figur 1.1: Sammenhengen mellom Pascal2100, kompilator, assembler og en x86-maskin

For eksempel kan man bruke «`addl`» for en instruksjon som legger sammen 32-bits heltall i stedet for til dømes tallet 129 ($= 81_{16}$), som kan være instruksjonens egentlige kode. Man lager så et enkelt program som oversetter sekvenser av slike tekstlige instruksjoner til utførbar maskinkode, og disse oversetterprogrammene kalles tradisjonelt **assemblere**. 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) **Pascal2100**, som kompilatoren skal oversette fra.
2) **Java**, som Pascal2100-kompilatoren skal skrives i.
3) **x86 assemblerkode** er en tekstlig form for maskininstruksjoner til x86-maskinen.
4) **x86s maskinspråk**, som assembleren skal oversette til.

1.5 Oppgaven og dens fire deler

Oppgaven skal løses i fire 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 annonsert i god tid.

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

1.5.1 Del 1: Skanneren

Første skritt, del 1, består i å få Pascal2100s **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 Pascal2100-språket.

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

1.5.2 Del 2: Parseren

Del 2 vil ta imot den symbolsekvensen som blir produsert av del 1, og det sentrale arbeidet her vil være å sjekke at denne sekvensen har den formen et riktig Pascal2100-program skal ha (altså, at den følger Pascal2100s **syntaks**).

Om alt er i orden, skal del 2 bygge opp et **syntakstre**, en **trestruktur** av objekter som direkte representerer det aktuelle Pascal2100-programmet, altså hvordan det er satt sammen av «expression» inne i «statement» inne i «func decl» osv. Denne trestrukturen skal så leveres videre til del 3 som grunnlag for sjekking.

1.5.3 Del 3: Sjekking

I del 3 skal man sjekke variabler og funksjoner mot sine deklarasjoner og kontrollere at de er brukt riktig, for eksempel at man ikke kaller på en variabel som om den var en funksjon. Det er også viktig å sjekke typene, slik at man for eksempel ikke tilordner en `Boolean`-verdi til en `Integer`-variabel.

1.5.4 Del 4: Kodegenerering

Til sist kan kompilatoren vår gjøre selve oversettelsen til x86-kode; da tar vi igjen utgangspunkt i den trestrukturen som del 2 produserte for det aktuelle Pascal2100-programmet. Koden skal legges på en fil og den skal være i såkalt x86 assemblerformat.

I avsnitt 4.5 på side 48 er det angitt hvilke sekvenser av x86-instruksjoner hver enkelt Pascal2100-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 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». Slikt har skjedd før.

1.7 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 kompilatoren 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 kompilatoren deres forlengs, baklengs og med bind for øynene.

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

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

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

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

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

1.8 Delta på øvingsgruppene

Ellers vil vi oppfordre studentene til å være aktive på de ukentlige øvingsgruppene. Oppgavene som blir gjennomgått, er meget relevante for skriving av Pascal2100-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

Programmering i Pascal2100

Programmeringsspråket **Pascal2100** er hoveddelen av **Pascal** som var et meget populært språk på 1970- og -80-tallet. Syntaksen er gitt av jernbanediagrammene i figur 2.2 til 2.43 på side 23–30 og bør være lett forståelig for alle som har programmert litt i Java. Et eksempel på et Pascal2100-program er vist i figur 2.1 på neste side.[1]

2.1 Kjøring

Inntil dere selv har laget en Pascal2100-kompilator, kan dere benytte referansekompilatoren:

```
$ ~inf2100/pascal2100 primes.pas
This is the Ifi Pascal2100 compiler (2015-07-27)
Parsing... checking... generating code...OK
Running gcc -m32 -o primes primes.s -L. -L/hom/inf2100 -lpas2100
$ ./primes
   2   3   5   7  11  13  17  19  23  29
  31  37  41  43  47  53  59  61  67  71
  73  79  83  89  97 101 103 107 109 113
 127 131 137 139 149 151 157 163 167 173
 179 181 191 193 197 199 211 223 227 229
 233 239 241 251 257 263 269 271 277 281
 283 293 307 311 313 317 331 337 347 349
 353 359 367 373 379 383 389 397 401 409
 419 421 431 433 439 443 449 457 461 463
 467 479 487 491 499 503 509 521 523 541
 547 557 563 569 571 577 587 593 599 601
 607 613 617 619 631 641 643 647 653 659
 661 673 677 683 691 701 709 719 727 733
 739 743 751 757 761 769 773 787 797 809
 811 821 823 827 829 839 853 857 859 863
 877 881 883 887 907 911 919 929 937 941
 947 953 967 971 977 983 991 997
```

Denne må kjøres på en av Ifis Linux-maskiner.

[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/.

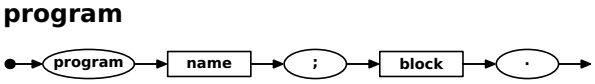
primes.pas

```
/*
This program prints all primes up to Limit
{which is 1000 in this example}
using a technique called 'The sieve of Eratosthenes'.
*/

program Primes;

const Limit = 1000;

var prime : array [2..Limit] of Boolean;
   i      : integer;


procedure FindPrimes;
var i1 : integer;    I2 : Integer;
begin
   i1 := 2;
   while i1 <= Limit div 2 do begin
      i2 := 2*i1;
      while i2 <= Limit do begin
         prime[i2] := false;
         i2 := i2+i1
      end;
      i1 := i1 + 1
   end
end; {FindPrimes}


function NDigits (v : integer): integer;
/* How many digits are there in v (which is >= 0)? */
begin
   if v <= 9 then NDigits := 1
             else NDigits := 1 + Ndigits(v div 10)
end; {NDigits}


procedure P4 (x : integer);
/* *Note* Equivalent to "printf("%4d",x);" in C. */
var NSpaces : integer;
begin
   NSpaces := 4 - NDigits(x);
   while NSpaces > 0 do begin
      write(' ');  NSpaces := NSpaces-1
   end;
   write(x);
end; {P4}


procedure PrintPrimes;
var i       : integer;
   NPrinted : integer;
begin
   i := 2;  NPrinted := 0;
   while i <= Limit do begin
      if prime[i] then begin
         if (NPrinted > 0) and (NPrinted mod 10 = 0) then write(eol);
         P4(i);  NPrinted := NPrinted + 1;
      end;
      i := i + 1;
   end;
   write(eol)
end; {PrintPrimes}


begin {main program}
   i := 2;
   while i <= Limit do begin prime[i] := true;  i := i+1 end;

   /* Find and print the primes: */
   FindPrimes;  PrintPrimes;
end. {main program}
```

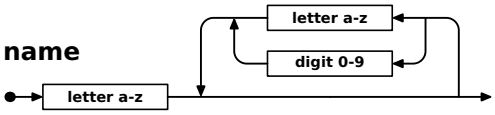
Figur 2.1: Eksempel på et Pascal2100-program

2.2 Pascal2100-program

Som vist i figur 2.2 er et Pascal2100-program rett og slett en ⟨block⟩ med et navn.



Figur 2.2: Jernbanediagram for ⟨program⟩



Figur 2.3: Jernbanediagram for ⟨name⟩

Forskjell fra Java: I Pascal2100 skiller man ikke på store og små bokstaver, så `integer`, `Integer` og `INTEGER` er tre ulike former av samme navn.

2.2.1 Blokker

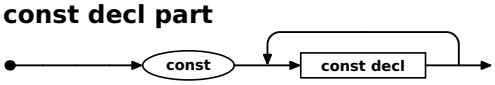
Pascal2100 er et såkalt **blokkstrukturert** programmeringsspråk der alle deklarasjoner er plassert i blokker; se figur 2.17 på side 26. Innmaten av program, funksjoner og prosedyrer er derfor blokker.

Forskjell fra Java: I Pascal2100 kan man ha blokker utenpå hverandre, så man kan ha funksjoner som er lokale inni andre funksjoner. Dette er ikke mulig i Java.

2.2.1.1 Konstantdeklarasjoner

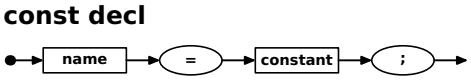
I en blokk kan man deklarere **konstanter**, dvs verdier med et navn. En konstant kan defineres som

- et navn på en annen konstant
- en heltallsliteral[2]
- en tegnliteral (f eks `'A'`)
- en oppramsliteral (f eks `false`)

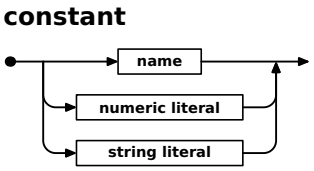


Figur 2.4: Jernbanediagram for ⟨const decl part⟩

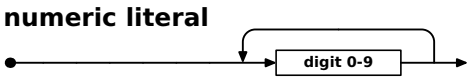
[2] I programmeringsspråk bruker man betegnelsen **literal** på noe som alltid angir «seg selv», for eksempel `123` eller `'x'`. Begrepet **konstant** brukes i stedet om et navn som er deklarert som uforanderlig under kjøringen.



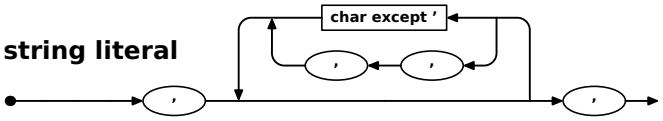
Figur 2.5: Jernbanediagram for ⟨const decl⟩



Figur 2.6: Jernbanediagram for ⟨constant⟩



Figur 2.7: Jernbanediagram for ⟨numeric literal⟩



Figur 2.8: Jernbanediagram for ⟨string literal⟩

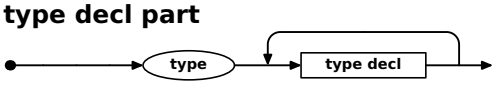
Forskjell fra Java: I Pascal2100 bruker man enkle anførselstegn rundt tekstliteraler. Forekomster av dette tegnet i teksten angis ved å skrive det dobbelt, som i `'O''Malley'`.

I Pascal2100 benytter vi enkle anførselstegn til både enkelttegn av typen `char` og tekster av vilkårlig lengde. Regelen er som følger: Hvis literalen inneholder nøyaktig ett tegn, er det en `char`; hvis ikke, er det en tekst av typen `array[1..`n`] of char` der n er antall tegn.

2.2.1.2 Typedeklarasjoner

I Pascal2100 er det enkelt å definere nye typer; en ny type kan være definert på flere ulike måter:

- et navn på en annen type (f eks `Integer`)
- en intervalltype («range type») der verdien skal ligge i et gitt intervall (f eks `1..10` eller `'a'..'z'`).
- en såkalt «**oppramstype**» («enum type») der man navngir de ulike verdiene (f eks `(Mann, Kvinne)`).
- en array

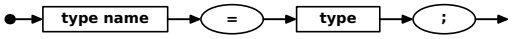


Figur 2.9: Jernbanediagram for ⟨type decl part⟩

2.2.1.3 Variabeldeklarasjoner

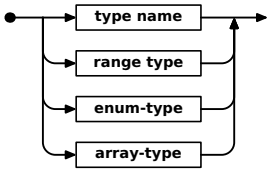
Variabler deklareres med en litt annen syntaks enn vi er vant til fra Java.

type decl



Figur 2.10: Jernbanediagram for ⟨type decl⟩

type



Figur 2.11: Jernbanediagram for ⟨type⟩

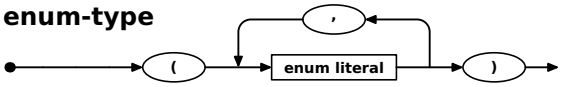
type name

Figur 2.12: Jernbanediagram for ⟨type name⟩

range type

Figur 2.13: Jernbanediagram for ⟨range type⟩

enum-type

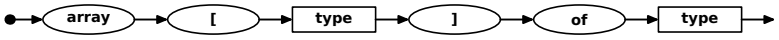


Figur 2.14: Jernbanediagram for ⟨enum type⟩

enum literal

Figur 2.15: Jernbanediagram for ⟨enum literal⟩

array-type



Figur 2.16: Jernbanediagram for ⟨array type⟩

Forskjell fra Java: I Java kan man angi en initialverdi for variabelen; det kan man ikke i Pascal2100.

2.2.1.4 Funksjons- og prosedyredeklarasjoner

I Pascal2100 skiller man mellom **funksjoner** og **prosedyrer**: de førstnevnte returnerer alltid en verdi, de sistnevnte kan ikke det.

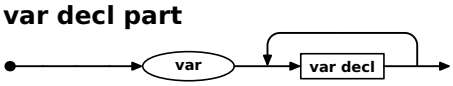
Forskjell fra Java: I Pascal2100 har man ingen **return**-setning; i stedet tilordner man en verdi til funksjonen selv, slik det er vist i funksjonen `NDigits` i eksempelet i figur 2.1 på side 22.

Figur 2.17: Jernbanediagram for ⟨block⟩

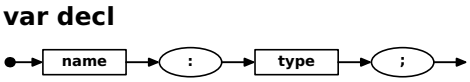
Figur 2.18: Jernbanediagram for ⟨func decl⟩

Figur 2.19: Jernbanediagram for ⟨proc decl⟩

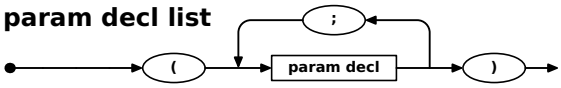
Figur 2.20: Jernbanediagram for ⟨if-statm⟩



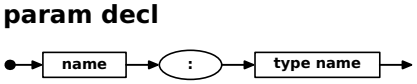
Figur 2.21: Jernbanediagram for ⟨var decl part⟩



Figur 2.22: Jernbanediagram for ⟨var decl⟩



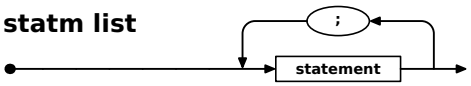
Figur 2.23: Jernbanediagram for ⟨param decl list⟩



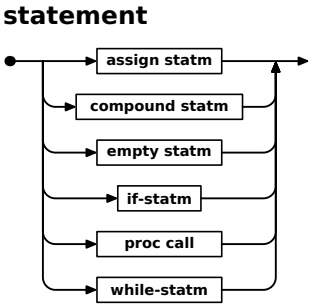
Figur 2.24: Jernbanediagram for ⟨param decl⟩

2.2.2 Setninger

Pascal2100 har mange av de samme setningene som C og Java.



Figur 2.25: Jernbanediagram for ⟨statm list⟩



Figur 2.26: Jernbanediagram for ⟨statement⟩

Forskjell fra Java: I Pascal2100 benyttes semikolon som *skilletegn* mellom setninger, så siste setning før en `end` skal ikke ha noe semikolon. (Men om man legger inn et semikolon der allikevel, betyr det bare at det står en ekstra tom setning før `end`, og det betyr jo ingenting for kjøringen av programmet.)

2.2.2.1 Den tomme setningen

Denne setningen gjør ingenting; den eksisterer bare slik at det skal være lov å ha ekstra semikolon.

2.2.2.2 Tilordning

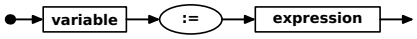
En tilordningssetning beregner et uttrykk og plasserer det i en variabel.

empty statm

Figur 2.27: Jernbanediagram for ⟨empty statm⟩

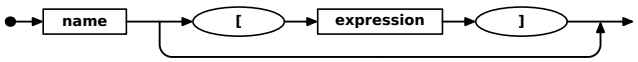
Forskjell fra Java: I Pascal2100 benyttes symbolet **:=** for tilordning.

assign statm



Figur 2.28: Jernbanediagram for ⟨assign statm⟩

variable

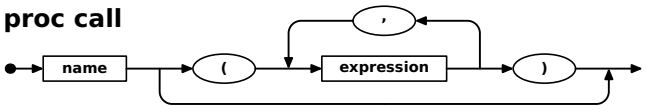


Figur 2.29: Jernbanediagram for ⟨variable⟩

2.2.2.3 Prosedyrekall

Denne setningen kaller en prosedyre.

proc call



Figur 2.30: Jernbanediagram for ⟨proc call⟩

Forskjell fra Java: I Pascal2100 er det ikke lov å kalle en funksjon som om den var en prosedyre.

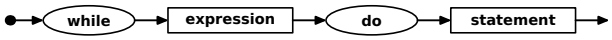
2.2.2.4 If-setninger

If-setninger fungerer slik vi kjenner dem fra andre språk som C og Java.

2.2.2.5 While-setninger

Disse setningene oppfører seg også slik vi er vant til.

while-statm



Figur 2.31: Jernbanediagram for ⟨while-statm⟩

2.2.2.6 Sammensatte setninger

Vi bruker denne konstruksjonen for å sette inn flere setninger der det i henhold til grammatikken bare skal være én. Den tilsvarer altså {...} i Java og C.

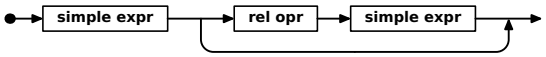
compound statm

Figur 2.32: Jernbanediagram for ⟨compound statm⟩

2.2.3 Uttrykk

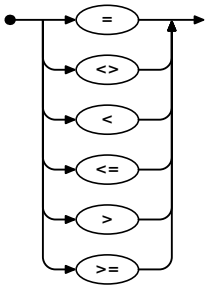
Uttrykk i Pascal2100 er ganske likt de språkene vi kjenner. Det benyttes ganske mange definisjoner for å sikre at presedensen[3] blir riktig.

expression



Figur 2.33: Jernbanediagram for ⟨expression⟩

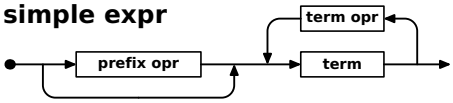
rel opr



Figur 2.34: Jernbanediagram for ⟨rel opr⟩

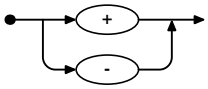
Forskjell fra Java: Testene på likhet og ulikhet angis med henholdsvis = og <> i Pascal2100 (og de tilsvarer altså == og != i Java).

simple expr



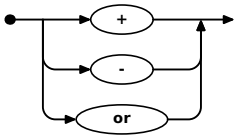
Figur 2.35: Jernbanediagram for ⟨simple expr⟩

prefix opr



Figur 2.36: Jernbanediagram for ⟨prefix opr⟩

term opr



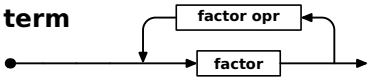
Figur 2.37: Jernbanediagram for ⟨term opr⟩

---

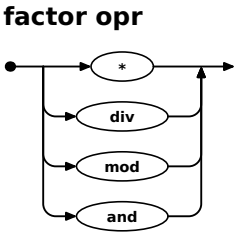
[3] Operatorer har ulik **presedens**, dvs at noen operatorer binder sterkere enn andre. Når vi skriver for eksempel

$$a + b \times c$$

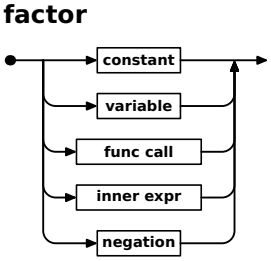
tolkes dette vanligvis som $a + (b \times c)$ fordi $\times$ normalt har høyere presedens enn $+$, dvs $\times$ binder sterkere enn $+$.



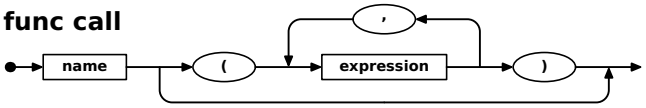
Figur 2.38: Jernbanediagram for ⟨term⟩



Figur 2.39: Jernbanediagram for ⟨factor opr⟩

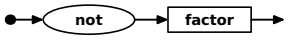


Figur 2.40: Jernbanediagram for ⟨factor⟩



Figur 2.41: Jernbanediagram for ⟨func call⟩

Figur 2.42: Jernbanediagram for ⟨inner expr⟩



Figur 2.43: Jernbanediagram for ⟨negation⟩

2.2.4 Andre ting

Det er et par andre ting man bør merke seg ved Pascal2100:

- Kommentarer kan skrives på to ulike former:

  `/*...*/` `{...}`

  Begge formene kan strekke seg over flere linjer.

2.3 Predefinerte deklarasjoner

I Pascal2100 er disse deklarasjonene ferdig definert og kan brukes i programmene:

```
const eol = '?';  { end-of-line, dvs LF }

type Boolean = (false, true);
     char    = '?' .. '?';  { egentlig NUL .. DEL }
     integer = -2147483648 .. 2147483647;

procedure write (a: ??, b: ??, ...);
```

(Spørsmålstegnene angir verdier som ikke er skrivbare eller ikke fast definert.)

`eol` brukes ved utskrift for å angi linjeskift.

`Boolean` er den velkjente logiske typen som også finnes i Java.

`char` er typen for å lagre enkelttegn (som i Java). I dette kurset er det bare aktuelt å bruke ASCII-tegn (se avsnitt 4.1.7 på side 40).

`integer` er typen for heltall.

`write` er standardprosedyren for utskrift; den er beskrevet i neste avsnitt.

2.3.1 Utskrift

Pascal2100-programmer kan skrive ut ved å benytte den helt spesielle prosedyren `write`. Denne prosedyren kan ha vilkårlig mange parametre, og parametrene kan være av vilkårlig type. Her er noen eksempler:

```
write('Hei!', eol);
write(2, ' + ', 2, ' = ', 2+2, eol);
```

2.4 Forskjeller til standard Pascal

Som nevnt er Pascal2100 en forenklet utgave av Pascal, og følgende er utelatt:

- `case`-, `for`-, `repeat`- og `with`-setningene
- flyt-tall
- filer og alt rundt lesing og skriving (unntatt `write`)
- dynamisk allokering og pekere
- record (som tilsvarer `struct` i C)
- var-, funksjons- og prosedyreparametre
- sjekking under kjøring av array-grenser og andre verdier

Kapittel 3

Datamaskinen x86

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

3.1 Minnet

Minnet[2] består av tre deler:

Datadelen brukes til å lagre globale variabler.

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

Kodedelen inneholder programkoden, altså programmet som utføres.

3.2 Prosessoren x86

x86-prosessoren er en 32-bits[3] 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.

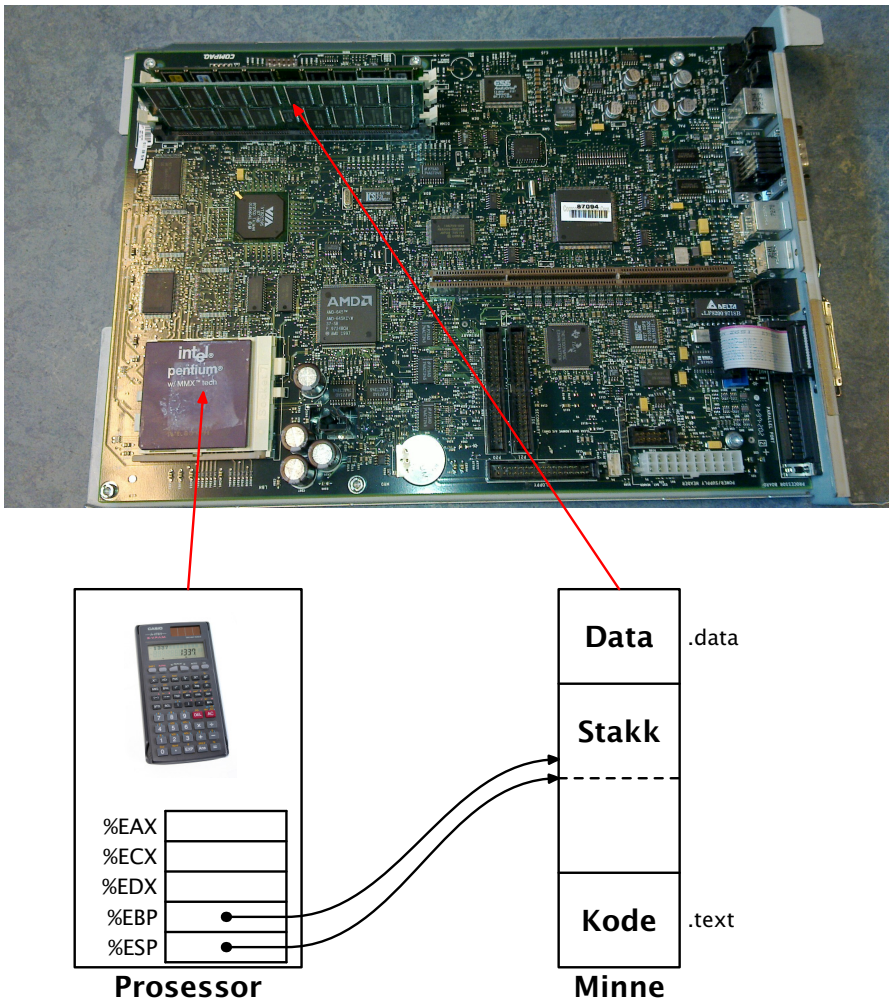
Registrene er spesielle heltallsvariabler som er ekstra tett koblet til regneenheten. Vi skal bruke disse registrene:

%EAX %ECX %EDX %EBP %ESP

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

[2] I figur 3.1 på neste side er datadelen tegnet øverst og kodedelen nederst, men det er ikke alltid slik. Dette har imidlertid ingen betydning for koden vi lager.

[3] 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

`%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.

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.

```
func:    movl    $0,%eax    # Initier til 0.
```

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.

`movl`	$\langle v_1 \rangle$,$\langle v_2 \rangle$	Flytt $\langle v_1 \rangle$ til $\langle v_2 \rangle$.
`cdq`		Omform 32-bits %EAX til 64-bits %EDX:%EAX.
`leal`	$\langle v_1 \rangle$,$\langle v_2 \rangle$	Flytt $\langle v_1 \rangle$s *adresse* til $\langle v_2 \rangle$.
`pushl`	$\langle v \rangle$	Legg $\langle v \rangle$ på stakken.
`popl`	$\langle v \rangle$	Fjern toppen av stakken og legg verdien i $\langle v \rangle$.
`negl`	$\langle v \rangle$	Skift fortegn på $\langle v \rangle$.
`addl`	$\langle v_1 \rangle$,$\langle v_2 \rangle$	Adder $\langle v_1 \rangle$ til $\langle v_2 \rangle$.
`subl`	$\langle v_1 \rangle$,$\langle v_2 \rangle$	Subtraher $\langle v_1 \rangle$ fra $\langle v_2 \rangle$.
`imull`	$\langle v_1 \rangle$,$\langle v_2 \rangle$	Multipliser $\langle v_1 \rangle$ med $\langle v_2 \rangle$.
`idivl`	$\langle v \rangle$	Del %EDX:%EAX med $\langle v \rangle$; svar i %EAX; rest i %EDX.
`andl`	$\langle v_1 \rangle$,$\langle v_2 \rangle$	Logisk AND.
`orl`	$\langle v_1 \rangle$,$\langle v_2 \rangle$	Logisk OR.
`xorl`	$\langle v_1 \rangle$,$\langle v_2 \rangle$	Logisk XOR.
`call`	$\langle$lab$\rangle$	Kall funksjon/prosedyre i $\langle$lab$\rangle$.
`enter`	\$$\langle n_1 \rangle$,\$$\langle n_2 \rangle$	Start funksjon/prosedyre på blokknivå $\langle n_2 \rangle$ med $\langle n_1 \rangle$ byte lokale variabler.
`leave`		Rydd opp når funksjonen/prosedyren er ferdig.
`ret`		Returner fra funksjon/prosedyre.
`cmpl`	$\langle v_1 \rangle$,$\langle v_2 \rangle$	Sammenligning $\langle v_1 \rangle$ og $\langle v_2 \rangle$.
`jmp`	$\langle$lab$\rangle$	Hopp til $\langle$lab$\rangle$.
`je`	$\langle$lab$\rangle$	Hopp til $\langle$lab$\rangle$ hvis =.
`sete`	$\langle v \rangle$	Sett $\langle v \rangle$=1 om =, ellers $\langle v \rangle$=0.
`setne`	$\langle v \rangle$	Sett $\langle v \rangle$=1 om $\neq$, ellers $\langle v \rangle$=0.
`setl`	$\langle v \rangle$	Sett $\langle v \rangle$=1 om $<$, ellers $\langle v \rangle$=0.
`setle`	$\langle v \rangle$	Sett $\langle v \rangle$=1 om $\leq$, ellers $\langle v \rangle$=0.
`setg`	$\langle v \rangle$	Sett $\langle v \rangle$=1 om $>$, ellers $\langle v \rangle$=0.
`setge`	$\langle v \rangle$	Sett $\langle v \rangle$=1 om $\geq$, ellers $\langle v \rangle$=0.

Tabell 3.1: x86-instruksjoner brukt i prosjektet. Følgende symboler er brukt i tabellen:
- $\langle v \rangle$ kan være en konstant («$17»), et register («%EAX»), en lokal variabel («−4(%EBP)») eller en parameter(«8(%EBP)»).
- $\langle n \rangle$ er en heltallskonstant.
- $\langle$lab$\rangle$ er en merkelapp som angir en minnelokasjon.

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

`%EAX` er et register.

`$17` er en tallkonstant.

`f` er navnet på en prosedyre eller en funksjon.

`8(%ESP)` angir en variabel eller en parameter.

Kommentarer ignoreres.

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

3.3.1 Assemblerdirektiver

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

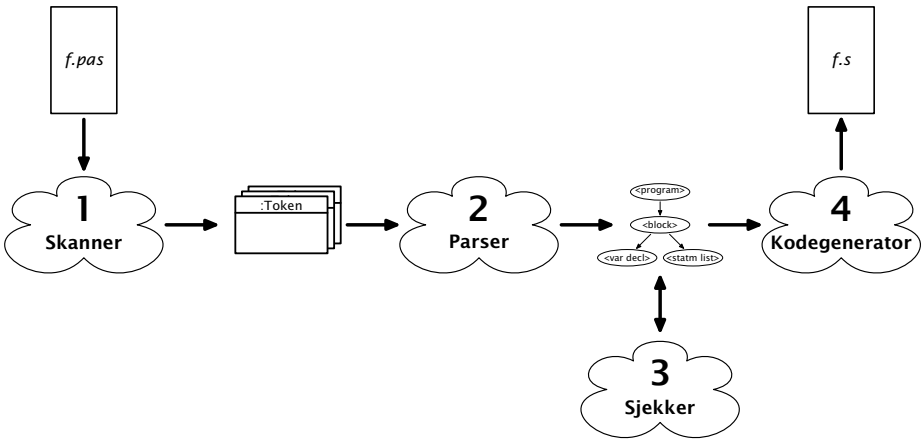
`.align` *n*	Angi at vi skal starte på en adresse delelig med 2^n.
`.asciz "`*xxxy zzz*`"`	Sett av plass til angitt tekst (med 0-byte som avslutning).
`.data`	Angi at vi nå skal plassere variabler i datadelen av minnet.
`.extern` *xxx*	Angi at navnet finnes i biblioteket som skal linkes inn senere.
`.globl` *xxx*	Angi at navnet skal være kjent utenfor filen.
`.text`	Angi at vi nå skal plassere instruksjoner i kodedelen av minnet.

Tabell 3.2: Assemblerdirektiver

Kapittel 4

Prosjektet

De aller fleste kompilatorer består av fire faser, som vist i figur 4.1. Hver av disse fire delene skal innleveres og godkjennes; se kursets nettside for frister.



Figur 4.1: Oversikt over prosjektet

4.1 Diverse informasjon om prosjektet

4.1.1 Basiskode

På emnets nettside ligger 2100-oblig.zip som er nyttig kode å starte med. Lag en egen mappe til prosjektet og legg ZIP-filen der. Gjør så dette:

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

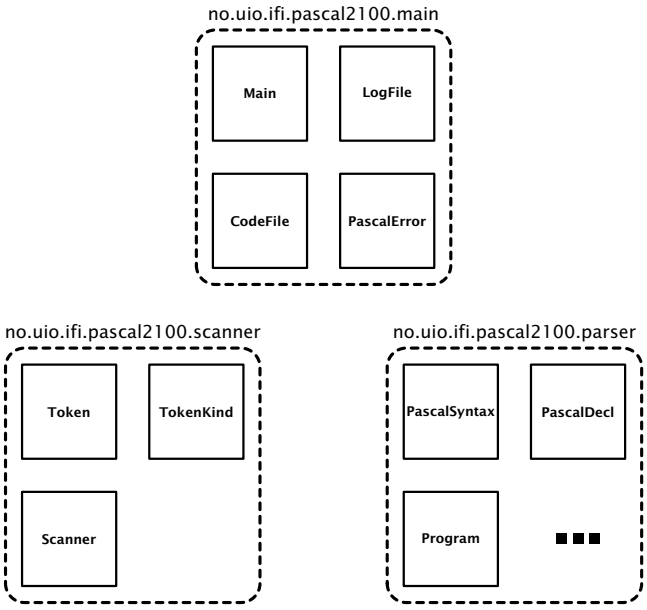
Dette vil resultere i en kjørbar fil `pascal2100.jar` som kan kjøres slik

```
$ java -jar pascal2100.jar minfil.pas
```

men den utleverte koden selvfølgelig ikke vil fungere! Denne er er bare en basis for å utvikle kompilatoren. Du kan fritt endre basiskoden, men den bør virke noenlunde likt.

4.1.2 Oppdeling i moduler

Alle større programmer bør deles opp i **moduler**, og i Java gjøres dette med package-mekanismen. Basiskoden er delt opp i tre moduler, som vist i figur 4.2.



Figur 4.2: De tre modulene i kompilatoren

`no.uio.ifi.pascal2100.main` innholder fire sentrale klasser som alle er ferdig programmert:

`Main` er «hovedprogrammet» som styrer hele kompileringen.

`LogFile` brukes til å opprette en loggfil (se avsnitt 4.1.4 på neste side).

`CodeFile` brukes til å skrive kodefilen som skal være resultatet av kompileringen.

`PascalError` benyttes til feilhåndteringen.

`no.uio.ifi.pascal2100.scanner` inneholder tre klasser som brukes av skanneren; se avsnitt 4.2 på side 40.

`no.uio.ifi.pascal2100.parser` inneholder (når prosjektet er ferdig) rundt 50 klasser som brukes til å bygge parseringstreet; se avsnitt 4.3 på side 43.

4.1.3 Selvidentifikasjon

Når man arbeider med objektorientert programmering, er det meget nyttig at alle objektene man lager, kan identifisere seg selv. På den måten er det enkelt å få nødvendig informasjon om objektene og lage greie status- og feilmeldinger.

I dette prosjektet skal vi la alle klassene ha en metode

```
public String identify() { ... }
```

som gir den informasjonen vi ønsker om objektet; se for eksempel på klassen `Token` i figur 4.4 på side 41.[1]

4.1.4 Logging

Som en hjelp under arbeidet, og for enkelt å sjekke om de ulike delene virker, skal koden kunne håndtere loggutskriftene vist i tabell 4.1.

Opsjon	Del	Hva logges
`-logB`	Del 3	Hvordan navnene bindes
`-logP`	Del 2	Hvilke parseringsmetoder som kalles
`-logS`	Del 1	Hvilke symboler som leses av skanneren
`-logT`	Del 3	Typesjekkingen
`-logY`	Del 2	Utskrift av parseringstreet

Tabell 4.1: Opsjoner for logging

4.1.5 Testprogrammer

Til hjelp under arbeidet finnes diverse testprogrammer:

- I mappen ~inf2100/oblig/test/ (som også er tilgjengelig fra en nettleser som http://inf2100.at.ifi.uio.no/oblig/test/) finnes noen Pascal2100-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 referansekompilatoren.

4.1.6 På egen datamaskin

Prosjektet er utviklet på Ifis 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 kompilere et system med mange Java-filer. Programmet kan hentes ned fra http://ant.apache.org/bindownload.cgi.

`gas` er assembleren. Den lastes gjerne ned sammen med C-kompilatoren `gcc`; se http://gcc.gnu.org/install/download.html.

`java` er en Java-interpreter (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`.

[1] Kan vi ikke bruke `toString`-metoden til dette? Svaret er nei, siden `toString` lager en tekst beregnet på *brukeren* av programmet, mens `identify` gir informasjon for *programmereren*.

javac er en Java-kompilator; du trenger *Java SE development kit* som kan hentes fra https://java.com/en/download/manual.jsp.

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

4.1.7 Tegnsett

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

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

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

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

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

4.2 Del 1: Skanneren

Skanneren leser programteksten fra en fil og deler den opp i **symboler** (på engelsk «tokens»), omtrent slik vi mennesker leser en tekst ord for ord.

mini.pas

```
1
2  /* Et minimalt Pascal-program */
3  program Mini;
4  begin
5     write('x');
6  end.
```

Figur 4.3: Et minimalt Pascal2100-program `mini.pas`

Programmet vist i figur 4.3 inneholder for eksempel disse symbolene:

`program` `Mini` `;` `begin` `write`

`(` `'x'` `)` `;` `end` `.`

Legg merke til at kommentarene er fjernet, og også all informasjon om blanke tegn og linjeskift; kun symbolene er tilbake.

NB! Det er viktig å huske at skanneren kun jobber med å finne symbolene i programkoden; den har ingen forståelse for hva som er et riktig eller fornuftig program. (Det kommer senere.)

Token.java

```
4
5   public class Token {
6       public TokenKind kind;
7       public String id, strVal;
8       public int intVal, lineNum;
            ⋮
70      public String identify() {
71          String t = kind.identify();
72          if (lineNum > 0)
73              t += " on line " + lineNum;
74
75          switch (kind) {
76          case nameToken:      t += ": " + id;  break;
77          case intValToken:    t += ": " + intVal;  break;
78          case stringValToken: t += ": '" + strVal + "'";  break;
79          }
80          return t;
81      }
82  }
```

Figur 4.4: Klassen Token

4.2.1 Representasjon av symboler

Hvert symbol i Pascal2100-programmet lagres i en instans av klassen Token vist i figur 4.4.

For hvert symbol må vi angi hva slags symbol det er, og dette angis med en TokenKind-referanse; se figur 4.5. Legg spesielt merke til eofToken («end-of-file-token»); det benyttes for å angi at det ikke er flere symboler igjen på filen.

TokenKind.java

```
5
6   public enum TokenKind {
7       nameToken("name"),
8       intValToken("number"),
9       stringValToken("text string"),
10
11      addToken("+"),
12      assignToken(":="),
            ⋮
69      eofToken("e-o-f");
70
71      private String image;
72
73      TokenKind(String im) {
74          image = im;
75      }
76
77
78      public String identify() {
79          return image + " token";
80      }
81
82      @Override public String toString() {
83          return image;
84      }
```

Figur 4.5: Klassen TokenKind

4.2.2 Skanneren

Selve skanneren er definert av klassen Scanner; se figur 4.6 på neste side. Legg merke til at den inneholder to symboler: curToken og nextToken,

nemlig det nåværende og det neste symbolet. Grunnen til det er at vi av og til ønsker å se litt forover etter hva som kommer senere i teksten.

Scanner.java

```
7
8   public class Scanner {
9       public Token curToken = null, nextToken = null;
10
11      private LineNumberReader sourceFile = null;
12      private String sourceFileName, sourceLine = "";
13      private int sourcePos = 0;
14
15      public Scanner(String fileName) {
16          sourceFileName = fileName;
17          try {
18              sourceFile = new LineNumberReader(new FileReader(fileName));
19          } catch (FileNotFoundException e) {
20              Main.error("Cannot read " + fileName + "!");
21          }
22
23          readNextToken();  readNextToken();
24      }
25
26
27      public String identify() {
28          return "Scanner reading " + sourceFileName;
29      }
            ⋮
242 }
```

Figur 4.6: Klassen `Scanner`

Den viktigste metoden i `Scanner` er `readNextToken` som leser neste symbol fra innfilen og lar `nextToken` peke på et nytt `Token`-objekt.

4.2.3 Logging

For å sjekke at skanningen fungerer rett, skal kompilatoren kunne kjøres med opsjonen `-testscanner`. Dette gir logging at to ting til loggfilen:

1) Hver gang `readNextToken` leser inn en ny linje, skal denne linjen logges.

2) Hovedprogrammet skal kalle gjentatte ganger på `readNextToken` og for hver gang skrive ut hvilket symbol som ble lest; kallet `curToken.identify()` brukes for å få symbolet på en passende form.

(Sjekk kildekoden til `Main.java` for å se at dette stemmer.)

For å demonstrere hva som ønskes av testutskrift, har jeg laget både et minimalt og litt større Pascal-program; se figur 4.7 på neste side og figur 4.14 på side 55. Når kompilatoren vår kjøres med opsjonen `-testscanner`, skriver de ut logginformasjonen vist i henholdsvis figur 4.7 på neste side og figur 4.15 til 4.16 på side 55–56.

mini.pas

```
1
2  /* Et minimalt Pascal-program */
3  program Mini;
4  begin
5     write('x');
6  end.
```

```
1     1:
2     2: /* Et minimalt Pascal-program */
3     3: program Mini;
4  Scanner: program token on line 3
5  Scanner: name token on line 3: Mini
6  Scanner: ; token on line 3
7     4: begin
8  Scanner: begin token on line 4
9     5:    write('x');
10 Scanner: name token on line 5: write
11 Scanner: ( token on line 5
12 Scanner: text string token on line 5: 'x'
13 Scanner: ) token on line 5
14 Scanner: ; token on line 5
15    6: end.
16 Scanner: end token on line 6
17 Scanner: . token on line 6
18 Scanner: e-o-f token
```

Figur 4.7: Loggfil med de symboler skanneren finner i `mini.pas`

4.2.4 Mål for del 1

Mål for del 1

Programmet skal utvikles slik at opsjonen `-testscanner` produserer loggfiler som vist i figurene 4.7 og 4.15–4.16.

4.3 Del 2: Parsering

Denne delen går ut på å skrive parseren som har to oppgaver:

- sjekke at programmet er korrekt i henhold til språkdefinisjonen (dvs grammatikken, ofte kalt syntaksen) og
- lage et tre som representerer programmet.

Testprogrammet `mini.pas` skal for eksempel gi treet vist i figur 4.8 på neste side.

4.3.1 Implementasjon

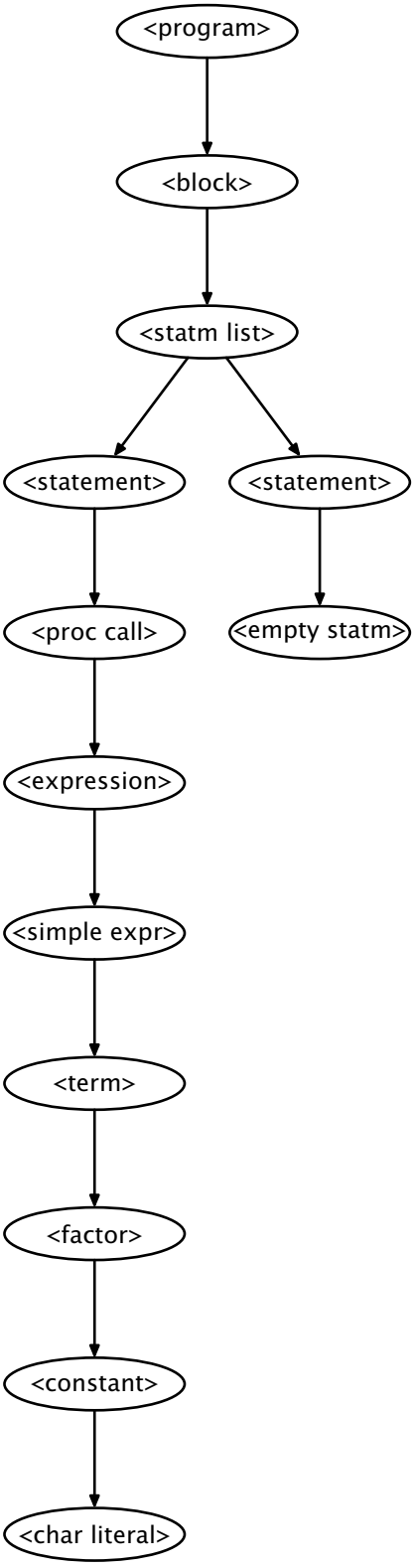
Aller først må det defineres en klasse per ikke-terminal («firkantene» i grammatikken), og alle disse må være subklasser av `PascalSyntax`. (Alle ikke-terminaler som representerer en deklarasjon, bør være subklasse av `PascalDecl`, men dette kan ordnes under del 3.) Klassene må inneholde tilstrekkelige deklarasjoner til å kunne representere ikke-terminalen. Som et eksempel er vist klassen `WhileStatm` som representerer ⟨while-statm⟩; se figur 4.9 på side 45.

Et par ting verdt å merke seg:

- Ikke-terminalene ⟨letter a-z⟩, ⟨digit 0-9⟩ og ⟨char except '⟩ er allerede tatt hånd om av skanneren, så de kan vi se bort fra nå.

mini.pas

```
1
2  /* Et minimalt Pascal-program */
3  program Mini;
4  begin
5     write('x');
6  end.
```



Figur 4.8: Syntakstreet laget utifra testprogrammet `mini.pas`

WhileStatm.java

```
8
9   class WhileStatm extends Statement {
10      Expression expr;
11      Statement body;
12
13      WhileStatm(int lNum) {
14          super(lNum);
15      }
16
17      @Override public String identify() {
18          return "<while-statm> on line " + lineNum;
19      }
            ⋮
41
42      @Override void prettyPrint() {
43          Main.log.prettyPrint("while ");  expr.prettyPrint();
44          Main.log.prettyPrintLn(" do");  body.prettyPrint();
45      }
46
47      static WhileStatm parse(Scanner s) {
48          enterParser("while-statm");
49
50          WhileStatm ws = new WhileStatm(s.curLineNum());
51          s.skip(whileToken);
52
53          ws.expr = Expression.parse(s);
54          s.skip(doToken);
55          ws.body = Statement.parse(s);
56
57          leaveParser("while-statm");
58          return ws;
59      }
60  }
```

Figur 4.9: Klassen `WhileStatm`

- ⟨name⟩ trenger ikke en egen klasse; en `String` er nok.
- Ikke-terminaler som kun er definert som et valgt mellom ulike andre ikke-terminaler (som f eks ⟨constant⟩ og ⟨type⟩) bør implementeres som en abstrakt klasse, og så bør alternativene være sub-klasser av denne abstrakte klassen.

4.3.2 Parsering

Den enkleste måte å parsere et Pascal-program på er å benytte såkalt «recursive descent» og legge inn en metode

```
1
2     static Xxxx parse(Scanner s) {
3       ...
4     }
```

i alle sub-klassene av `PascalSyntax`. Den skal parsere «seg selv» og lagre dette i et objekt; se for eksempel `WhileStatm.parse` i figur 4.9. (Metodene `test` og `skip` er nyttige i denne sammenhengen; de er definert i `Scanner`-klassen.)

4.3.3 Syntaksfeil

Ved å benytte denne parseringsmetoden er det enkelt å finne grammatikkfeil: Når det ikke finnes noe lovlig alternativ i jernbanediagrammene, har vi

```
   1:
   2: /* Et minimalt Pascal-program */
   3: program Mini;
Parser:   <program>
   4: begin
   5:    write('x');
Parser:     <block>
Parser:       <statm list>
Parser:         <statement>
Parser:           <proc call>
Parser:             <expression>
Parser:               <simple expr>
Parser:                 <term>
Parser:                   <factor>
Parser:                     <constant>
Parser:                       <char literal>
Parser:                       </char literal>
Parser:                     </constant>
Parser:                   </factor>
Parser:                 </term>
Parser:               </simple expr>
Parser:             </expression>
   6: end.
Parser:           </proc call>
Parser:         </statement>
Parser:         <statement>
Parser:           <empty statm>
Parser:           </empty statm>
Parser:         </statement>
Parser:       </statm list>
Parser:     </block>
Parser:   </program>
```

Figur 4.10: Loggfil som viser parsering av `mini.pas`

en feilsituasjon, og vi må kalle `PascalSyntax.error`. (Metodene `test` og `skip` gjør dette automatisk for oss.)

4.3.4 Logging

For å sjekke at parseringen går greit (og enda mer for å finne ut hvor langt vi er kommet om noe går galt), skal `parse`-metodene kalle på `PascalSyntax.enterParser` når de starter og `PascalSyntax.leaveParser` når de avslutter. Dette vil gi en oversiktlig opplisting av hvordan parsering forløper.

Våre to vanlige testprogram vist i henholdsvis figur 4.3 på side 40 og figur 4.14 på side 55, vil produsere loggfilene i henholdsvis figur 4.10 og figurene 4.17 til 4.20 på side 57 og etterfølgende når kompilatoren kjøres med opsjonen `-logP` eller `-testparser`.

Dette er imidlertid ikke nok. Selv om parsering forløp feilfritt, kan det hende at parseringstreet ikke er riktig bygget opp. Den enkleste måten å sjekke dette på er å skrive ut det opprinnelige programmet basert på representasjonen i syntakstreet.[2] Dette ordnes best ved å legge inn en metode

```

  void prettyPrint() { ... }
```

i hver subklasse av `PascalSyntax`.

[2] En slik automatisk utskrift av et program kalles gjerne «pretty-printing» siden resultatet gjerne blir penere enn en travel programmerer produserer. Denne finessen var mye vanligere i tiden før man fikk interaktive datamaskiner og gode redigeringsprogrammer.

```
1  program Mini;
2  begin
3    write('x');
4
5  end.
```

Figur 4.11: Loggfil med «skjønnskrift» av `mini.pas`

> ***Mål for del 2***
>
> *Programmet skal implementere parsering og også utskrift av det lagrede programmet; med andre ord skal opsjonen* `-testparser` *gi utskrift som vist i figurene 4.10–4.11 og 4.17–4.21.*

4.4 Del 3: Sjekking

Den tredje delen er å få sjekkingen og kodegenereringen på plass.

Del 3 skal sjekke fire ting, og dette gjøres ved å traversere hele syntakstreet med metoden `check`; noen av testene gjøres på vei nedover i treet og noen på vei tilbake.

4.4.1 Sjekke navn ved deklarasjoner

Det må sjekkes at navn er deklarert riktig. I Pascal2100 er dette enkelt, for det er bare én mulig feil: å deklarere navn flere ganger i samme blokk.

4.4.2 Sjekke deklarasjoner

Dette innebærer å se på alle navneforekomster og så finne hvilke deklarasjoner som definerer navnet.

```
1  Binding on line 5: write was declared in <proc decl> in the library
```

Figur 4.12: Loggfil med navnebinding for `mini.pas`

4.4.3 Sjekke navnebruk

Så må det sjekkes om navnene er brukt riktig, for eksempel sjekke om brukeren har benyttet et variabelnavn for å kalle en funksjon eller et funksjonsnavn for å finne et arrayelement. Dette gjøres ved å definere og kalle på `checkWhetherAssignable` og tilsvarende.

4.4.4 Bestemme typer

For alle uttrykk og deluttrykk må vi finne hvilken type de har; dette settes inn i `Expression.type`, `Term.type` og alle andre klasser for deluttrykk. (Alle deklarasjoner har fått satt sin `PascalDecl.type` i del 2.) Ved å angi `-logT` kan brukeren få logget alle typesjekkene som foretas; se eksempel i figur 4.23 på side 61. (Vårt minimale testprogram `mini.pas` fra figur 4.3 på side 40 er så enkelt at det ikke produserer noen typesjekklogg.)

Mål for del 3

Kompilatoren skal foreta navnebindinger og kunne produsere data om dette som vist i figur 4.12 og 4.22. Sjekk av navnebruk og typefeil er ikke nødvendig for å få godkjent prosjektet.

4.5 Del 4: Kodegenerering

4.5.1 Konvensjoner

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

4.5.2 Registre

Vi vil bruke disse registrene:

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

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

%EDX brukes til arrayadresser og som hjelperegister ved tilordning og divisjon.

%ESP peker på toppen av kjørestakken.

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

4.5.2.1 Navn

Hovedprogrammet, funksjoner og prosedyrer beholder sitt Pascal2100-navn men med en endelse så vi unngår dobbeltdeklarasjoner: `proc$`*`name_nnn`*.

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

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 eller prosedyre.

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

4.5.3 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.

```
1  # Code file created by Pascal2100 compiler 2015-08-18 15:19:47
2          .extern write_char
3          .extern write_int
4          .extern write_string
5          .globl  _main
6          .globl  main
7  _main:
8  main:   call    prog$Mini_1             # Start program
9          movl    $0,%eax                 # Set status 0 and
10         ret                             # terminate the program
11 prog$Mini_1:
12         enter   $32,$1                  # Start of Mini
13         movl    $120,%eax               #   char 120
14         pushl   %eax                    # Push param #1.
15         call    write_char
16         addl    $4,%esp                 # Pop parameter.
17         leave                           # End of Mini
18         ret
```

Figur 4.13: Kodefil laget fra `mini.pas`

4.5.3.1 Operander i uttrykk

I tabell 4.2 er vist hvilken kode som må genereres for å hente en verdi ⟨n⟩, en enkel variabel $\langle v^{(b)o} \rangle$ (der b er blokknivået og o er variabelens «offset»), et arrayelement $\langle a^{(b)o} \rangle$`[`⟨e⟩`]` eller et uttrykk i parenteser (⟨e⟩) inn i register %EAX.

Legg merke til at når vi slår opp i en array, må vi trekke fra nedre indeksgrense; mao, hvis hvis arrayen er deklarert som `array[10..20] of ...`, må vi trekke fra 10 ved hvert oppslag.

Enum-verdier blir representert av heltall, dvs første verdi er 0, andre verdi er 1 osv. Spesielt er det viktig at

`false` = 0, `true` = 1

(Kode for funksjonskall er ikke tatt med her – dette er beskrevet i avsnitt 4.8 på side 52.)

⟨n⟩	⇒	`movl` `$`⟨n⟩`,%eax`
$\langle v^{(b)o} \rangle$	⇒	`movl` $-4b$`(%ebp),%edx` `movl` $-o$`(%edx),%eax`

$\langle a^{(b)o} \rangle$`[`⟨e⟩`]`	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩
		`subl` `$`*low*`,%eax`
		`movl` $-4b$`(%ebp),%edx`
		`leal` $-o$`(%edx),%edx`
		`movl` `(%edx,%eax,4),%eax`

(⟨e⟩)	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩

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

4.5.3.2 Operatorer i uttrykk

Her følges også konvensjonen om at alle verdier skal lages i %EAX.

Unære operatorer Tabell 4.3 viser hvordan vi skal oversette de unære operatorene.

+ ⟨e⟩	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩
- ⟨e⟩	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩ `negl %eax`
`not` ⟨e⟩	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩ `xorl $1,%eax`

Tabell 4.3: Kode generert av unære operatorer i uttrykk

Binære operatorer I tabell 4.4 er vist hvordan de binære operatorene +, `div` (som trenger litt annen kode enn de andre regneoperatorene) og == skal oversettes. De øvrige finner du sikkert selv.

⟨e_1⟩ + ⟨e_2⟩	⇒	⟨Beregn ⟨e_1⟩ med svar i %EAX⟩
		`pushl %eax`
		⟨Beregn ⟨e_2⟩ med svar i %EAX⟩
		`movl %eax,%ecx`
		`popl %eax`
		`addl %ecx,%eax`

⟨e_1⟩ `div` ⟨e_2⟩	⇒	⟨Beregn ⟨e_1⟩ med svar i %EAX⟩
		`pushl %eax`
		⟨Beregn ⟨e_2⟩ med svar i %EAX⟩
		`movl %eax,%ecx`
		`popl %eax`
		`cdq`
		`idivl %ecx`

⟨e_1⟩ == ⟨e_2⟩	⇒	⟨Beregn ⟨e_1⟩ med svar i %EAX⟩
		`pushl %eax`
		⟨Beregn ⟨e_2⟩ med svar i %EAX⟩
		`popl %ecx`
		`cmpl %eax,%ecx`
		`movl $0,%eax`
		`sete %al`

Tabell 4.4: Kode generert av binære operatorer i uttrykk

4.5.4 Oversettelse av setninger

4.5.4.1 Oversettelse av tomme setninger

Dette er den enkleste setningen å oversette, som vist i tabell 4.5.

`;`	⇒	

Tabell 4.5: Kode generert av tom setning

4.5.4.2 Oversettelse av sammensatte setninger

En sammensatt setning er ganske så enkel å oversette; se tabell 4.6.

`begin` ⟨S_1⟩`;` ⟨S_2⟩`;` `... end`	⇒	⟨S_1⟩ ⟨S_2⟩ ⋮

Tabell 4.6: Kode generert av sammensatt setning

4.5.4.3 Oversettelse av tilordningssetninger

Kodegenerering for slike setninger er vist i tabell 4.7. Husk at venstresiden kan være enten en vanlig variabel ⟨$v^{(b)o}$⟩, et arrayelement ⟨$a^{(b)o}$⟩`[`⟨e⟩`]` eller et funksjonsnavn ⟨f⟩.

⟨$v^{(b)o}$⟩ `:=` ⟨e⟩`;`	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩
		`movl` $-4b$`(%ebp),%edx`
		`movl` `%eax,`$-o$`(%edx)`

⟨$a^{(b)o}$⟩`[`⟨e_1⟩`]` `:=` ⟨e_2⟩	⇒	⟨Beregn ⟨e_2⟩ med svar i %EAX⟩
		`pushl %eax`
		⟨Beregn ⟨e_1⟩ med svar i %EAX⟩
		`subl $`low`,%eax`
		`movl` $-4b$`(%ebp),%edx`
		`leal` $-o$`(%edx),%edx`
		`popl %ecx`
		`movl %ecx,(%edx,%eax,4)`

⟨f⟩ `:=` ⟨e⟩`;`	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩ `movl %eax,-32(%ebp)`

Tabell 4.7: Kode generert av tilordning

4.5.4.4 Oversettelse av kallsetninger

Kallsetninger og funksjonskall oversettes på akkurat samme måte, nemlig til de tre kodesekvensene vist i tabell 4.8 på neste side.

1) Parametrene legges på stakken (i *omvendt rekkefølge*).

2) Funksjonen kalles.

3) Parametrene fjernes fra stakken.

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

f(⟨e_1⟩, ⟨e_2⟩)	⇒	⟨Beregn ⟨e_2⟩ med svar i %EAX⟩
		`pushl %eax`
		⟨Beregn ⟨e_1⟩ med svar i %EAX⟩
		`pushl %eax`
		
		`call proc$`*f_n*
		
		`addl $8,%esp`

Tabell 4.8: Kode generert av funksjonskall

Kall på `write` Prosedyren `write` er som nevnt spesiell: den kan ha vilkårlig mange parametre, og de kan være av enhver type (unntatt arrayer av annet enn `char`). Hver parameter oversettes til et kall på en egen biblioteksfunksjon.

`write(`⟨int-e⟩`)`	⇒	⟨Beregn ⟨int-e⟩ med svar i %EAX⟩
		`pushl %eax`
		`call write_int`
		`addl $4,%esp`

`write(`⟨char-e⟩`)`	⇒	⟨Beregn ⟨char-e⟩ med svar i %EAX⟩
		`pushl %eax`
		`call write_char`
		`addl $4,%esp`

`write(`⟨string-e⟩`)`	⇒	`.data`
		⟨lab⟩`: .asciz "`⟨string-e⟩`"`
		`.align 2`
		`.text`
		`leal` ⟨lab⟩`,%eax`
		`pushl %eax`
		`call write_string`
		`addl $4,%esp`

Tabell 4.9: Kode generert av kall på `write`

4.5.4.5 Oversettelse av if-setninger

Tabell 4.10 på neste side viser oversettelse av en if-setning, både uten og med en else-gren.

`if` ⟨e⟩ `then` ⟨S⟩	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩ `cmpl $0,%eax` `je` ⟨lab⟩ ⟨S⟩ ⟨lab⟩:
`if` ⟨e⟩ `then` ⟨S_1⟩ `else` ⟨S_2⟩	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩ `cmpl $0,%eax` `je` ⟨lab_1⟩ ⟨S_1⟩ `jmp` ⟨lab_2⟩ ⟨lab_1⟩: ⟨S_2⟩ ⟨lab_2⟩:

Tabell 4.10: Kode generert av if-setning

4.5.4.6 Oversettelse av while-setninger

Oversettelse av en while-setning innebærer å lage en løkke og en løkketest; dette er vist i tabell 4.11.

`while` ⟨e⟩ `do` ⟨S⟩	⇒	⟨lab_1⟩: ⟨Beregn ⟨e⟩ med svar i %EAX⟩ `cmpl $0,%eax` `je` ⟨lab_2⟩ ⟨S⟩ `jmp` ⟨lab_1⟩ ⟨lab_2⟩:

Tabell 4.11: Kode generert av while-setning

4.5.5 Oversettelse av funksjoner og prosedyrer

Som vist i figur 4.12 på neste side legger vi inn litt fast kode i begynnelsen og slutten av funksjonen. Legg også merke til at:

- 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. Til denne summen skal vi addere **32** for systeminformasjon.

  Vi må også hvilket **blokknivå** funksjonen/prosedyren har.
- Vi må bruke `leave`-instruksjonen til å frigjøre plassen vi satte av til lokale variabler før vi hopper tilbake med en `ret`.

`function ⟨f⟩ (...): ⟨type⟩;`
`⟨D⟩`
`begin`
`  ⟨S⟩`
`end`
	⇒	`func$⟨f⟩_n: enter $⟨32+ant byte i ⟨D⟩⟩,$⟨blokknivå⟩`
		`  ⟨S⟩`
		`  movl  -32(%ebp),%eax`
		`  leave`
		`  ret`

`procedure ⟨p⟩ (...);`
`⟨D⟩`
`begin`
`  ⟨S⟩`
`end`
	⇒	`proc$⟨p⟩_n: enter $⟨32+ant byte i ⟨D⟩⟩,$⟨blokknivå⟩`
		`  ⟨S⟩`
		`  leave`
		`  ret`

Tabell 4.12: Kode generert av funksjonsdeklarasjon

4.5.5.1 Oversettelse av hovedprogrammet

Det enkleste er å late som om hoveprogrammet er en prosedyre som kalles av en minimal `main`;[3] se tabell 4.13.

`program xx;`
`⟨D⟩`
`begin`
`  ⟨S⟩`
`end.`
	⇒	`  .extern write_char`
		`  .extern write_int`
		`  .extern write_string`
		`  .globl  _main`
		`  .globl  main`
		`_main:`
		`main: call  prog$xx_n`
		`  movl  $0,%eax`
		`  ret`
		`prog$xx_n: enter $⟨32+ant byte i ⟨D⟩⟩,$1`
		`  ⟨S⟩`
		`  leave`
		`  ret`

Tabell 4.13: Kode generert av hovedprogrammet

4.5.6 Deklarasjon av variabler

4.5.6.1 Deklarasjon av lokale variabler

Programmet/prosedyren/funksjonen sørger selv for å sette av plass til sine lokale variabler på stakken (se tabell 4.12).

4.5.6.2 Deklarasjon av parametre

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

> ***Mål for del 4***
>
> *Kompilatoren skal generere kode som lar seg assemblere på Ifis Linux-maskiner og som utfører det kompilerte programmet korrekt.*

[3] Det er et krav at hovedprogrammet heter `main` i Unix og `_main` i Windows og Mac OS X.

gcd.pas

```
1  program GCD;
2  /* A program to compute the greatest common of two numbers,
3     i.e., the biggest numbers by which the two original
4     numbers can be divide without a remainder. */
5
6  const v1 = 1071; v2 = 462;
7
8  var res: integer;
9
10 function GCD (m: integer; n: integer): integer;
11 begin
12    if n = 0 then
13       GCD := m
14    else
15       GCD := GCD(n, m mod n)
16 end; { GCD }
17
18 begin
19    res := GCD(v1,v2);
20    write('GCD(', v1, ',', v2, ') = ', res, eol);
21 end.
```

Figur 4.14: Et litt større Pascal2100-program `gcd.pas`

```
1     1: program GCD;
2  Scanner: program token on line 1
3  Scanner: name token on line 1: GCD
4  Scanner: ; token on line 1
5     2: /* A program to compute the greatest common of two numbers,
6     3:    i.e., the biggest numbers by which the two original
7     4:    numbers can be divide without a remainder. */
8     5:
9     6: const v1 = 1071; v2 = 462;
10 Scanner: const token on line 6
11 Scanner: name token on line 6: v1
12 Scanner: = token on line 6
13 Scanner: number token on line 6: 1071
14 Scanner: ; token on line 6
15 Scanner: name token on line 6: v2
16 Scanner: = token on line 6
17 Scanner: number token on line 6: 462
18 Scanner: ; token on line 6
19    7:
20    8: var res: integer;
21 Scanner: var token on line 8
22 Scanner: name token on line 8: res
23 Scanner: : token on line 8
24 Scanner: name token on line 8: integer
25 Scanner: ; token on line 8
26    9:
27   10: function GCD (m: integer; n: integer): integer;
28 Scanner: function token on line 10
29 Scanner: name token on line 10: GCD
30 Scanner: ( token on line 10
31 Scanner: name token on line 10: m
32 Scanner: : token on line 10
33 Scanner: name token on line 10: integer
34 Scanner: ; token on line 10
35 Scanner: name token on line 10: n
36 Scanner: : token on line 10
37 Scanner: name token on line 10: integer
38 Scanner: ) token on line 10
39 Scanner: : token on line 10
40 Scanner: name token on line 10: integer
41 Scanner: ; token on line 10
42   11: begin
43 Scanner: begin token on line 11
44   12:    if n = 0 then
45 Scanner: if token on line 12
46 Scanner: name token on line 12: n
47 Scanner: = token on line 12
48 Scanner: number token on line 12: 0
49 Scanner: then token on line 12
50   13:       GCD := m
```

Figur 4.15: Loggfil som demonstrerer hvilke symboler skanneren finner i `gcd.pas` (del 1)

```
Scanner: name token on line 13: GCD
Scanner: := token on line 13
Scanner: name token on line 13: m
  14:    else
Scanner: else token on line 14
  15:       GCD := GCD(n, m mod n)
Scanner: name token on line 15: GCD
Scanner: := token on line 15
Scanner: name token on line 15: GCD
Scanner: ( token on line 15
Scanner: name token on line 15: n
Scanner: , token on line 15
Scanner: name token on line 15: m
Scanner: mod token on line 15
Scanner: name token on line 15: n
Scanner: ) token on line 15
  16: end; { GCD }
Scanner: end token on line 16
Scanner: ; token on line 16
  17:
  18: begin
Scanner: begin token on line 18
  19:    res := GCD(v1,v2);
Scanner: name token on line 19: res
Scanner: := token on line 19
Scanner: name token on line 19: GCD
Scanner: ( token on line 19
Scanner: name token on line 19: v1
Scanner: , token on line 19
Scanner: name token on line 19: v2
Scanner: ) token on line 19
Scanner: ; token on line 19
  20:    write('GCD(', v1, ',', v2, ') = ', res, eol);
Scanner: name token on line 20: write
Scanner: ( token on line 20
Scanner: text string token on line 20: 'GCD('
Scanner: , token on line 20
Scanner: name token on line 20: v1
Scanner: , token on line 20
Scanner: text string token on line 20: ','
Scanner: , token on line 20
Scanner: name token on line 20: v2
Scanner: , token on line 20
Scanner: text string token on line 20: ') = '
Scanner: , token on line 20
Scanner: name token on line 20: res
Scanner: , token on line 20
Scanner: name token on line 20: eol
Scanner: ) token on line 20
Scanner: ; token on line 20
  21: end.
Scanner: end token on line 21
Scanner: . token on line 21
Scanner: e-o-f token
```

Figur 4.16: Loggfil som demonstrerer hvilke symboler skanneren finner i `gcd.pas` (del 2)

```
1     1: program GCD;
2  Parser:   <program>
3     2: /* A program to compute the greatest common of two numbers,
4     3:    i.e., the biggest numbers by which the two original
5     4:    numbers can be divide without a remainder. */
6     5:
7     6: const v1 = 1071; v2 = 462;
8  Parser:     <block>
9  Parser:       <const decl part>
10 Parser:         <const decl>
11 Parser:           <constant>
12 Parser:             <number literal>
13 Parser:             </number literal>
14 Parser:           </constant>
15 Parser:         </const decl>
16 Parser:         <const decl>
17 Parser:           <constant>
18 Parser:             <number literal>
19    7:
20    8: var res: integer;
21 Parser:             </number literal>
22 Parser:           </constant>
23 Parser:         </const decl>
24 Parser:       </const decl part>
25 Parser:       <var decl part>
26 Parser:         <var decl>
27 Parser:           <type>
28 Parser:             <type name>
29    9:
30   10: function GCD (m: integer; n: integer): integer;
31 Parser:             </type name>
32 Parser:           </type>
33 Parser:         </var decl>
34 Parser:       </var decl part>
35 Parser:       <func decl>
36 Parser:         <param decl list>
37 Parser:           <param decl>
38 Parser:             <type name>
39 Parser:             </type name>
40 Parser:           </param decl>
41 Parser:           <param decl>
42 Parser:             <type name>
43 Parser:             </type name>
44 Parser:           </param decl>
45 Parser:         </param decl list>
46 Parser:         <type name>
47   11: begin
48 Parser:         </type name>
49   12:    if n = 0 then
50 Parser:         <block>
51 Parser:           <statm list>
52 Parser:             <statement>
53 Parser:               <if-statm>
54 Parser:                 <expression>
55 Parser:                   <simple expr>
56 Parser:                     <term>
57 Parser:                       <factor>
58 Parser:                         <variable>
59 Parser:                         </variable>
60 Parser:                       </factor>
61 Parser:                     </term>
62 Parser:                   </simple expr>
63 Parser:                   <rel opr>
64 Parser:                   </rel opr>
65 Parser:                   <simple expr>
66 Parser:                     <term>
67 Parser:                       <factor>
68 Parser:                         <constant>
69 Parser:                           <number literal>
70   13:       GCD := m
71 Parser:                           </number literal>
72 Parser:                         </constant>
73 Parser:                       </factor>
74 Parser:                     </term>
75 Parser:                   </simple expr>
```

Figur 4.17: Loggfil som viser parsering av `gcd.pas` (del 1)

```
76  Parser:                 </expression>
77  Parser:                 <statement>
78  Parser:                   <assign statm>
79  Parser:                     <variable>
80  Parser:                     </variable>
81    14:    else
82  Parser:                     <expression>
83  Parser:                       <simple expr>
84  Parser:                         <term>
85  Parser:                           <factor>
86  Parser:                             <variable>
87    15:       GCD := GCD(n, m mod n)
88  Parser:                             </variable>
89  Parser:                           </factor>
90  Parser:                         </term>
91  Parser:                       </simple expr>
92  Parser:                     </expression>
93  Parser:                   </assign statm>
94  Parser:                 </statement>
95  Parser:                 <statement>
96  Parser:                   <assign statm>
97  Parser:                     <variable>
98  Parser:                     </variable>
99  Parser:                     <expression>
100 Parser:                       <simple expr>
101 Parser:                         <term>
102 Parser:                           <factor>
103 Parser:                             <func call>
104 Parser:                               <expression>
105 Parser:                                 <simple expr>
106 Parser:                                   <term>
107 Parser:                                     <factor>
108 Parser:                                       <variable>
109 Parser:                                       </variable>
110 Parser:                                     </factor>
111 Parser:                                   </term>
112 Parser:                                 </simple expr>
113 Parser:                               </expression>
114 Parser:                               <expression>
115 Parser:                                 <simple expr>
116 Parser:                                   <term>
117 Parser:                                     <factor>
118 Parser:                                       <variable>
119 Parser:                                       </variable>
120 Parser:                                     </factor>
121 Parser:                                     <factor opr>
122 Parser:                                     </factor opr>
123 Parser:                                     <factor>
124 Parser:                                       <variable>
125   16: end; { GCD }
126 Parser:                                       </variable>
127 Parser:                                     </factor>
128 Parser:                                   </term>
129 Parser:                                 </simple expr>
130 Parser:                               </expression>
131 Parser:                             </func call>
132 Parser:                           </factor>
133 Parser:                         </term>
134 Parser:                       </simple expr>
135 Parser:                     </expression>
136 Parser:                   </assign statm>
137 Parser:                 </statement>
138 Parser:               </if-statm>
139 Parser:             </statement>
140 Parser:           </statm list>
141   17:
142   18: begin
143 Parser:         </block>
144   19:    res := GCD(v1,v2);
145 Parser:       </func decl>
146 Parser:       <statm list>
147 Parser:         <statement>
148 Parser:           <assign statm>
149 Parser:             <variable>
150 Parser:             </variable>
```

Figur 4.18: Loggfil som viser parsering av `gcd.pas` (del 2)

```
151 Parser:              <expression>
152 Parser:                <simple expr>
153 Parser:                  <term>
154 Parser:                    <factor>
155 Parser:                      <func call>
156 Parser:                        <expression>
157 Parser:                          <simple expr>
158 Parser:                            <term>
159 Parser:                              <factor>
160 Parser:                                <variable>
161 Parser:                                </variable>
162 Parser:                              </factor>
163 Parser:                            </term>
164 Parser:                          </simple expr>
165 Parser:                        </expression>
166 Parser:                        <expression>
167 Parser:                          <simple expr>
168 Parser:                            <term>
169 Parser:                              <factor>
170 Parser:                                <variable>
171 Parser:                                </variable>
172 Parser:                              </factor>
173 Parser:                            </term>
174 Parser:                          </simple expr>
175 Parser:                        </expression>
176   20:    write('GCD(', v1, ',', v2, ') = ', res, eol);
177 Parser:                      </func call>
178 Parser:                    </factor>
179 Parser:                  </term>
180 Parser:                </simple expr>
181 Parser:              </expression>
182 Parser:            </assign statm>
183 Parser:          </statement>
184 Parser:          <statement>
185 Parser:            <proc call>
186 Parser:              <expression>
187 Parser:                <simple expr>
188 Parser:                  <term>
189 Parser:                    <factor>
190 Parser:                      <constant>
191 Parser:                        <string literal>
192 Parser:                        </string literal>
193 Parser:                      </constant>
194 Parser:                    </factor>
195 Parser:                  </term>
196 Parser:                </simple expr>
197 Parser:              </expression>
198 Parser:              <expression>
199 Parser:                <simple expr>
200 Parser:                  <term>
201 Parser:                    <factor>
202 Parser:                      <variable>
203 Parser:                      </variable>
204 Parser:                    </factor>
205 Parser:                  </term>
206 Parser:                </simple expr>
207 Parser:              </expression>
208 Parser:              <expression>
209 Parser:                <simple expr>
210 Parser:                  <term>
211 Parser:                    <factor>
212 Parser:                      <constant>
213 Parser:                        <char literal>
214 Parser:                        </char literal>
215 Parser:                      </constant>
216 Parser:                    </factor>
217 Parser:                  </term>
218 Parser:                </simple expr>
219 Parser:              </expression>
220 Parser:              <expression>
221 Parser:                <simple expr>
222 Parser:                  <term>
223 Parser:                    <factor>
224 Parser:                      <variable>
225 Parser:                      </variable>
```

Figur 4.19: Loggfil som viser parsering av `gcd.pas` (del 3)

```
226 Parser:                    </factor>
227 Parser:                  </term>
228 Parser:                </simple expr>
229 Parser:              </expression>
230 Parser:              <expression>
231 Parser:                <simple expr>
232 Parser:                  <term>
233 Parser:                    <factor>
234 Parser:                      <constant>
235 Parser:                        <string literal>
236 Parser:                        </string literal>
237 Parser:                      </constant>
238 Parser:                    </factor>
239 Parser:                  </term>
240 Parser:                </simple expr>
241 Parser:              </expression>
242 Parser:              <expression>
243 Parser:                <simple expr>
244 Parser:                  <term>
245 Parser:                    <factor>
246 Parser:                      <variable>
247 Parser:                      </variable>
248 Parser:                    </factor>
249 Parser:                  </term>
250 Parser:                </simple expr>
251 Parser:              </expression>
252 Parser:              <expression>
253 Parser:                <simple expr>
254 Parser:                  <term>
255 Parser:                    <factor>
256 Parser:                      <variable>
257 Parser:                      </variable>
258 Parser:                    </factor>
259 Parser:                  </term>
260 Parser:                </simple expr>
261 Parser:              </expression>
262   21: end.
263 Parser:            </proc call>
264 Parser:          </statement>
265 Parser:          <statement>
266 Parser:            <empty statm>
267 Parser:            </empty statm>
268 Parser:          </statement>
269 Parser:        </statm list>
270 Parser:      </block>
271 Parser:    </program>
```

Figur 4.20: Loggfil som viser parsering av `gcd.pas` (del 4)

```
1  program GCD;
2  const
3    v1 = 1071;
4    v2 = 462;
5  var
6    res: integer;
7
8  function GCD (m: integer; n: integer): integer;
9  begin
10   if n = 0 then
11     GCD := m
12   else
13     GCD := GCD(n, m mod n)
14 end; {GCD}
15
16 begin
17   res := GCD(v1, v2);
18   write('GCD(', v1, ',', v2, ') = ', res, eol);
19
20 end.
```

Figur 4.21: Loggfil med «skjønnskrift» av `gcd.pas`

```
1  Binding on line 8: integer was declared in <type decl> in the library
2  Binding on line 10: integer was declared in <type decl> in the library
3  Binding on line 10: integer was declared in <type decl> in the library
4  Binding on line 10: integer was declared in <type decl> in the library
5  Binding on line 12: n was declared in <param decl> on line 10
6  Binding on line 13: GCD was declared in <func decl> on line 10
7  Binding on line 13: m was declared in <param decl> on line 10
8  Binding on line 15: GCD was declared in <func decl> on line 10
9  Binding on line 15: GCD was declared in <func decl> on line 10
10 Binding on line 15: n was declared in <param decl> on line 10
11 Binding on line 15: m was declared in <param decl> on line 10
12 Binding on line 15: n was declared in <param decl> on line 10
13 Binding on line 19: res was declared in <var decl> on line 8
14 Binding on line 19: GCD was declared in <func decl> on line 10
15 Binding on line 19: v1 was declared in <const decl> on line 6
16 Binding on line 19: v2 was declared in <const decl> on line 6
17 Binding on line 20: write was declared in <proc decl> in the library
18 Binding on line 20: v1 was declared in <const decl> on line 6
19 Binding on line 20: v2 was declared in <const decl> on line 6
20 Binding on line 20: res was declared in <var decl> on line 8
21 Binding on line 20: eol was declared in <const decl> in the library
```

Figur 4.22: Loggfil med navnebinding for `gcd.pas`

```
1  Type check on line 12: <type name> integer on line 10 {=} <range-type> in the library
2  Type check on line 12: {if} <enum-type> (false, true) in the library
3  Type check on line 13: <type name> integer on line 10 {:=} <type name> integer on line 10
4  Type check on line 15: <type name> integer on line 10 {param #1} <type name> integer on line 10
5  Type check on line 15: <type name> integer on line 10 {mod} <type name> integer on line 10
6  Type check on line 15: <type name> integer on line 10 {param #2} <range-type> in the library
7  Type check on line 15: <type name> integer on line 10 {:=} <type name> integer on line 10
8  Type check on line 19: <type name> integer on line 10 {param #1} <range-type> in the library
9  Type check on line 19: <type name> integer on line 10 {param #2} <range-type> in the library
10 Type check on line 19: <type name> integer on line 8 {:=} <type name> integer on line 10
```

Figur 4.23: Loggfil med typesjekking for `gcd.pas`

```
1  # Code file created by Pascal2100 compiler 2015-08-18 15:19:46
2          .extern write_char
3          .extern write_int
4          .extern write_string
5          .globl  _main
6          .globl  main
7  _main:
8  main:   call    prog$GCD_1              # Start program
9          movl    $0,%eax                 # Set status 0 and
10         ret                             # terminate the program
11 func$GCD_2:
12         enter   $32,$2                  # Start of GCD
13                                         # Start if-statement
14         movl    -8(%ebp),%edx
15         movl    12(%edx),%eax           #   n
16         pushl   %eax
17         movl    $0,%eax                 #   0
18         popl    %ecx
19         cmpl    %eax,%ecx
20         movl    $0,%eax
21         sete    %al                     # Test =
22         cmpl    $0,%eax
23         je      .L0003
24         movl    -8(%ebp),%edx
25         movl    8(%edx),%eax            #   m
26         movl    %eax,-32(%ebp)          # GCD =
27         jmp     .L0004
28 .L0003:
29         movl    -8(%ebp),%edx
30         movl    8(%edx),%eax            #   m
31         pushl   %eax
32         movl    -8(%ebp),%edx
33         movl    12(%edx),%eax           #   n
34         movl    %eax,%ecx
35         popl    %eax
36         cdq
37         idivl   %ecx
38         movl    %edx,%eax               #   mod
39         pushl   %eax                    # Push param #2
40         movl    -8(%ebp),%edx
41         movl    12(%edx),%eax           #   n
42         pushl   %eax                    # Push param #1
43         call    func$GCD_2
44         addl    $8,%esp                 # Pop parameters
45         movl    %eax,-32(%ebp)          # GCD =
46 .L0004:                                 # End if-statement
47         movl    -32(%ebp),%eax          #   Fetch return value
48         leave                           # End of GCD
49         ret
50 prog$GCD_1:
51         enter   $36,$1                  # Start of GCD
52         movl    $462,%eax               #   462
53         pushl   %eax                    # Push param #2
54         movl    $1071,%eax              #   1071
55         pushl   %eax                    # Push param #1
56         call    func$GCD_2
57         addl    $8,%esp                 # Pop parameters
58         movl    -4(%ebp),%edx
59         movl    %eax,-36(%edx)          # res =
60         .data
61 .L0005: .asciz   "GCD("
62         .align  2
63         .text
64         leal    .L0005,%eax             # Addr("GCD(")
65         pushl   %eax                    # Push param #1.
66         call    write_string
67         addl    $4,%esp                 # Pop parameter.
68         movl    $1071,%eax              #   1071
69         pushl   %eax                    # Push param #2.
70         call    write_int
71         addl    $4,%esp                 # Pop parameter.
72         movl    $44,%eax                #   char 44
73         pushl   %eax                    # Push param #3.
74         call    write_char
75         addl    $4,%esp                 # Pop parameter.
```

Figur 4.24: Kodefil produsert fra `gcd.pas` (del 1)

```
76          movl    $462,%eax               #   462
77          pushl   %eax                    # Push param #4.
78          call    write_int
79          addl    $4,%esp                 # Pop parameter.
80          .data
81  .L0006: .asciz   ") = "
82          .align  2
83          .text
84          leal    .L0006,%eax             # Addr(") = ")
85          pushl   %eax                    # Push param #5.
86          call    write_string
87          addl    $4,%esp                 # Pop parameter.
88          movl    -4(%ebp),%edx
89          movl    -36(%edx),%eax          #   res
90          pushl   %eax                    # Push param #6.
91          call    write_int
92          addl    $4,%esp                 # Pop parameter.
93          movl    $10,%eax                #   char 10
94          pushl   %eax                    # Push param #7.
95          call    write_char
96          addl    $4,%esp                 # Pop parameter.
97          leave                           # End of GCD
98          ret
```

Figur 4.25: Kodefil produsert fra `gcd.pas` (del 2)

Kapittel 5

Programmeringsstil

5.1 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.

5.1.1 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 6.1 på side 69.)

4) Selve klassen.

5.1.2 Variabler

Variabler bør deklareres én og én på hver linje:

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

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

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

```
do {
    setninger;
} while (uttrykk);

for (init;  betingelse;  oppdatering) {
    setninger;
}

if (uttrykk) {
    setninger;
}

if (uttrykk) {
    setninger;
} else {
    setninger;
}

if (uttrykk) {
    setninger;
} else if (uttrykk) {
    setninger;
} else if (uttrykk) {
    setninger;
}

return uttrykk;

switch (uttrykk) {
case xxx:
   setninger;
   break;

case xxx:
    setninger;
    break;

default:
    setninger;
    break;
}

try {
    setninger;
} catch (ExceptionClass e) {
    setninger;
}

while (uttrykk) {
    setninger;
}
```

Figur 5.1: Suns forslag til hvordan setninger bør skrives

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

5.1.3 Setninger

Enkle setninger bør stå én og én på hver linje:

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

De ulike sammensatte setningene skal se ut slik figur 5.1 viser. De skal alltid ha {} rundt innmaten, og innmaten skal indenteres 4 posisjoner.

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

Tabell 5.1: Suns forslag til navnevalg i Java-programmer

5.1.4 Navn

Navn bør velges slik det er angitt i tabell 5.1.

5.1.5 Utseende

5.1.5.1 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.

5.1.5.2 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.

5.1.5.3 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 6

Dokumentasjon

6.1 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 kodefilene 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/.

6.1.1 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.

6.1.2 Eksempel

I figur 6.1 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 6.1: Java-kode med JavaDoc-kommentarer

6.2 «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 hjelpemidler som figurer, matematiske formler, fotnoter, kapittelinndeling, fontskifte og annet.

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

6.2.1 Et eksempel

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

1) Skriv kildefilen `bubble.w0` (vist i figur 6.2 og 6.3). Dette gjøres med en vanlig tekstbehandler som for eksempel Emacs.
2) Bruk programmet `weave0`[1] til å lage det ferdige dokumentet som er vist i figur 6.4–6.7:

```
$ 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 web_0; for mer informasjon, se http://dag.at.ifi.uio.no/public/doc/web0.pdf.

bubble.w0 del 1

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

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

\begin{document}
\maketitle

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

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


<<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 6.2: «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 6.3: «Lesbar programmering» — kildefilen bubble.w0 del 2

Bubble sort

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

July 29, 2015

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:

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

#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 #4a (p.2). It is used in #1 (p.1).)

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

File: *bubble.w0* page 1

Figur 6.4: «Lesbar programmering» — utskrift side 1

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

#5 ⟨*swap a[i] and a[i+1]*⟩ ≡

```
14  temp = a[i];  a[i] = a[i+1];  a[i+1] = temp;
15  ++n_swaps;
```

(This code is used in #3 (p.1).)

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

#4a ⟨*local variables* #4 (p.1)⟩ +≡

```
16  int temp, n_swaps;
```

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

#6 ⟨*initialize*⟩ ≡

```
17  n_swaps = 0;
```

(This code is used in #3 (p.1).)

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

#7 ⟨*not sorted*⟩ ≡

```
18  n_swaps > 0
```

(This code is used in #2 (p.1).)

File: *bubble.w0* page 2

Figur 6.5: «Lesbar programmering» — utskrift side 2

Variables

A

a 1, 12, 14

I

i 11, 12, 13, 14

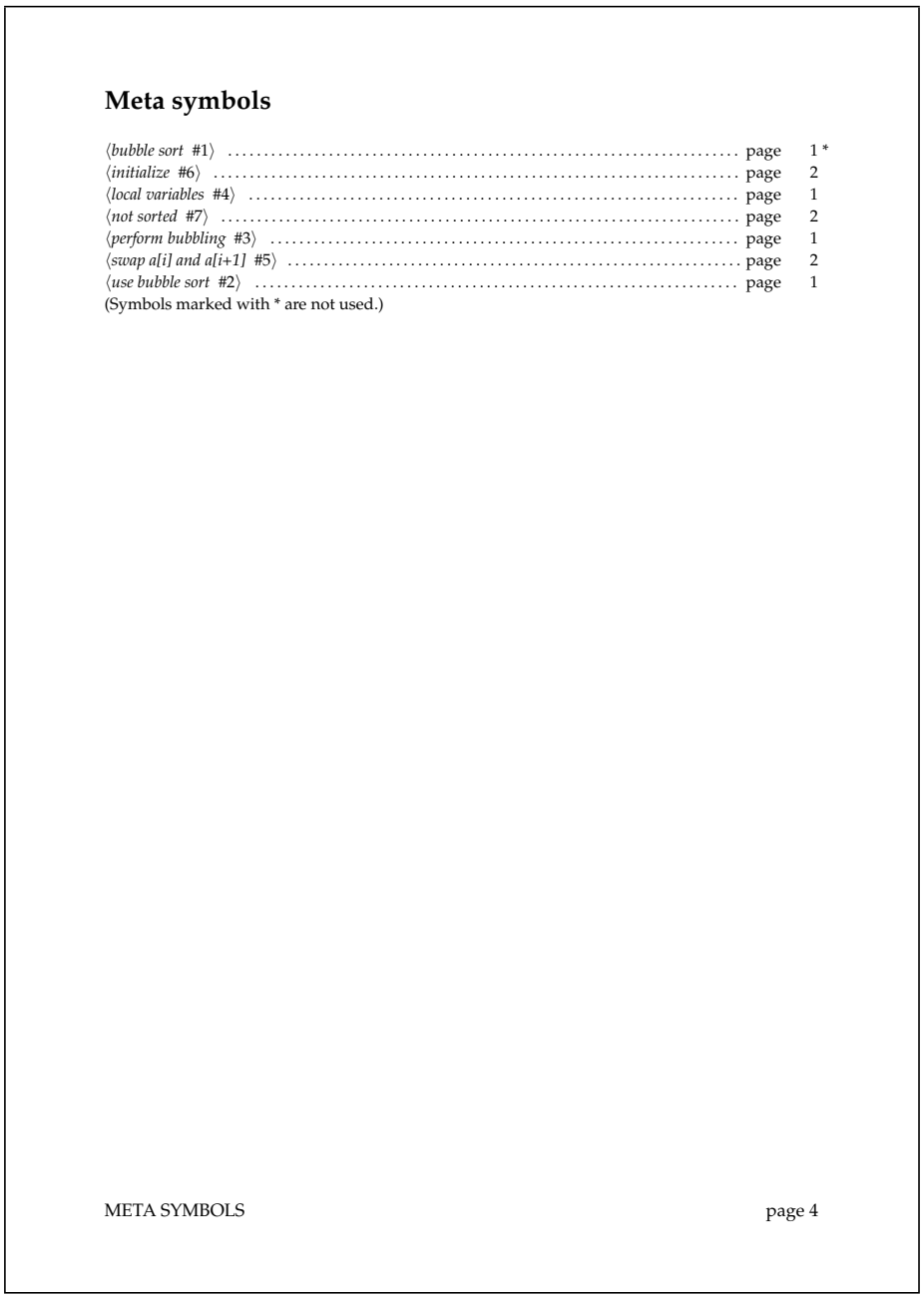
N

n 1, 11
n_swaps 15, 16, 17, 18

T

temp 14, 16

Figur 6.6: «Lesbar programmering» — utskrift side 3



Meta symbols

⟨*bubble sort* #1⟩ ..... page 1 *
⟨*initialize* #6⟩ ..... page 2
⟨*local variables* #4⟩ ..... page 1
⟨*not sorted* #7⟩ ..... page 2
⟨*perform bubbling* #3⟩ ..... page 1
⟨*swap a[i] and a[i+1]* #5⟩ ..... page 2
⟨*use bubble sort* #2⟩ ..... page 1
(Symbols marked with * are not used.)

META SYMBOLS page 4

Figur 6.7: «Lesbar programmering» — utskrift side 4

Register