

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

En kompilator for Pascal

Kompendium for INF2100

Stein Krogdahl, Dag Langmyhr
Høsten 2016



Innhold

Forord	9
1 Innledning	11
1.1 Hva er emnet INF2100?	11
1.2 Hvorfor lage en kompilator?	12
1.3 Om kompilatorer og liknende verktøy	12
1.3.1 Preprosessorer	13
1.3.2 Interpreting	14
1.3.3 Kompilering og kjøring av Java-programmer	14
1.4 Språkene i oppgaven	15
1.4.1 Programmeringsspråket Pascal2016	15
1.4.2 Prosessoren x86 og dens maskinspråk	15
1.4.3 Assembleren	15
1.4.4 Oversikt over de ulike språkene i oppgaven	16
1.5 Oppgaven og dens fire deler	16
1.5.1 Del 1: Skanneren	17
1.5.2 Del 2: Parseren	17
1.5.3 Del 3: Sjekking	17
1.5.4 Del 4: Kodegenerering	17
1.6 Krav til samarbeid og gruppetilhørighet	18
1.7 Kontroll av innlevert arbeid	18
1.8 Delta på øvingsgruppene	19
2 Programmering i Pascal2016	21
2.1 Kjøring	21
2.2 Pascal2016-program	23
2.2.1 Blokker	23
2.2.2 Setninger	25
2.2.3 Uttrykk	28
2.2.4 Andre ting	28
2.3 Predefinerte deklarasjoner	29
2.3.1 Utskrift	30
2.4 Forskjeller til standard Pascal	30
3 Datamaskinen x86	31
3.1 Minnet	31
3.2 Prosessoren x86	31
3.3 Assemblerkode	33
3.3.1 Assemblerdirektiver	34
4 Prosjektet	35

4.1	Diverse informasjon om prosjektet	35
4.1.1	Basiskode	35
4.1.2	Oppdeling i moduler	36
4.1.3	Selvidentifikasjon	36
4.1.4	Logging	37
4.1.5	Testprogrammer	37
4.1.6	På egen datamaskin	37
4.1.7	Tegnsett	38
4.2	Del 1: Skanneren	38
4.2.1	Representasjon av symboler	39
4.2.2	Skanneren	39
4.2.3	Logging	39
4.2.4	Mål for del 1	41
4.3	Del 2: Parsing	41
4.3.1	Implementasjon	41
4.3.2	Parsing	43
4.3.3	Syntaksfeil	44
4.3.4	Logging	44
4.4	Del 3: Sjekking	45
4.4.1	Sjekke navn ved deklarasjoner	45
4.4.2	Sjekke deklarasjoner	45
4.4.3	Sjekke navnebruk	46
4.4.4	Bestemme typer	46
4.4.5	Beregne konstanter	46
4.5	Del 4: Kodegenerering	47
4.5.1	Konvensjoner	47
4.5.2	Registre	47
4.5.3	Oversettelse av uttrykk	48
4.5.4	Oversettelse av setninger	50
4.5.5	Oversettelse av funksjoner og prosedyrer	52
4.5.6	Deklarasjon av variabler	53
4.6	Et litt større eksempel	55
5	Kompilering av blokkorienterte språk	65
5.1	Bakgrunn	65
5.1.1	Kontekstvektor	65
5.1.2	Et eksempel	65
5.2	Start av hovedprogrammet	66
5.3	Start av en prosedyre	66
5.4	Start av en indre funksjon	67
6	Programmeringsstil	71
6.1	Suns anbefalte Java-stil	71
6.1.1	Klasser	71
6.1.2	Variabler	71
6.1.3	Setninger	72
6.1.4	Navn	73
6.1.5	Utseende	73
7	Dokumentasjon	75
7.1	JavaDoc	75

7.1.1	Hvordan skrive JavaDoc-kommentarer	75
7.1.2	Eksempel	76
7.2	«Lesbar programmering»	76
7.2.1	Et eksempel	77
Register		85

Figurer

1.1	Sammenhengen mellom Pascal2016, kompilator, assembler og en x86-maskin	16
2.1	Eksempel på et Pascal2016-program	22
2.2	Jernbanediagram for <program>	23
2.3	Jernbanediagram for <name>	23
2.4	Jernbanediagram for <block>	23
2.5	Jernbanediagram for <const decl part>	24
2.6	Jernbanediagram for <const decl>	24
2.7	Jernbanediagram for <constant>	24
2.8	Jernbanediagram for <unsigned constant>	24
2.9	Jernbanediagram for <numeric literal>	24
2.10	Jernbanediagram for <prefix opr>	24
2.11	Jernbanediagram for <char literal>	24
2.12	Jernbanediagram for <var decl part>	25
2.13	Jernbanediagram for <var decl>	25
2.14	Jernbanediagram for <type>	25
2.15	Jernbanediagram for <type name>	25
2.16	Jernbanediagram for <array-type>	25
2.17	Jernbanediagram for <func decl>	25
2.18	Jernbanediagram for <proc decl>	25
2.19	Jernbanediagram for <param decl list>	26
2.20	Jernbanediagram for <param decl>	26
2.21	Jernbanediagram for <statm list>	26
2.22	Jernbanediagram for <statement>	26
2.23	Jernbanediagram for <empty statm>	26
2.24	Jernbanediagram for <assign statm>	26
2.25	Jernbanediagram for <variable>	27
2.26	Jernbanediagram for <proc call>	27
2.27	Jernbanediagram for <if-statm>	27
2.28	Jernbanediagram for <while-statm>	27
2.29	Jernbanediagram for <compound statm>	27
2.30	Jernbanediagram for <expression>	28
2.31	Jernbanediagram for <rel opr>	28
2.32	Jernbanediagram for <simple expr>	28
2.33	Jernbanediagram for <term opr>	28
2.34	Jernbanediagram for <term>	28
2.35	Jernbanediagram for <factor opr>	29
2.36	Jernbanediagram for <factor>	29
2.37	Jernbanediagram for <func call>	29
2.38	Jernbanediagram for <inner expr>	29

2.39	Jernbanediagram for <negation>	29
3.1	Hovedkortet i en datamaskin	33
3.2	Instruksjonslinje i assemblerkode	33
4.1	Oversikt over prosjektet	35
4.2	De fire modulene i kompilatoren	36
4.3	Et minimalt Pascal2016-program <code>mini.pas</code>	38
4.4	Klassen <code>Token</code>	39
4.5	Enum-klassen <code>TokenKind</code>	40
4.6	Klassen <code>Scanner</code>	40
4.7	Skanning av <code>mini.pas</code>	41
4.8	Syntakstreet laget utifra testprogrammet <code>mini.pas</code>	42
4.9	Klassen <code>WhileStatm</code>	43
4.10	Parsering av <code>mini.pas</code>	44
4.11	Utskrift av treet til <code>mini.pas</code>	45
4.12	Navnebinding i <code>mini.pas</code>	45
4.13	Et program med konstanter	47
4.14	Generert kodelfil for <code>mini.pas</code>	48
4.15	Et litt større Pascal2016-program <code>gcd.pas</code>	55
4.16	Skanning av <code>gcd.pas</code> (del 1)	55
4.17	Skanning av <code>gcd.pas</code> (del 2)	56
4.18	Parsering av <code>gcd.pas</code> (del 1)	57
4.19	Parsering av <code>gcd.pas</code> (del 2)	58
4.20	Parsering av <code>gcd.pas</code> (del 3)	59
4.21	Parsering av <code>gcd.pas</code> (del 4)	60
4.22	Parsering av <code>gcd.pas</code> (del 5)	61
4.23	Utskrift av treet til <code>gcd.pas</code>	61
4.24	Navnebinding i <code>gcd.pas</code>	61
4.25	Typesjekking i <code>gcd.pas</code>	62
4.26	Generert kodelfil for <code>gcd.pas</code> (del 1)	62
4.27	Generert kodelfil for <code>gcd.pas</code> (del 2)	63
5.1	En enkelt testprogram	66
5.2	Stakken når hovedprogrammet starter	67
5.3	Stakken når prosedyren har startet	68
5.4	Stakken når den indre funksjonen har startet	69
5.5	Assemblerkoden generert for programmet i figur 5.1 på side 66	70
6.1	Suns forslag til hvordan setninger bør skrives	72
7.1	Java-kode med JavaDoc-kommentarer	76
7.2	«Lesbar programmering» – kildefilen <code>bubble.w0</code> del 1	78
7.3	«Lesbar programmering» – kildefilen <code>bubble.w0</code> del 2	79
7.4	«Lesbar programmering» – utskrift side 1	80
7.5	«Lesbar programmering» – utskrift side 2	81
7.6	«Lesbar programmering» – utskrift side 3	82
7.7	«Lesbar programmering» – utskrift side 4	83

Tabeller

3.1	x86-instruksjoner brukt i prosjektet	32
3.2	Assemblerdirektiver	34
4.1	Opsjoner for logging	37
4.2	Notasjon for typesjekking	46
4.3	Typesjekking av setninger	46
4.4	Typesjekking av uttrykk	47
4.5	Kode for å hente en verdi inn i %EAX	49
4.6	Kode generert av unære operatorer i uttrykk	49
4.7	Kode generert av binære operatorer i uttrykk	50
4.8	Kode generert av tom setning	50
4.9	Kode generert av sammensatt setning	50
4.10	Kode generert av tilordning	51
4.11	Kode generert av prosedyrekall	51
4.12	Kode generert av kall på <code>write</code>	52
4.13	Kode generert av if-setning	52
4.14	Kode generert av while-setning	53
4.15	Kode generert av funksjons- og prosedyredeklarasjon	53
4.16	Kode generert av hovedprogrammet	54
6.1	Suns forslag til navnevalg i Java-programmer	73

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 renoveret av *Dag Langmyhr*: Minila ble erstattet av en minimal variant av C kalt *RusC* og datamaskinen Flink ble avløst av en annen ikkeksisterende maskin kalt *Rask*. I 2010 ble det besluttet å lage ekte kode for Intel-prosessoren x86 slik at den genererte koden 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 2016, og hele opplegget har gått gjennom en revisjon. Det gjelder også språket som skal kompileres: fra og med 2016 er det *Pascal2016* 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 studentene *Einar Løvhøiden Antonsen, Jonny Bekkevold, Eivind Alexander Bergem, Marius Ekeberg, Arne Olav Hallingstad, Espen Tørressen Hangård, Sigmund Hansen, Simen Heggestøy, Brendan Johan Lee, Håvard Koller Noren, Vegard Nossum, Hans Jørgen Nygårdshaug, David J Ofstedal, Mikael Olausson, Cathrine Elisabeth Olsen, Bendik Rønning Opstad, Christian Resell, Christian Andre Finnøy Ruud, Ryhor Sivuda, Yrjab Skrimstad, Herman Torjussen, Christian Tryti, Jørgen Vigdal, Olga Voronkova, Aksel L Webster* og *Sindre Wilting* som har påpekt skrivefeil eller foreslått forbedringer i tidligere utgaver. Om flere studenter gjør dette, vil de også få navnet sitt på trykk.

Blindern, 22. august 2016
Stein Krogdahl Dag Langmyhr

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

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

— Forfatterne

Kapittel I

Innledning

1.1 Hva er emnet INF2100?

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

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

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

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

I dette kompendiet beskrives stort sett bare selve programmeringsoppgaven som skal løses. I tillegg til dette kan det komme ytterligere krav, for eksempel angående bruk av verktøy eller skriftlige arbeider som skal leveres. Dette vil i så fall bli opplyst om på forelesningene og på kursets 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 uoversiktig oppgave. Noe av poenget med kurset er å demonstrere at med en hensiktsmessig oppsplittning av programmet i deler som hver tar ansvaret for en avgrenset del av oppgaven, så kan både de enkelte deler og den helheten de danner, bli høyst medgjørlig. Det er denne erfaringen, og forståelsen av hvordan slik oppdeling kan gjøres på et reelt eksempel, som er det viktigste studentene skal få med seg fra dette kurset.

Vi skal i neste avsnitt se litt mer på hva en kompilator er og hvordan den står i forhold til liknende verktøy. Det vil da også raskt bli klart at det å skrive en kompilator for et «ekte» programmeringsspråk som skal oversettes til maskinspråket til en datamaskin, vil bli en altfor omfattende oppgave. Vi skal derfor forenkle oppgaven en del ved å lage vårt eget lille programmeringsspråk **Pascal2016**. 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 maskinstruksjoner for en gitt datamaskin. Det er slike oversettelsesprogrammer som kalles komplilatorer. En komplilator er altså et helt vanlig program som leser data inn og leverer data fra seg. Dataene det leser inn er et tekstlig program (i det programmeringsspråket denne komplilatoren skal oversette fra), og data det leverer fra seg er en sekvens av maskinstruksjoner for den aktuelle maskinen. Disse maskinstruksjonene vil komplilatoren vanligvis legge på en fil i et passelig format med tanke på at de senere kan kopieres inn i en maskin og bli utført.

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

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

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

1.3.1 Preprosessorer

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

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

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

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

1.3.2 Interpreting

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

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

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

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

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

1.3.3 Kompilering og kjøring av Java-programmer

En av de opprinnelige ideene ved Java var knyttet til datanett ved at et program skulle kunne kompileres på én maskin for så å kunne sendes over nettet til en hvilken som helst annen maskin (for eksempel som en såkalt *applet*) og bli utført der. For å få til dette definerte man en tenkt datamaskin kalt *Java Virtual Machine* (JVM) og lot kompilatorene produsere maskinkode (gjerne kalt *byte-kode*) for denne maskinen. Det er imidlertid ingen datamaskin som har elektronikk for direkte å utføre slik byte-kode, og maskinen der programmet skal utføres må derfor ha et program som simulerer JVM-maskinen og dens utføring av byte-kode. Vi kan da gjerne si at et slikt simulatingsprogram interpreterer maskinkoden til JVM-maskinen. I dag har for eksempel de fleste nettlesere (Firefox, Opera,

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

Slik interperetering av maskinkode går imidlertid normalt en del saktere enn om man hadde oversatt til «ekte» maskinkode og kjørt den direkte på «hardware». Typisk kan dette for Javas byte-kode gå 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 interperitere byte-koden, oversetter den videre til den aktuelle maskinkoden umiddelbart før programmet startes opp. Dette kan gjøres for hele programmer, eller for eksempel for klasse etterhvert som de tas i bruk første gang.

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

1.4 Språkene i oppgaven

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

1.4.1 Programmeringsspråket Pascal2016

Det å lage en kompilator for til dømes Java ville sprengje kursrammen på ti studiepoeng. I stedet har vi laget et språk spesielt for dette kurset med tanke på at det skal være overkommelig å oversette. Dette språket er en miniversjon av Pascal kalt *Pascal2016*. 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 Pascal2016 blir beskrevet i detalj i kapittel 2 på [side 21](#).

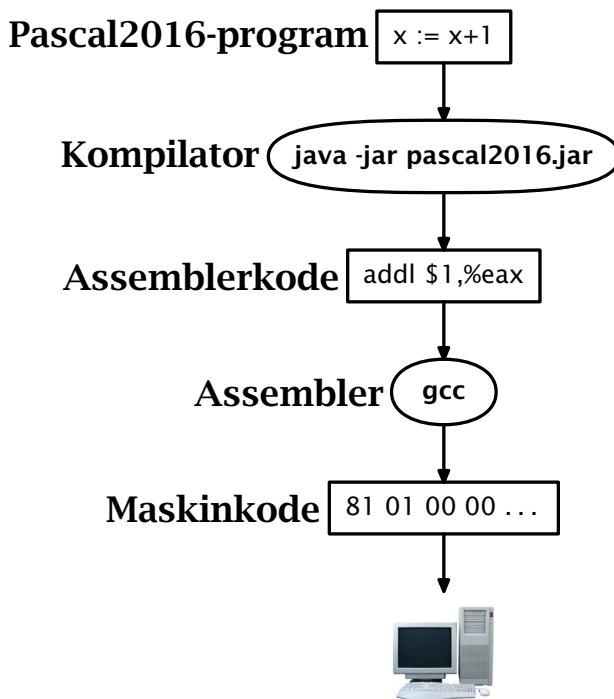
1.4.2 Prosessoren x86 og dens maskinspråk

En kompilator produserer vanligvis kode for en gitt prosessor og det skal også vår Pascal2016-kompilator gjøre. Som prosessor er valgt *x86* siden den finnes overalt, for eksempel på Ifis datalababer og i de aller fleste hjemmemaskiner. Dermed kan dere teste koden uansett hvor dere måtte befinner dere.

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

1.4.3 Assembleren

Når man skal programmere direkte i maskinstruksjoner, er det svært tungt å bruke tallkoder hele tiden, og så godt som alle maskiner har derfor laget en tekstkode som er lettere å huske enn et tall. For eksempel



Figur 1.1: Sammenhengen mellom Pascal2016, kompilator, assembler og en x86-maskin

kan man bruke «`addl`» for en instruksjon som legger sammen 32-bits hel-tall 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 oversetter-programmene kalles tradisjonelt **assembler**e. 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) **Pascal2016**, som kompilatoren skal oversette fra.
- 2) **Java**, som Pascal2016-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 verktøy bruk

eller levering av skriftlige tilleggsarbeider, men også dette vil i så fall bli annonseret i god tid.

Hele programmet kan grovt regnet bli på fra 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 Pascal2016s **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 Pascal2016-språket.

Denne «reneskårne» sekvensen av symboler vil være det grunnlaget som resten av komplilatoren 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 Pascal2016-program skal ha (altså, at den følger Pascal2016s **syntaks**).

Om alt er i orden, skal del 2 bygge opp et **syntakstre**, en **trestruktur** av objekter som direkte representerer det aktuelle Pascal2016-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 komplilatoren vår gjøre selve oversettelsen til x86-kode; da tar vi igjen utgangspunkt i den trestrukturen som del 2 produserte for det aktuelle Pascal2016-programmet. Koden skal legges på en fil og den skal være i såkalt x86 assemblerformat.

I avsnitt 4.5 på side 47 er det angitt hvilke sekvenser av x86-instruksjoner hver enkelt Pascal2016-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 gruppertilhørighet

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 komplataoren de har skrevet. Med litt støtte og hint skal de for eksempel kunne gjenskape deler av selve programmet på en tavle.

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

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

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

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

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

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

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

1.8 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 Pascal2016-kompilatoren. Om man tar en liten titt på oppgavene før gruppetime, vil man antagelig få svært mye mer ut av gjennomgåelsen.

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

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

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

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

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

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

Kapittel 2

Programmering i Pascal2016

Programmeringsspråket **Pascal2016** 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 2.39 på side 23–29 og bør være lett forståelig for alle som har programmert litt i Java. Et eksempel på et Pascal2016-program er vist i figur 2.1 på neste side.¹

2.1 Kjøring

Inntil dere selv har laget en Pascal2016-kompilator, kan dere benytte referansekompiletoren:

```
$ ~inf2100/pascal2016 primes.pas
This is the Ifi Pascal2016 compiler (2016-05-04) running on Linux
Parsing... checking... generating code...OK
Running gcc -m32 -o primes primes.s -L. -L/hom/inf2100 -lpas2016
$ ./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

```

1  /* This program prints all primes less than 1000
2   using a technique called "The sieve of Eratosthenes". */
3
4  program Primes;
5
6  const Limit = 1000;
7
8  var prime : array [2..Limit] of Boolean;
9    i      : integer;
10
11 procedure FindPrimes;
12 var i1 : integer;
13     I2 : Integer;
14 begin
15   i1 := 2;
16   while i1 <= Limit do
17     begin
18       i2 := 2*i1;
19       while i2 <= Limit do
20         begin
21           prime[i2] := false;
22           i2 := i2+i1
23         end;
24       i1 := i1 + 1
25     end
26   end; {FindPrimes}
27
28 procedure P4 (x : integer);
29 begin
30   if x < 1000 then write(' ');
31   if x < 100 then write(' ');
32   if x < 10 then write(' ');
33   write(x);
34 end; {P4}
35
36 procedure PrintPrimes;
37 var i      : integer;
38     NPrinted : integer;
39 begin
40   i := 2; NPrinted := 0;
41   while i <= Limit do
42     begin
43       if prime[i] then
44         begin
45           if (NPrinted > 0) and (NPrinted mod 10 = 0) then write(eol);
46           P4(i); NPrinted := NPrinted + 1;
47         end;
48       i := i + 1;
49     end;
50   write(eol)
51 end; {PrintPrimes}
52
53 begin {main program}
54   i := 2;
55   while i <= Limit do begin prime[i] := true; i := i+1 end;
56
57   /* Find and print the primes: */
58   FindPrimes; PrintPrimes;
59 end. {main program}
60

```

Figur 2.1: Eksempel på et Pascal2016-program

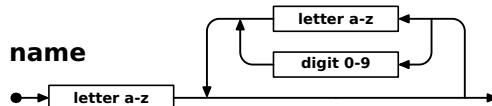
2.2 Pascal2016-program

Som vist i figur 2.2 er et Pascal2016-program rett og slett en `<block>` med et navn.

program



Figur 2.2: Jernbanediagram for `<program>`



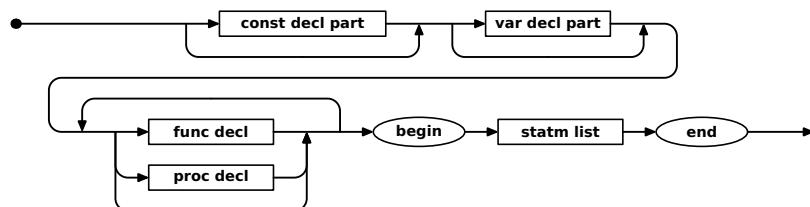
Figur 2.3: Jernbanediagram for `<name>`

Forskjell fra Java: I Pascal2016 skiller man ikke på store og små bokstaver, så `integer`, `Integer` og `INTEGER` er tre ulike former av samme navn.

2.2.1 Blokker

Pascal2016 er et såkalt **blokkstrukturert** programmeringsspråk der alle deklarasjoner er plassert i blokker; se figur 2.4. Innmaten av program, funksjoner og prosedyrer er alle blokker.

block



Figur 2.4: Jernbanediagram for `<block>`

Forskjell fra Java: I Pascal2016 kan man ha blokker utenpå hverandre i så mange nivåer man ønsker, og man kan ha funksjoner som er lokale inni andre funksjoner. Dette er ikke mulig i Java.

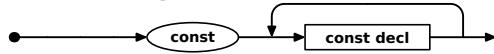
2.2.1.1 Konstantdeklarasjoner

I en blokk kan man deklarere **konstanter**, dvs navngitte verdier kjent av kompilatoren. En konstant kan defineres som

- et navn på en annen konstant, muligens med skifte av fortegn
- en heltallsliteral²
- en tegnliteral (f eks 'A')

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 deklareret som uforanderlig under kjøringen.

const decl part



Figur 2.5: Jernbanediagram for <const decl part>

const decl



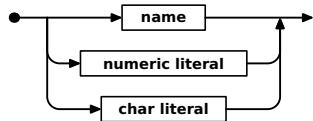
Figur 2.6: Jernbanediagram for <const decl>

constant



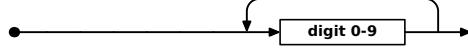
Figur 2.7: Jernbanediagram for <constant>

unsigned constant



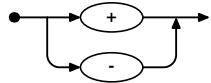
Figur 2.8: Jernbanediagram for <unsigned constant>

numeric literal



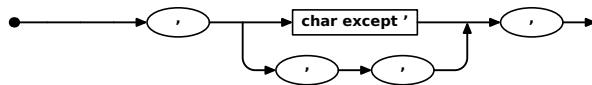
Figur 2.9: Jernbanediagram for <numeric literal>

prefix opr



Figur 2.10: Jernbanediagram for <prefix opr>

char literal



Figur 2.11: Jernbanediagram for <char literal>

Forskjell fra Java: Der man i Java skriver «'\''» for å få et enkelt anførselstegn, skriver vi i Pascal2016 «''''».

2.2.1.2 Variabeldeklarasjoner

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

Forskjell fra Java: I Java kan man angi en initialverdi for variabelen; det kan man ikke i Pascal2016.

var decl part



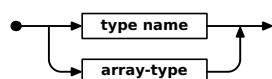
Figur 2.12: Jernbanediagram for <var decl part>

var decl



Figur 2.13: Jernbanediagram for <var decl>

type



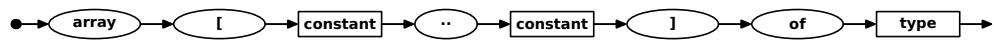
Figur 2.14: Jernbanediagram for <type>

type name



Figur 2.15: Jernbanediagram for <type name>

array-type

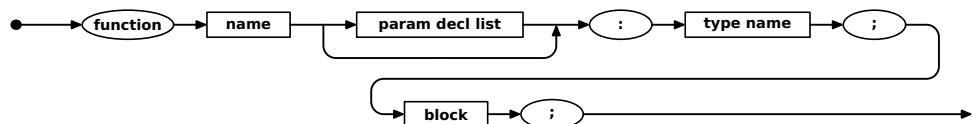


Figur 2.16: Jernbanediagram for <array-type>

2.2.1.3 Funksjons- og prosedyredeklarasjoner

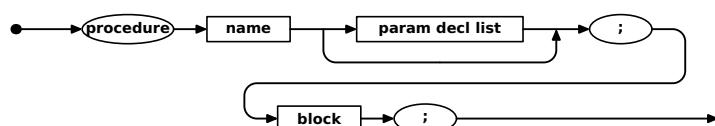
I Pascal2016 skiller man mellom **funksjoner** og **prosedyrer**: de førstnevnte returnerer alltid en verdi, de sistnevnte kan ikke det.

func decl



Figur 2.17: Jernbanediagram for <func decl>

proc decl

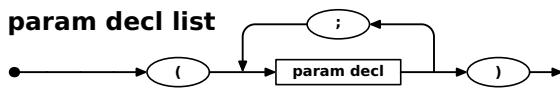
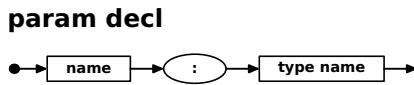
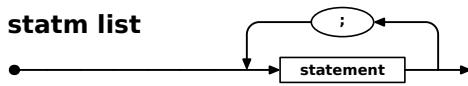
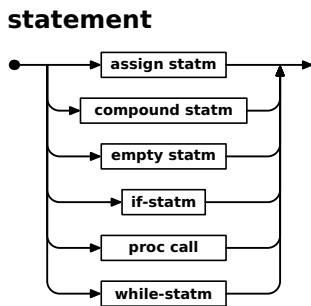


Figur 2.18: Jernbanediagram for <proc decl>

Forskjell fra Java: I Pascal2016 har man ingen **return**-setning; i stedet tilordner man en verdi til funksjonen selv. Du kan se et eksempel på det i GCD-funksjonen vist i figur 4.15 på side 55.

2.2.2 Setninger

Pascal2016 har mange av de samme setningene som C og Java.

**Figur 2.19:** Jernbanediagram for `(param decl list)`**Figur 2.20:** Jernbanediagram for `(param decl)`**Figur 2.21:** Jernbanediagram for `(statm list)`**Figur 2.22:** Jernbanediagram for `(statement)`

Forskjell fra Java: I Pascal2016 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.

empty statm

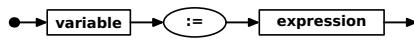
**Figur 2.23:** Jernbanediagram for `(empty statm)`

2.2.2.2 Tilordning

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

Forskjell fra Java: I Pascal2016 benyttes symbolet `:=` for tilordning.

assign statm

**Figur 2.24:** Jernbanediagram for `(assign statm)`

variable

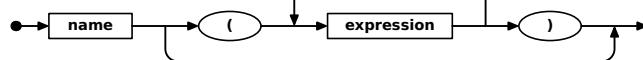


Figur 2.25: Jernbanediagram for `<variable>`

2.2.2.3 Prosedyrekall

Denne setningen kaller en prosedyre.

proc call



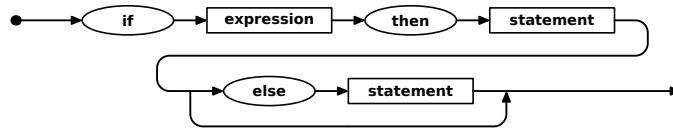
Figur 2.26: Jernbanediagram for `<proc call>`

Forskjell fra Java: I Pascal2016 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.

if-statm

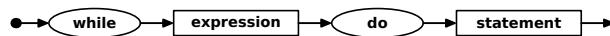


Figur 2.27: Jernbanediagram for `<if-statm>`

2.2.2.5 While-setninger

Disse setningene oppfører seg også slik vi er vant til.

while-statm



Figur 2.28: 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 2.29: Jernbanediagram for `<compound statm>`

2.2.3 Uttrykk

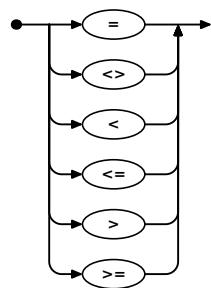
Uttrykk i Pascal2016 er ganske likt de språkene vi kjenner. Det benyttes ganske mange definisjoner for å sikre at presedensen³ blir riktig.

expression



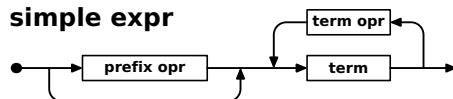
Figur 2.30: Jernbanediagram for `<expression>`

rel opr



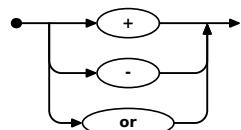
Figur 2.31: Jernbanediagram for `<rel opr>`

Forskjell fra Java: Testene på likhet og ulikhet angis med henholdsvis `=` og `<>` i Pascal2016 (og de tilsvarer altså `==` og `!=` i Java).

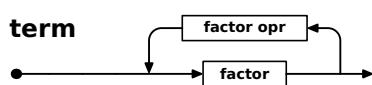


Figur 2.32: Jernbanediagram for `<simple expr>`

term opr



Figur 2.33: Jernbanediagram for `<term opr>`



Figur 2.34: Jernbanediagram for `<term>`

2.2.4 Andre ting

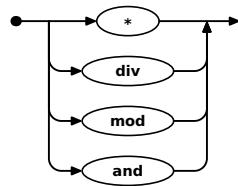
Det er et par andre ting man bør merke seg ved Pascal2016:

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

$$a + b \times c$$

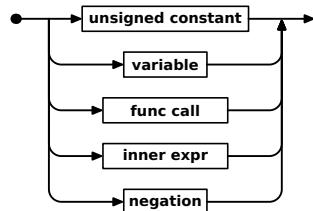
tolkes dette vanligvis som $a + (b \times c)$ fordi \times normalt har høyere presedens enn $+$, dvs \times binder sterkere enn $+$.

factor opr



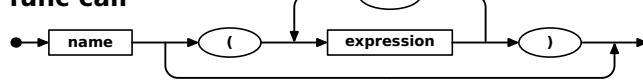
Figur 2.35: Jernbanediagram for {factor opr}

factor



Figur 2.36: Jernbanediagram for {factor}

func call



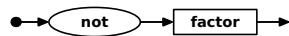
Figur 2.37: Jernbanediagram for {func call}

inner expr



Figur 2.38: Jernbanediagram for {inner expr}

negation



Figur 2.39: Jernbanediagram for {negation}

- Kommentarer kan skrives på to ulike former:

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

Begge formene kan strekke seg over flere linjer.

2.3 Predefinerte deklarasjoner

I Pascal2016 er disse deklarasjonene ferdig definert og kan brukes i programmene:

Boolean er en type som har to logiske verdier: `False` og `True`.

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 38).

EoL⁴ er en konstant av typen Char. Internrepresentasjonen er 10 og konstanten brukes ved utskrift for å angi linjeskift.

False er en konstant av typen Boolean; den representeres internt med verdien 0.

Integer er typen for heltall.

True er en konstant av typen Boolean; den representeres internt med verdien 1.

Write er standardprosedyren for utskrift; den er beskrevet i neste avsnitt.

2.3.1 Utskrift

Pascal2016-programmer kan skrive ut ved å benytte den helt spesielle prosedyren **write**. Denne prosedyren kan ha vilkårlig mange parametre, og parametrene kan være av typen **integer**, **char** eller **Boolean**. Her er noen eksempler:

```
write('H', 'e', 'i', '!', eol);
write(2, '+', 2, '=', 2+2, eol);
```

2.4 Forskjeller til standard Pascal

Som nevnt er Pascal2016 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
- typedeklarasjoner
- sjekking under kjøring av array-grenser og andre verdier

⁴ Navnet **EoL** er en forkortelse for «end-of-line».

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 side 33. I denne omgang er vi bare interessert i prosessoren og minnet, så dette kapittelet er ingen utfyllende beskrivelse av hvordan en datamaskin er bygget opp – det forteller kun akkurat det vi trenger å vite for å skrive kompilatoren vår.

3.1 Minnet

Minnet består av tre deler:

Datadelen brukes til å lagre globale variabler.

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

Kodedelen inneholder programkoden, altså programmet som utføres.

3.2 Prosessoren x86

x86-prosessoren er en 32-bits⁵ prosessor som inneholder fire viktige deler:

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

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

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

%EAX %ECX %EDX %EBP %ESP

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

⁵ 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.

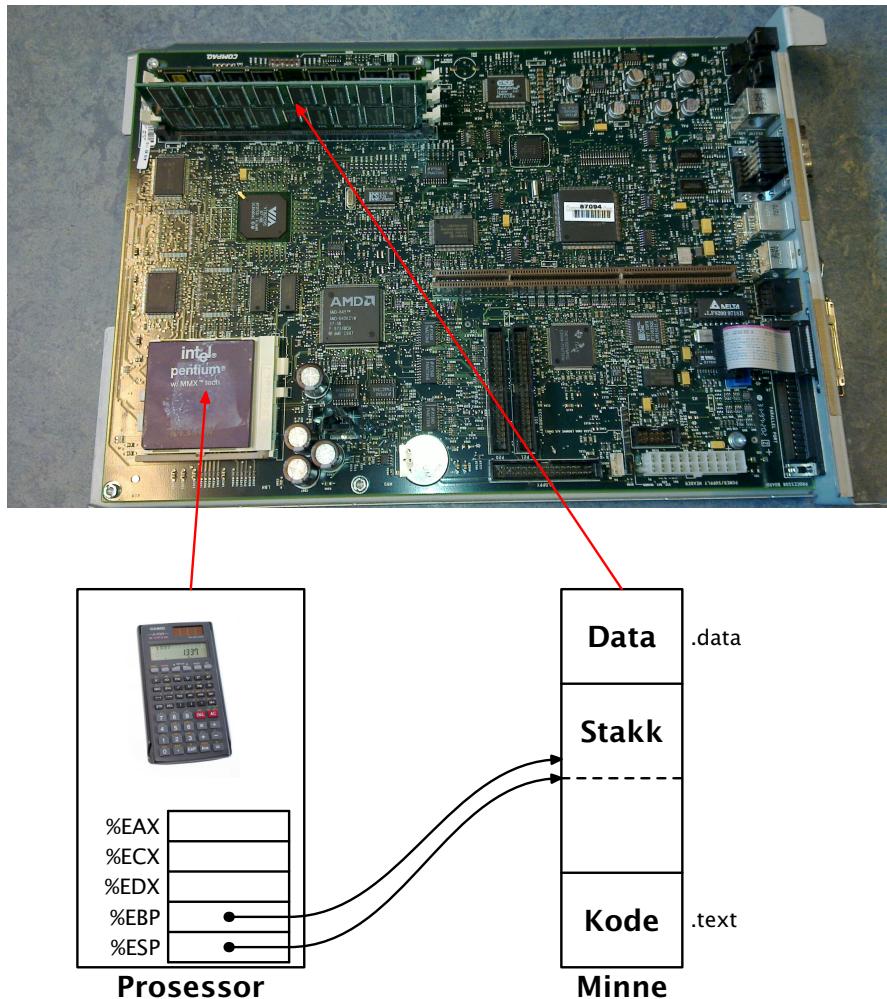
<code>movl</code>	$\langle v_1 \rangle, \langle v_2 \rangle$	Flytt $\langle v_1 \rangle$ til $\langle v_2 \rangle$.
<code>cdq</code>		Omform 32-bits %EAX til 64-bits %EDX:%EAX.
<code>leal</code>	$\langle v_1 \rangle, \langle v_2 \rangle$	Flytt $\langle v_1 \rangle$'s adresse til $\langle v_2 \rangle$.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



Figur 3.1: Hovedkortet med prosessor og minne i en datamaskin

3.3 Assemblerkode

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

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

Figur 3.2: Instruksjonslinje i assemblerkode

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

Instruksjon er en av instruksjonene i tabell 3.1 på forrige side.

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 **direktiver** som er en form for beskjed til assembleren. Vi skal bruke det direktivet som er vist i tabell 3.2.

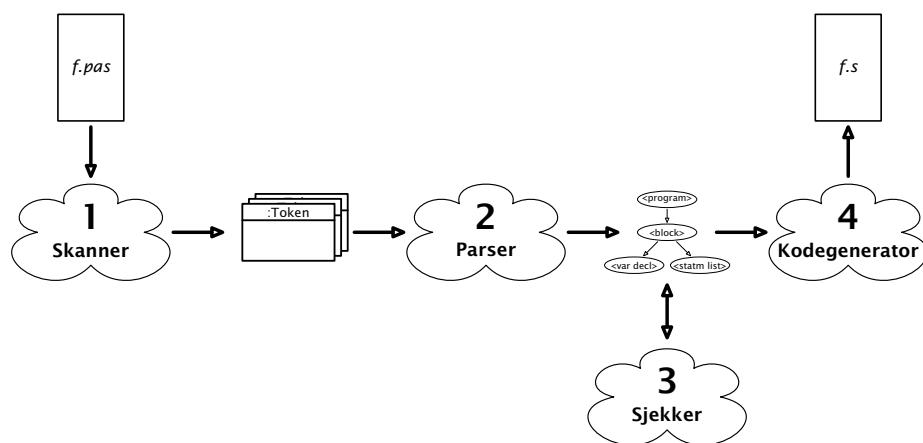
.globl xxx	Navnet xxx skal være kjent utenfor filen.
------------	---

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-2016.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-2016.zip  
$ ant
```

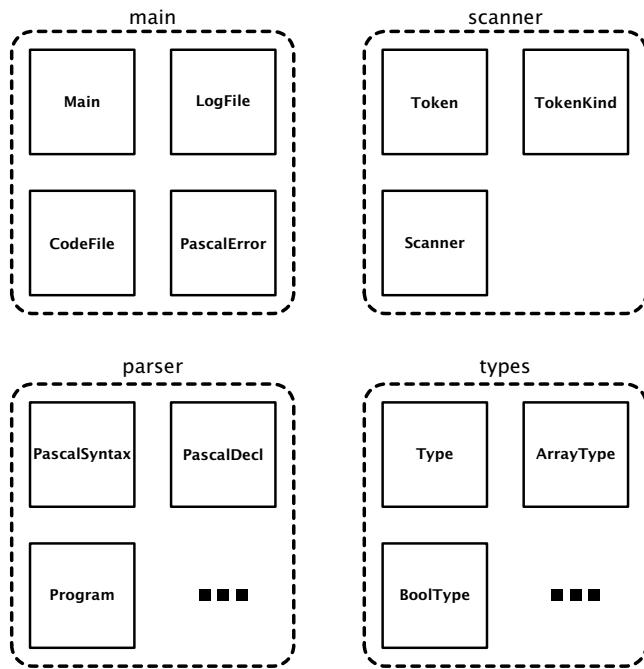
Dette vil resultere i en kjørbar fil *pascal2016.jar* som kan kjøres slik

```
$ java -jar pascal2016.jar minfil.pas
```

men den utleverte koden selvfølgelig ikke vil fungere som en ferdig kompilator! Denne 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 fire moduler, som vist i figur 4.2.



Figur 4.2: De fire modulene i kompilatoren

main innholder fire sentrale klasser som alle er ferdig programmet:

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.

scanner inneholder tre klasser som brukes av skanneren; se avsnitt 4.2 på side 38.

parser inneholder (når prosjektet er ferdig) rundt 50 klasser som brukes til å bygge parseringstreet; se avsnitt 4.3 på side 41.

types inneholder klassen **Type** og noen subklasser av den. Objekter av disse klassene representerer datatypene i programmet og brukes under sjekkingen.

4.1.3 Selvidentifikasjon

Når man arbeider med objektorientert programering, 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 39.⁶

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
<code>-logB</code>	Del 3	Hvordan navnene bindes
<code>-logP</code>	Del 2	Hvilke parseringsmetoder som kalles
<code>-logS</code>	Del 1	Hvilke symboler som leses av skanneren
<code>-logT</code>	Del 3	Typesjekkingen
<code>-logY</code>	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 Pascal2016-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 eller en raritet. Kompilatoren din bør håndtere disse programmene på samme måte som referanse-kompilatoren.

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>.

⁶ 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*.

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**.

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 tegnkoder 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 tegnkodene.

4.2 Del I: Skanneren

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

```

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

```

mini.pas

Figur 4.3: Et minimalt Pascal2016-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.

Legg også merke til at alle navn og reserverte ord (**program**, **Mini**, **write** og **end**) er omformet til *små bokstaver*. Dette gjøres fordi Pascal2016 ikke ser forskjell på store og små bokstaver.

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.)

4.2.1 Representasjon av symboler

Hvert symbol i Pascal2016-programmet lagres i en instans av klassen Token vist i figur 4.4.

Token.java

```

4
5  public class Token {
6      public TokenKind kind;
7      public String id;
8      public char charVal;
9      public int intValue, lineNumber;
10
11     :
12
13     public String identify() {
14         String t = kind.identify();
15         if (lineNumber > 0)
16             t += " on line " + lineNumber;
17
18         switch (kind) {
19             case nameToken:   t += ":" + id; break;
20             case intValueToken: t += ":" + intValue; break;
21             case charValToken: t += ":" + charVal + "'"; break;
22         }
23         return t;
24     }
25 }
```

Figur 4.4: Klassen Token

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

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.

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 av 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.

TokenKind.java

```

5   public enum TokenKind {
6       nameToken("name"),
7       intValToken("number"),
8       charValToken("char"),
9
10      addToken("+"),
11      assignToken(":="),
12
13      :
14
15      eofToken("e-o-f");
16
17      private String image;
18
19      TokenKind(String im) {
20          image = im;
21      }
22
23
24      public String identify() {
25          return image + " token";
26      }
27
28      @Override public String toString() {
29          return image;
30      }
31
32
33
34

```

Figur 4.5: Enum-klassen TokenKind

Scanner.java

```

7   public class Scanner {
8       public Token curToken = null, nextToken = null;
9
10      private LineNumberReader sourceFile = null;
11      private String sourceFileName, sourceLine = "";
12      private int sourcePos = 0;
13
14      public Scanner(String fileName) {
15          sourceFileName = fileName;
16          try {
17              sourceFile = new LineNumberReader(new FileReader(fileName));
18          } catch (FileNotFoundException e) {
19              Main.error("Cannot read " + fileName + "!");
20          }
21
22          readNextToken();  readNextToken();
23      }
24
25
26      public String identify() {
27          return "Scanner reading " + sourceFileName;
28      }
29
30      :
31
32
33
34

```

Figur 4.6: Klassen Scanner

mini.pas

```

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

(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 og figur 4.15 på side 55. Når kompilatoren vår kjøres med opsjonen `-testscanner`, skriver de ut logginformasjonen vist i henholdsvis figur 4.7 og figur 4.16 til 4.17 på side 55–56.

4.2.4 Mål for del I

Mål for del I

Programmet skal utvikles slik at opsjonen `-testscanner` produserer loggfiler som vist i figurene 4.7 og 4.16–4.17.

4.3 Del 2: Parsing

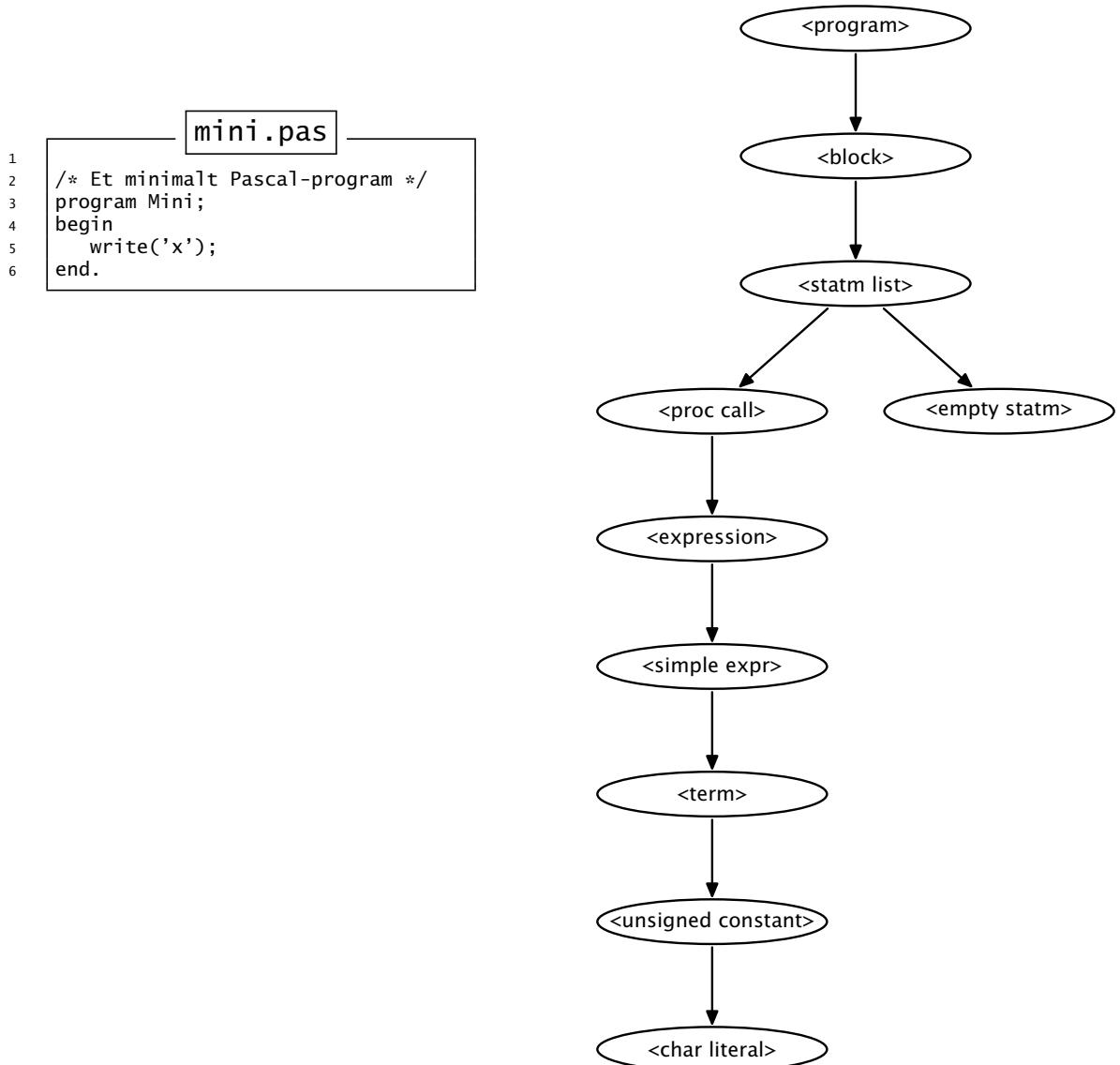
Denne delen går ut på å skrive parseren som har to oppgaver:

- 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

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

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 neste side.

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.
- <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.

WhileStatm.java

```

1 package parser;
2
3 import main.*;
4 import scanner.*;
5 import static scanner.TokenKind.*;
6
7 /* <while-statm> ::= 'while' <expression> 'do' <statement> */
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 " + lineNumber;
19     }
20
21     :
22
23     @Override void prettyPrint() {
24         Main.log.prettyPrint("while ");
25         expr.prettyPrint();
26         Main.log.prettyPrintLn(" do");
27         Main.log.prettyIndent();
28         body.prettyPrint(); Main.log.prettyOutdent();
29     }
30
31     static WhileStatm parse(Scanner s) {
32         enterParser("while-statm");
33
34         WhileStatm ws = new WhileStatm(s.curLineNum());
35         s.skip(whileToken);
36
37         ws.expr = Expression.parse(s);
38         s.skip(doToken);
39         ws.body = Statement.parse(s);
40
41         leaveParser("while-statm");
42         return ws;
43     }
44 }

```

Figur 4.9: Klassen WhileStatm

4.3.2 Parsing

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

```

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

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.)

4.3.2.1 Tvetydigheter

Grammatikken til Pascal2016 er nesten alltid entydig, men ikke alltid. I setningen

```
1   v := a
```

```
1:      1:  
2: /* Et minimalt Pascal-program */  
3: program Mini;  
4: Parser:  <program>  
5:   4: begin  
6:     5:   write('x');  
7:   Parser:  <block>  
8:   Parser:  <statm list>  
9:   Parser:    <statement>  
10:  Parser:      <proc call>  
11:  Parser:        <expression>  
12:  Parser:          <simple expr>  
13:  Parser:            <term>  
14:  Parser:              <factor>  
15:  Parser:                <unsigned constant>  
16:  Parser:                  <char literal>  
17:  Parser:                    </char literal>  
18:  Parser:                  </unsigned constant>  
19:  Parser:                </factor>  
20:  Parser:              </term>  
21:  Parser:            </simple expr>  
22:  Parser:          </expression>  
23:  6: end.  
24: Parser:        </proc call>  
25: Parser:      </statement>  
26: Parser:    <statement>  
27: Parser:      <empty statm>  
28: Parser:    </empty statm>  
29: Parser:  </statement>  
30: Parser: </statm list>  
31: Parser:  </block>  
32: Parser: </program>
```

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

kan a være tre forskjellige ting i henhold til syntaksen:

- 1) <unsigned constant>
- 2) <variable>
- 3) <func call> (uten paramtre)

Hva som er riktig, kan vi ikke avgjøre uten å sjekke hva a er deklarert som, og det skjer ikke før senere. Hva gjør vi så? Det enkleste er å anta at a er en variabel (som nok er det vanligste) og så ta oss av problemet senere.

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 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 slik den skal (og enda mer for å finne ut hvor langt prosessen er kommet om noe går galt), skal parse-metodene kalles på `PascalSyntax.enterParser` når de starter og så på `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 38 og figur 4.15 på side 55 vil produsere loggfilene i figur 4.10 og figurene 4.18 til 4.22 på side 57 og etterfølgende når kompilatoren kjøres med opsjonen `-logP` eller `-testparser`.

```

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

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

Dette er imidlertid ikke nok. Selv om parsering forløp feilfritt, kan det hende at parsingstreet ikke er riktig bygget opp. Den enkleste måten å sjekke dette på er å skrive ut det opprinnelige programmet basert på representasjonen i syntakstreet.⁷ Dette ordnes best ved å legge inn en metode

```

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

i hver subklasse av `PascalSyntax`.

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 4.18–4.23.

4.4 Del 3: Sjekking

Den tredje delen er å få sjekkingen 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 treeet og noen på vei tilbake. Dessuten skal del 3 beregne alle konstantene.

4.4.1 Sjekke navn ved deklarasjoner

Det må sjekkes at navn er deklarert riktig. I Pascal2016 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 navnforekomster og så finne hvilke deklarasjoner som definerer navnet.

```

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

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

⁷ En slik automatisk utskrift av et program kalles gjerne «pretty-printing» siden resultatet ofte blir penere enn en travel programmerer tar seg tid til. Denne finessen var mye vanligere i tiden før man fikk interaktive datamaskiner og gode redigeringsprogrammer.

Symbol	Betydning
\mathcal{T}_x	Typen til x
\mathcal{T}_f^F	Funksjonstypen til f
$\mathcal{T}_f^{P_i}$	Typen til f 's formelle parameter ⁸ nr i
$\{\mathcal{A}\}$	Mengden av alle array-typer
\mathcal{T}_a^E	Typen til elementene i arrayen a
\mathcal{T}_a^I	Typen til indeksen i arrayen a

Tabell 4.2: Notasjon for typesjekking

Setning	Sjekk
$v := e$	$\mathcal{T}_v = \mathcal{T}_e \wedge \mathcal{T}_v \notin \{\mathcal{A}\}$
if e then ...	$\mathcal{T}_e = Boolean$
while e do ...	$\mathcal{T}_e = Boolean$
$p(e_1, e_2, \dots)$	$\forall i : \mathcal{T}_{e_i} = \mathcal{T}_p^{P_i}$
write (e_1, e_2, \dots)	$\forall i : \mathcal{T}_{e_i} \notin \{\mathcal{A}\}$

Tabell 4.3: Typesjekking av setninger

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. Ved å angi `-logT` kan brukeren få logget alle typesjekkene som foretas; se eksempel i figur 4.25 på side 62. (Vårt minimale testprogram `mini.pas` fra figur 4.3 på side 38 er så enkelt at det ikke produserer noen typesjekklogg.)

Tabellene 4.3 og 4.4 på neste side angir hvilke typeregler som må sjekkes. De bruker en kvasimatematisk notasjon som er vist i tabell 4.2.

4.4.5 Beregne konstanter

I tillegg til sjekkingen skal del 3 beregne verdien av alle konstanter i programmet. Det er heldigvis ganske enkelt siden de er definert enten som literaler eller lik andre, tidligere definerte konstanter. Det vil si, de kan også skifte fortegn, som vist i figur 4.13 på neste side.

⁸ En **formell parameter** er en parameter i funksjons- eller prosedyredeklarasjonen, mens en **aktuell parameter** er en parameter i *kallet* på funksjonen eller prosedyren.

(Del-)uttrykk	Sjekk	Resultat
$+ e$	$\mathcal{T}_e = \text{Integer}$	Integer
$- e$	$\mathcal{T}_e = \text{Integer}$	Integer
$e_1 + e_2$	$\mathcal{T}_{e_1} = \text{Integer} \wedge \mathcal{T}_{e_2} = \text{Integer}$	Integer
$\text{not } e$	$\mathcal{T}_e = \text{Boolean}$	Boolean
$e_1 \text{ and } e_2$	$\mathcal{T}_{e_1} = \text{Boolean} \wedge \mathcal{T}_{e_2} = \text{Boolean}$	Boolean
$e_1 \text{ or } e_2$	$\mathcal{T}_{e_1} = \text{Boolean} \wedge \mathcal{T}_{e_2} = \text{Boolean}$	Boolean
$e_1 = e_2$	$\mathcal{T}_{e_1} = \mathcal{T}_{e_2} \wedge \mathcal{T}_{e_1} \notin \{\mathcal{A}\}$	Boolean
$f(e_1, e_2, \dots)$	$\forall i : \mathcal{T}_{e_i} = \mathcal{T}_f^{P_i}$	\mathcal{T}_f^F
$a[e]$	$a \in \{\mathcal{A}\} \wedge \mathcal{T}_e = \mathcal{T}_a^I$	\mathcal{T}_a^E

Tabell 4.4: Typesjekking av uttrykk

```

1 program Konstanter;
2 const a = 25;
3   b = -a;  c = +a;
4 begin
5   write('a', '=', a, ', ');
6   write('b', '=', b, ', ');
7   write('c', '=', c, EoL);
8 end.

```

Figur 4.13: Et program med konstanter; komplilatoren skal i del 3 beregne at a er 25, b er -25 og c er 25

Mål for del 3

Kompilatoren skal foreta navnebindinger og sjekke typer, og når den kjøres med opsjonen `-testchecker` produsere data om dette som vist i figur 4.12 og 4.24.

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 Registrer

Vi vil bruke disse registrene:

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

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

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

%ESP peker på toppen av kjørestakken.

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

```

1 # Code file created by Pascal2016 compiler 2016-07-29 10:48:08
2 .globl main
3 main:
4     call  prog$mini_1          # Start program
5     movl $0,%eax             # Set status 0 and
6     ret                      # terminate the program
7 prog$mini_1:
8     enter $32,$1              # Start of mini
9     movl $120,%eax            #
10    pushl %eax               # Push next param.
11    call  write_char          #
12    addl $4,%esp              # Pop param.
13    leave                     # End of mini
14    ret

```

Figur 4.14: Kodefil laget fra `mini.pas`

4.5.2.1 Navn

Hovedprogrammet, funksjoner og prosedyrer beholder sitt Pascal2016-navn men med en endelse så vi unngår dobbeltdeklarasjoner: `proc$name_nnn`.

Eksterne navn benyttes ved kall på biblioteksprosedyrer; navnet på startpunktet, `dvsmain`, er også et eksternt navn. Slike navn skrives som de er i Linux, mens de trenger en understrekning (`«_»`) foran navnet i Windows and Mac OS X.

Parametre trenger ikke navn i assemblerkoden siden de er gitt utfra posisjonen i parameterlisten: Første parameter har tillegg (`«offset»`) 8, andre parameter 12, tredje parameter 16 etc.

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`. Dette gjør kodegenereringen svært mye enklere, men vi får ikke alltid den mest optimale koden.⁹

4.5.3.1 Operander i uttrykk

I tabell 4.5 på neste side er vist hvilken kode som må genereres for å hente en verdi `<n>`, en enkel variabel `<v(b)o>` (der `b` er blokknivået og `o` er variabelens «offset»), et arrayelement `<a(b)o>[<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 `low`; mao, hvis arrayen er deklarert som `array[10..20] of xxx`, må vi trekke fra 10 ved hvert oppslag.¹⁰

⁹ Optimalisering av kodegenerering er et helt eget fagfelt som vi ikke har tid til å se på i INF2100.

¹⁰ Hvis nedre grense er 0, kan vi droppe denne instruksjonen.

Boolean-verdier blir representert av heltall, nærmere bestemt:

$$\text{false} = 0, \quad \text{true} = 1$$

(Kode for funksjonskall er ikke tatt med her – dette er beskrevet i avsnitt [4.11 på side 51.](#))

$\langle n \rangle$	\Rightarrow	<code>movl \$n,%eax</code>
$\langle v^{(b)o} \rangle$	\Rightarrow	<code>movl -4b(%ebp),%edx</code> <code>movl o(%edx),%eax</code>
$\langle a^{(b)o} \rangle [\langle e \rangle]$	\Rightarrow	<code>(Beregn $\langle e \rangle$ med svar i %EAX)</code> <code>subl \$low,%eax</code> (Dropp om $low=0$) <code>movl -4b(%ebp),%edx</code> <code>leal o(%edx),%edx</code> <code>movl (%edx,%eax,4),%eax</code>
$\langle e \rangle$	\Rightarrow	<code>(Beregn $\langle e \rangle$ med svar i %EAX)</code>

Tabell 4.5: 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 4.6 viser hvordan vi skal oversette de unære operatorene.

$+ \langle e \rangle$	\Rightarrow	<code>(Beregn $\langle e \rangle$ med svar i %EAX)</code>
$- \langle e \rangle$	\Rightarrow	<code>(Beregn $\langle e \rangle$ med svar i %EAX)</code> <code>negl %eax</code>
$\text{not } \langle e \rangle$	\Rightarrow	<code>(Beregn $\langle e \rangle$ med svar i %EAX)</code> <code>xorl \$1,%eax</code>

Tabell 4.6: Kode generert av unære operatorer i uttrykk

Binære operatorer I tabell [4.7 på neste side](#) 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.

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

Tabell 4.7: 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 4.8.

	⇒	
--	---	--

Tabell 4.8: Kode generert av tom setning

4.5.4.2 Oversettelse av sammensatte setninger

En sammensatt setning er ganske så enkel å oversette; se tabell 4.9.

<code>begin $\langle S_1 \rangle$; $\langle S_2 \rangle$; ... end</code>	⇒	$\langle S_1 \rangle$ $\langle S_2 \rangle$ \vdots
--	---	--

Tabell 4.9: Kode generert av sammensatt setning

4.5.4.3 Oversettelse av tilordningssetninger

Kodegenerering for slike setninger er vist i tabell 4.10 på neste side. Husk at venstresiden kan være enten en vanlig variabel $\langle v^{(b)o} \rangle$, et arrayelement $\langle a^{(b)o} \rangle[\langle e \rangle]$ eller et funksjonsnavn $\langle f \rangle$.

$\langle v^{(b)o} \rangle := \langle e \rangle ;$	⇒	(Beregn $\langle e \rangle$ med svar i %EAX) movl -4b(%ebp),%edx movl %eax,o(%edx)
$\langle a^{(b)o} \rangle [\langle e_1 \rangle] := \langle e_2 \rangle$	⇒	(Beregn $\langle e_2 \rangle$ med svar i %EAX) pushl %eax (Beregn $\langle e_1 \rangle$ med svar i %EAX) subl \$low,%eax (Dropp om low=0) movl -4b(%ebp),%edx leal o(%edx),%edx popl %ecx movl %ecx,(%edx,%eax,4)
$\langle f^{(b)} \rangle := \langle e \rangle ;$	⇒	(Beregn $\langle e \rangle$ med svar i %EAX) movl -4(b+1)(%ebp),%edx movl %eax,-32(%edx)

Tabell 4.10: Kode generert av tilordning

4.5.4.4 Oversettelse av kallsetninger

Kallsetninger og funksjonskall oversettes på akkurat samme måte, nemlig til kodesekvensen vist i tabell 4.11.

- 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 parametere, inkludert 0.

$f(\langle e_1 \rangle, \langle e_2 \rangle)$	⇒	(Beregn $\langle e_2 \rangle$ med svar i %EAX) pushl %eax (Beregn $\langle e_1 \rangle$ med svar i %EAX) pushl %eax call proc\$f_n addl \$8,%esp
---	---	---

Tabell 4.11: Kode generert av prosedyrekall

Kall på write Prosedyren `write` er som nevnt spesiell: den kan ha vilkårlig mange parametere, og de kan være av enhver type (unntatt arrayer). Hver parameter oversettes til et kall på en egen biblioteksfunksjon slik det er vist i tabell 4.12 på neste side.

write(<int-e>)	⇒	<Beregner <int-e> med svar i %EAX> pushl %eax call write_int addl \$4,%esp
write(<char-e>)	⇒	<Beregner <char-e> med svar i %EAX> pushl %eax call write_char addl \$4,%esp
write(<bool-e>)	⇒	<Beregner <bool-e> med svar i %EAX> pushl %eax call write_bool addl \$4,%esp

Tabell 4.12: Kode generert av kall på write

4.5.4.5 Oversettelse av if-setninger

Tabell 4.13 viser oversettelse av en if-setning, både uten og med en elsegren.

if <e> then <S>	⇒	<Beregner <e> med svar i %EAX> cmpl \$0,%eax je <lab> <S> <lab>:
if <e> then <S ₁ > else <S ₂ >	⇒	<Beregner <e> med svar i %EAX> cmpl \$0,%eax je <lab ₁ > <S ₁ > jmp <lab ₂ > <lab ₁ >: <S ₂ > <lab ₂ >:

Tabell 4.13: 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 4.14 på neste side.

4.5.5 Oversettelse av funksjoner og prosedyrer

Som vist i figur 4.15 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.

<pre>while <e> do <S></pre>	⇒	<pre>(lab₁): (Beregn <e> med svar i %EAX) cmpl \$0,%eax je <lab₂> <S> jmp <lab₁> <lab₂>:</pre>
---	---	---

Tabell 4.14: Kode generert av while-setning

- 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 angi 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**.

<pre>function <f> (...): <type>; <D> begin <S> end</pre>	⇒	<pre>func\$(<f>)_n: enter \$(32+ant byte i <D>),\${blokknivå} <S> movl -32(%ebp),%eax leave ret</pre>
<pre>procedure <p> (...); <D> begin <S> end</pre>	⇒	<pre>proc\$(<p>)_n: enter \$(32+ant byte i <D>),\${blokknivå} <S> leave ret</pre>

Tabell 4.15: Kode generert av funksjons- og prosedyrededeclarasjon

4.5.5.1 Oversettelse av hovedprogrammet

Det enkleste er å late som om hovedprogrammet er en prosedyre som kalles av en minimal **main**,¹¹ se tabell 4.16 på neste side.

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 4.15).

4.5.6.2 Deklarasjon av parametere

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

¹¹ Det er et krav at startpunktet heter **main** i Linux/Unix og **_main** i Windows og Mac OS X.

```
program xx;  
(D)  
begin  
(S)  
end.
```

⇒

```
.globl main  
main:  
    call    prog$xx_n  
    movl    $0,%eax  
    ret  
prog$xx_n:  
    enter  $32+ant byte i (D) , $1  
    (S)  
    leave  
    ret
```

Tabell 4.16: Kode generert av hovedprogrammet

Mål for del 4

Kompilatoren skal generere kode som lar seg assemblere på Ifis Linux-maskiner og som utfører det kompilerte programmet korrekt.

4.6 Et litt større eksempel

```

1 program GCD;
2 /* A program to compute the {greatest common} of two numbers,
3    i.e., the biggest number by which the two original
4    numbers can be divided 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('G', 'C', 'D', '(', v1, ',', v2, ')', '=', res, eol);
21 end.

```

Figur 4.15: Et litt større Pascal2016-program gcd.pas

```

1: program GCD;
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:      i.e., the biggest number by which the two original
7:      numbers can be divided 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

```

Figur 4.16: Loggfil som demonstrerer hvilke symboler skanneren finner i gcd.pas (del 1)

```

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
51   Scanner: name token on line 13: gcd
52   Scanner: := token on line 13
53   Scanner: name token on line 13: m
54     14: else
55   Scanner: else token on line 14
56     15:      GCD := GCD(n, m mod n)
57   Scanner: name token on line 15: gcd
58   Scanner: := token on line 15
59   Scanner: name token on line 15: gcd
60   Scanner: ( token on line 15
61   Scanner: name token on line 15: n
62   Scanner: , token on line 15
63   Scanner: name token on line 15: m
64   Scanner: mod token on line 15
65   Scanner: name token on line 15: n
66   Scanner: ) token on line 15
67     16: end; { GCD }
68   Scanner: end token on line 16
69   Scanner: ; token on line 16
70     17:
71       18: begin
72   Scanner: begin token on line 18
73     19:      res := GCD(v1,v2);
74   Scanner: name token on line 19: res
75   Scanner: := token on line 19
76   Scanner: name token on line 19: gcd
77   Scanner: ( token on line 19
78   Scanner: name token on line 19: v1
79   Scanner: , token on line 19
80   Scanner: name token on line 19: v2
81   Scanner: ) token on line 19
82   Scanner: ; token on line 19
83     20:      write('G', 'C', 'D', '(', v1, ',', v2, ')', '=', res, eol);
84   Scanner: name token on line 20: write
85   Scanner: ( token on line 20
86   Scanner: char token on line 20: 'G'
87   Scanner: , token on line 20
88   Scanner: char token on line 20: 'C'
89   Scanner: , token on line 20
90   Scanner: char token on line 20: 'D'
91   Scanner: , token on line 20
92   Scanner: char token on line 20: '('
93   Scanner: , token on line 20
94   Scanner: name token on line 20: v1
95   Scanner: , token on line 20
96   Scanner: char token on line 20: ','
97   Scanner: , token on line 20
98   Scanner: name token on line 20: v2
99   Scanner: , token on line 20
100  Scanner: char token on line 20: ')'
101  Scanner: , token on line 20
102  Scanner: char token on line 20: '='
103  Scanner: , token on line 20
104  Scanner: name token on line 20: res
105  Scanner: , token on line 20
106  Scanner: name token on line 20: eol
107  Scanner: ) token on line 20
108  Scanner: ; token on line 20
109    21: end.
110  Scanner: end token on line 21
111  Scanner: . token on line 21
112  Scanner: e-o-f token

```

Figur 4.17: Logfil som demonstrerer hvilke symboler skanneren finner i gcd.pas (del 2)

```

1: program GCD;
2: Parser:   <program>
3:   2: /* A program to compute the {greatest common} of two numbers,
4:      i.e., the biggest number by which the two original
5:      numbers can be divided 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:   <unsigned constant>
13: Parser:   <number literal>
14: Parser:   </number literal>
15: Parser:   </unsigned constant>
16: Parser:   </constant>
17: Parser:   </const decl>
18: Parser:   <const decl>
19: Parser:   <constant>
20: Parser:   <unsigned constant>
21: Parser:   <number literal>
22:   7:
23:   8: var res: integer;
24: Parser:   </number literal>
25: Parser:   </unsigned constant>
26: Parser:   </constant>
27: Parser:   </const decl>
28: Parser:   </const decl part>
29: Parser:   <var decl part>
30: Parser:   <var decl>
31: Parser:   <type>
32: Parser:   <type name>
33:   9:
34:   10: function GCD (m: integer; n: integer): integer;
35: Parser:   </type name>
36: Parser:   </type>
37: Parser:   </var decl>
38: Parser:   </var decl part>
39: Parser:   <func decl>
40: Parser:   <param decl list>
41: Parser:   <param decl>
42: Parser:   <type name>
43: Parser:   </type name>
44: Parser:   </param decl>
45: Parser:   <param decl>
46: Parser:   <type name>
47: Parser:   </type name>
48: Parser:   </param decl>
49: Parser:   </param decl list>
50: Parser:   <type name>
51:   11: begin
52: Parser:   </type name>
53:   12: if n = 0 then
54: Parser:   <block>
55: Parser:   <stmt list>
56: Parser:   <statement>
57: Parser:   <if-stmt>
58: Parser:   <expression>
59: Parser:   <simple expr>
60: Parser:   <term>
61: Parser:   <factor>
62: Parser:   <variable>
63: Parser:   </variable>
64: Parser:   </factor>
65: Parser:   </term>
66: Parser:   </simple expr>
67: Parser:   <rel opr>
68: Parser:   </rel opr>
69: Parser:   <simple expr>
70: Parser:   <term>
71: Parser:   <factor>
72: Parser:   <unsigned constant>
73: Parser:   <number literal>
74:   13:   GCD := m
75: Parser:   </number literal>

```

Figur 4.18: Logfil som viser parsing av gcd.pas (del I)

```

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

```

Figur 4.19: Loggfil som viser parsering av gcd.pas (del 2)

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

Figur 4.20: Loggfil som viser parsing av gcd.pas (del 3)

```
226 Parser:          <expression>
227 Parser:            <simple expr>
228 Parser:              <term>
229 Parser:                <factor>
230 Parser:                  <unsigned constant>
231 Parser:                    <char literal>
232 Parser:                      </char literal>
233 Parser:                    </unsigned constant>
234 Parser:                </factor>
235 Parser:              </term>
236 Parser:            </simple expr>
237 Parser:          </expression>
238 Parser:          <expression>
239 Parser:            <simple expr>
240 Parser:              <term>
241 Parser:                <factor>
242 Parser:                  <variable>
243 Parser:                      </variable>
244 Parser:                    </factor>
245 Parser:              </term>
246 Parser:            </simple expr>
247 Parser:          </expression>
248 Parser:          <expression>
249 Parser:            <simple expr>
250 Parser:              <term>
251 Parser:                <factor>
252 Parser:                  <unsigned constant>
253 Parser:                    <char literal>
254 Parser:                      </char literal>
255 Parser:                    </unsigned constant>
256 Parser:                </factor>
257 Parser:              </term>
258 Parser:            </simple expr>
259 Parser:          </expression>
260 Parser:          <expression>
261 Parser:            <simple expr>
262 Parser:              <term>
263 Parser:                <factor>
264 Parser:                  <variable>
265 Parser:                      </variable>
266 Parser:                    </factor>
267 Parser:              </term>
268 Parser:            </simple expr>
269 Parser:          </expression>
270 Parser:          <expression>
271 Parser:            <simple expr>
272 Parser:              <term>
273 Parser:                <factor>
274 Parser:                  <unsigned constant>
275 Parser:                    <char literal>
276 Parser:                      </char literal>
277 Parser:                    </unsigned constant>
278 Parser:                </factor>
279 Parser:              </term>
280 Parser:            </simple expr>
281 Parser:          </expression>
282 Parser:          <expression>
283 Parser:            <simple expr>
284 Parser:              <term>
285 Parser:                <factor>
286 Parser:                  <unsigned constant>
287 Parser:                    <char literal>
288 Parser:                      </char literal>
289 Parser:                    </unsigned constant>
290 Parser:                </factor>
291 Parser:              </term>
292 Parser:            </simple expr>
293 Parser:          </expression>
294 Parser:          <expression>
295 Parser:            <simple expr>
296 Parser:              <term>
297 Parser:                <factor>
298 Parser:                  <variable>
299 Parser:                      </variable>
300 Parser:                    </factor>
```

Figur 4.21: Loggfil som viser parsering av gcd.pas (del 4)

```

301 Parser:           </term>
302 Parser:           </simple expr>
303 Parser:           </expression>
304 Parser:           <expression>
305 Parser:           <simple expr>
306 Parser:           <term>
307 Parser:           <factor>
308 Parser:           <variable>
309 Parser:           </variable>
310 Parser:           </factor>
311 Parser:           </term>
312 Parser:           </simple expr>
313 Parser:           </expression>
314   21: end.
315 Parser:           </proc call>
316 Parser:           </statement>
317 Parser:           <statement>
318 Parser:           <empty statm>
319 Parser:           </empty statm>
320 Parser:           </statement>
321 Parser:           </statm list>
322 Parser:           </block>
323 Parser:           </program>

```

Figur 4.22: Logfil som viser parsing av gcd.pas (del 5)

```

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   writeln('G', 'C', 'D', '(', v1, ',', v2, ')', '=', res, eol);
19 end.

```

Figur 4.23: Logfil med «skjønnskrift» av gcd.pas

```

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

```

Figur 4.24: Logfil med navnebinding for gcd.pas

```

1 Type check = operands on line 12: type Integer vs type Integer
2 Type check if-test on line 12: type Boolean vs type Boolean
3 Type check := on line 13: type Integer vs type Integer
4 Type check param #1 on line 15: type Integer vs type Integer
5 Type check left mod operand on line 15: type Integer vs type Integer
6 Type check right mod operand on line 15: type Integer vs type Integer
7 Type check param #2 on line 15: type Integer vs type Integer
8 Type check := on line 15: type Integer vs type Integer
9 Type check param #1 on line 19: type Integer vs type Integer
10 Type check param #2 on line 19: type Integer vs type Integer
11 Type check := on line 19: type Integer vs type Integer

```

Figur 4.25: Logfil med typesjekking for gcd.pas

```

1 # Code file created by Pascal2016 compiler 2016-07-29 11:47:15
2 .globl main
3 main:
4     call prog$gcd_1          # Start program
5     movl $0,%eax            # Set status 0 and
6     ret                      # terminate the program
7 func$gcd_2:
8     enter $32,$2            # Start of gcd
9                         # Start if-statement
10    movl -8(%ebp),%edx
11    movl 12(%edx),%eax      #   n
12    pushl %eax
13    movl $0,%eax            #   0
14    popl %ecx
15    cmpl %eax,%ecx
16    movl $0,%eax
17    sete %al                # Test =
18    cmpl $0,%eax
19    je .L0003
20    movl -8(%ebp),%edx
21    movl 8(%edx),%eax       #   m
22    movl -8(%ebp),%edx
23    movl %eax,-32(%edx)     # gcd :=
24    jmp .L0004
25 .L0003:
26    movl -8(%ebp),%edx
27    movl 8(%edx),%eax       #   m
28    pushl %eax
29    movl -8(%ebp),%edx
30    movl 12(%edx),%eax      #   n
31    movl %eax,%ecx
32    popl %eax
33    cdq
34    idivl %ecx,%eax        #   mod
35

```

Figur 4.26: Kodefil produsert fra gcd.pas (del I)

```

36      pushl %eax          # Push param #2
37      movl -8(%ebp),%edx
38      movl 12(%edx),%eax   # n
39      pushl %eax          # Push param #1
40      call func$gcd_2
41      addl $8,%esp         # Pop parameters
42      movl -8(%ebp),%edx
43      movl %eax,-32(%edx)  # gcd :=
44 .L0004:
45      movl -32(%ebp),%eax
46      leave
47      ret
48 prog$gcd_1:
49      enter $36,$1          # Start of gcd
50      movl $462,%eax        # 462
51      pushl %eax          # Push param #2
52      movl $1071,%eax       # 1071
53      pushl %eax          # Push param #1
54      call func$gcd_2
55      addl $8,%esp         # Pop parameters
56      movl -4(%ebp),%edx
57      movl %eax,-36(%edx)  # res :=
58      movl $71,%eax         # 'G'
59      pushl %eax          # Push next param.
60      call write_char
61      addl $4,%esp         # Pop param.
62      movl $67,%eax         # 'C'
63      pushl %eax          # Push next param.
64      call write_char
65      addl $4,%esp         # Pop param.
66      movl $68,%eax         # 'D'
67      pushl %eax          # Push next param.
68      call write_char
69      addl $4,%esp         # Pop param.
70      movl $40,%eax         # '('
71      pushl %eax          # Push next param.
72      call write_char
73      addl $4,%esp         # Pop param.
74      movl $1071,%eax       # 1071
75      pushl %eax          # Push next param.
76      call write_int
77      addl $4,%esp         # Pop param.
78      movl $44,%eax         # ','
79      pushl %eax          # Push next param.
80      call write_char
81      addl $4,%esp         # Pop param.
82      movl $462,%eax        # 462
83      pushl %eax          # Push next param.
84      call write_int
85      addl $4,%esp         # Pop param.
86      movl $41,%eax         # ')'
87      pushl %eax          # Push next param.
88      call write_char
89      addl $4,%esp         # Pop param.
90      movl $61,%eax         # '='
91      pushl %eax          # Push next param.
92      call write_char
93      addl $4,%esp         # Pop param.
94      movl -4(%ebp),%edx
95      movl -36(%edx),%eax  # res
96      pushl %eax          # Push next param.
97      call write_int
98      addl $4,%esp         # Pop param.
99      movl $10,%eax         # 10
100     pushl %eax          # Push next param.
101     call write_char
102     addl $4,%esp         # Pop param.
103     leave
104     ret                  # End of gcd
105

```

Figur 4.27: Kodefil produsert fra gcd.pas (del 2)

Kapittel 5

Kompilering av blokkorienterte språk

Blokkorienterte språk som Pascal krever litt ekstra omtanke når man skal generere kode for dem. I INF2100 trenger man ikke å vite så mye om dette siden det er vist en oppskrift for hva man skal gjøre, men det er alltid noen som gjerne vil vite nøyaktige hva som skjer. Dette kapitlet er for dem.

5.1 Bakgrunn

Når man skal kompilere et blokkorientert språk som Pascal, der man kan deklarere prosedyrer inni prosedyrer inni prosedyrer så dypt man vil, gir dette et par utfordringer under kompileringen:

- Variabler i en blokk må opprettes på stakken når den tilhørende programmet/funksjonen/prosedyren kalles og fjernes når den er ferdig.
- Det må være mulig å aksessere variabler ikke bare i den lokale blokken men også alle variabler i globale blokker som er synlige.

5.1.1 Kontekstvektor

Én av flere løsninger på problemet er å opprette en såkalt **kontekstvektor**, dvs en tabell over hvor alle de synlige globale blokkene befinner seg på stakken. På den måten får man enkelt tilgang til dem alle.

Noen implementasjoner har bare én kontekstvektor mens andre velger å ha én for hver aktiv blokk. Denne siste løsningen er valgt i INF2100-prosjektet siden prosessoren vår x86 har to instruksjoner som gjør dette usedvanlig enkelt: `enter` og `leave`.

5.1.2 Et eksempel

Som eksempel skal vi bruke programmet vist i figur 5.1 på neste side; det inneholder en funksjon inni en prosedyre inni hovedprogrammet. Den koden som referansekomplilatoren genererer, er vist i figur 5.5 på side 70.

```

1 program Blokker;
2 var V1A: Integer;  V1B: Integer;
3
4     procedure P1 (A1A : Integer;  A1B: Integer);
5     var V2: Integer;
6
7         function F2 (A2: Integer): Integer;
8             var V3 : Integer;
9             begin
10                 V3 := A2+1;  F2 := V3
11             end; { F2 }
12
13         begin
14             V2 := F2(A1A);
15             V1A := V2*A1B
16         end; { P1 }
17
18     begin
19         P1(-3, 7);
20         Write(V1A, EoL)
21     end.

```

Figur 5.1: En enkelt testprogram

5.2 Start av hovedprogrammet

I Pascal er det enklest å behandle hovedprogrammet på samme måte som funksjoner og prosedyrer. Følgende skjer da:

- 1) Hovedprogrammet kalles med en `call`-instruksjon som legger returnadressen (dvs adressen til instruksjonen etter `call`-instruksjonen) på stakken.
- 2) Den første instruksjonen i hovedprogrammet er `enter` som gjør flere ting:
 - (a) Innholdet i `%EBP`-registeret gjemmes unna på stakken.
 - (b) Det settes av plass til kontekstvektoren (28 byte),¹² returverdien (4 byte)¹³ og 2 lokale variabler ($2 \times 4 = 8$ byte); tilsammen 40 byte.
 - (c) Kontekstvektoren fra blokken utenfor kopieres inn, men siden hovedprogrammet er på blokknivå 1, er det ingen yttere blokk.
 - (d) Kontekstvektoren utvides med en peker til denne blokken.

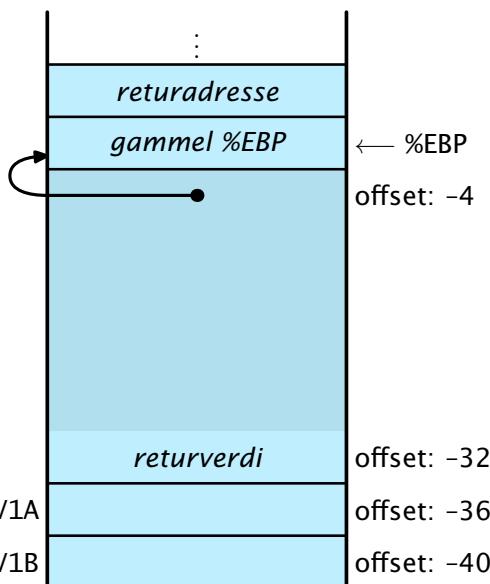
Vi får da situasjonen vist i figur 5.2 på neste side. Kontekstvektoren er markert med litt mørkere farge.

5.3 Start av en prosedyre

Når hovedprogrammet kaller prosedyren `P1`, skjer akkurat det samme, bortsett fra at parametrene legges på stakken før kallet skjer.

¹² Siden vi setter av 28 byte til kontekstvektoren, kan vi ikke ha indre blokker dypere enn 7 nivåer, men det er nok for alle praktiske formål. (Vi kunne ha valgt å sette av et antall byte avhengig av blokknivået, men det ville gitt mer komplisert kode, så i INF2100 har jeg valgt å sette av et fast antall.)

¹³ Selv om vi bare trenger å lagre en returverdi i funksjoner, setter vi av plassen også i hovedprogrammet og i prosedyrer; det blir enklere kode da.

**Figur 5.2:** Stakken når hovedprogrammet starter

Prosedyren er på blokknivå 2, så kontekstvektor fra blokken utenfor (hovedprogrammet på blokknivå 1) kopieres inn i vår nye kontekstvektor før den utvides med en peker til den lokale blokken (vår egen).

Etter `enter`-instruksjonen ser stakken ut som vist i figur 5.3 på neste side.

5.4 Start av en indre funksjon

Prosedyren P1 kaller funksjonen F2, og igjen skjer det samme. Figur 5.4 på side 69 viser situasjonen etter at `enter`-instruksjonen i F2 er ferdig.

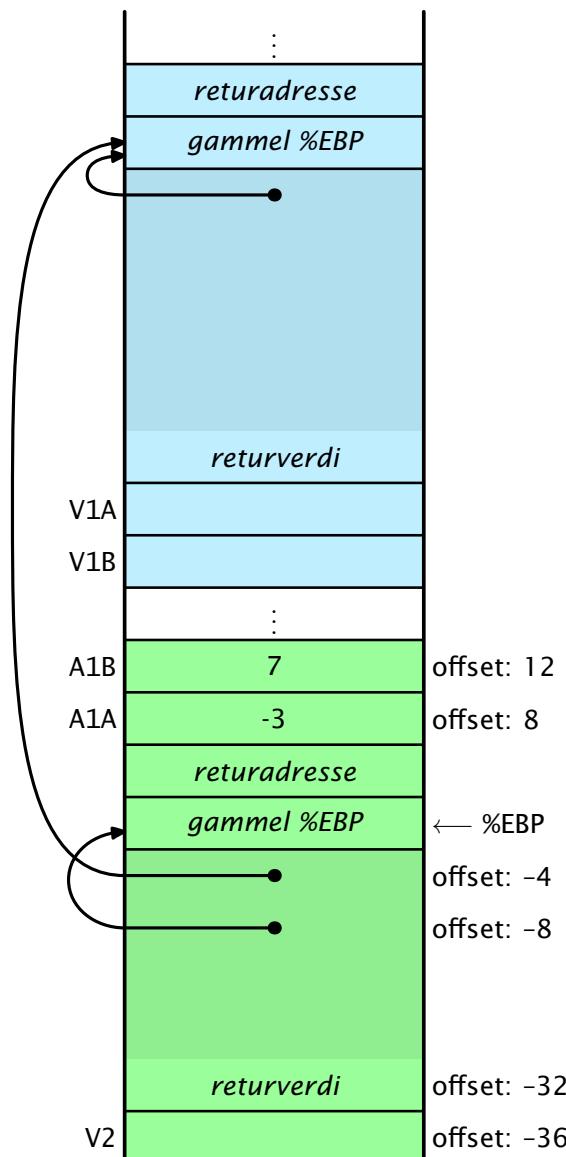
Vi ser nå at vi kan få tak i alle synlige variabler ved å gå via kontekstvektoren. For eksempel får vi tak i den lokale V3 som ligger på blokknivå 3 ved først å gjøre følgende:

- 1) Slå opp på element nr 3 i kontekstvektoren (og dette har offset $4 \times -3 = -12$).
- 2) Nå har vi adressen til riktig blokk, og der finner vi variablene V3 med offset -36.

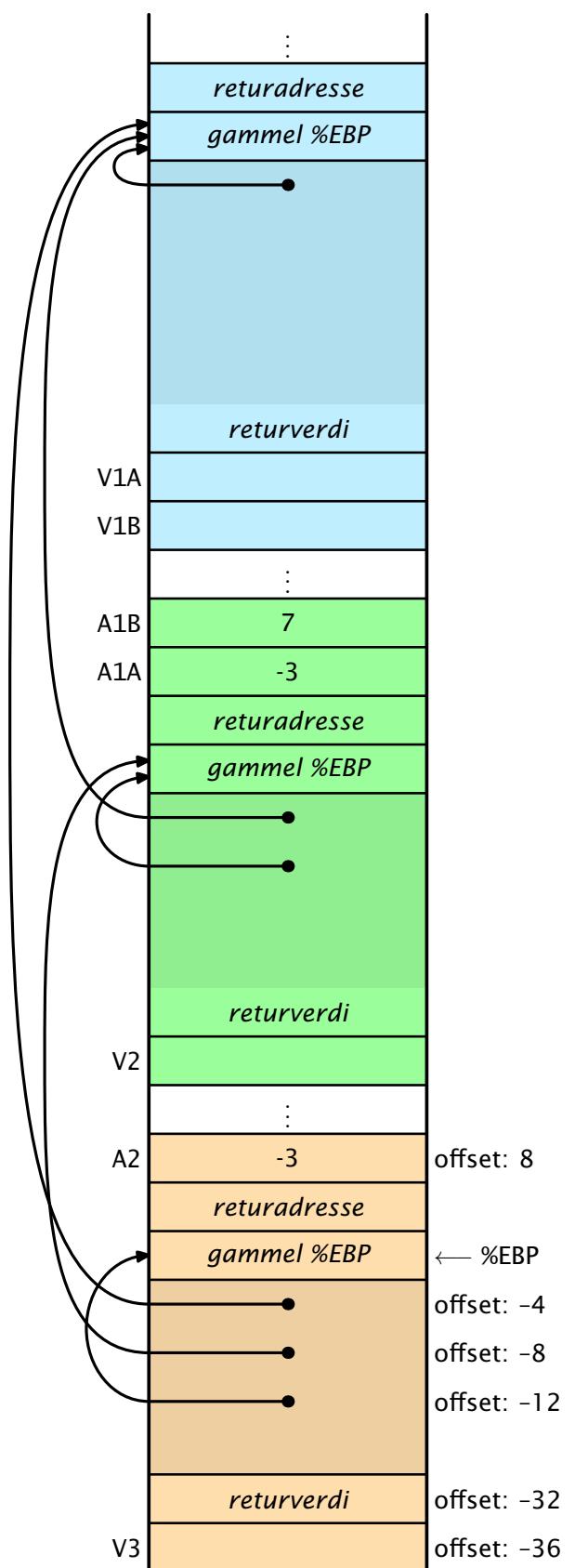
1	<code>movl -12(%ebp),%edx</code>			
2	<code>movl -36(%edx),%eax</code>	#	v3	

På samme måte kan vi få tak i den globale variablene V1B som ligger på blokknivå 1 og har offset -40:

1	<code>movl -4(%ebp),%edx</code>			
2	<code>movl -40(%edx),%eax</code>	#	v1b	



Figur 5.3: Stakken når prosedyren har startet



Figur 5.4: Stakken når den indre funksjonen har startet

```

1 # Code file created by Pascal2016 compiler 2016-05-07 11:02:35
2 .globl main
3 main:
4     call  prog$blokker_1          # Start program
5     movl $0,%eax
6     ret
7 func$f2_3:
8     enter $36,$3
9     movl -12(%ebp),%edx
10    movl 8(%edx),%eax
11    pushl %eax
12    movl $1,%eax
13    movl %eax,%ecx
14    popl %eax
15    addl %ecx,%eax
16    movl -12(%ebp),%edx
17    movl %eax,-36(%edx)
18    movl -12(%ebp),%edx
19    movl -36(%edx),%eax
20    movl %eax,-32(%ebp)
21    movl -32(%ebp),%eax
22    leave
23    ret
24 proc$p1_2:
25     enter $36,$2
26     movl -8(%ebp),%edx
27     movl 8(%edx),%eax
28     pushl %eax
29     call  func$f2_3
30     addl $4,%esp
31     movl -8(%ebp),%edx
32     movl %eax,-36(%edx)
33     movl -8(%ebp),%edx
34     movl -36(%edx),%eax
35     pushl %eax
36     movl -8(%ebp),%edx
37     movl 12(%edx),%eax
38     movl %eax,%ecx
39     popl %eax
40     imull %ecx,%eax
41     movl -4(%ebp),%edx
42     movl %eax,-36(%edx)
43     leave
44     ret
45 prog$blokker_1:
46     enter $40,$1
47     movl $7,%eax
48     pushl %eax
49     movl $3,%eax
50     negl %eax
51     pushl %eax
52     call  proc$p1_2
53     addl $8,%esp
54     movl -4(%ebp),%edx
55     movl -36(%edx),%eax
56     pushl %eax
57     call  write_int
58     addl $4,%esp
59     movl $10,%eax
60     pushl %eax
61     call  write_char
62     addl $4,%esp
63     leave
64     ret

```

Figur 5.5: Assemblerkoden generert for programmet i figur 5.1 på side 66

Kapittel 6

Programmeringsstil

6.1 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.

6.1.1 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 7.1 på side 75.)
- 4) Selve klassen.

6.1.2 Variabler

Variabler bør deklarereres én og én på hver linje:

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

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

```
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 6.1: Suns forslag til hvordan setninger bør skrives

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

6.1.3 Setninger

Enkle setninger bør stå én og én på hver linje:

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

De ulike sammensatte setningene skal se ut slik figur 6.1 viser. De skal alltid ha {} rundt innmaten, og innmaten skal indenteres 4 posisjoner.

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

Tabell 6.1: Suns forslag til navnevalg i Java-programmer

6.1.4 Navn

Navn bør velges slik det er angitt i tabell 6.1.

6.1.5 Utseende

6.1.5.1 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.

6.1.5.2 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.

6.1.5.3 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 7

Dokumentasjon

7.1 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/>.

7.1.1 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.

7.1.2 Eksempel

I figur 7.1 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 7.1: Java-kode med JavaDoc-kommentarer

7.2 «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 L^AT_EX) og kan benytte alle tilgjengelige typografiske hjelpe midler som figurer, matematiske formler, fotnoter, kapittelinndeling, fontskifte og annet.

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

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

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

7.2.1 Et eksempel

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

- 1) Skriv kildefilen `bubble.w0` (vist i figur 7.2 og 7.3). Dette gjøres med en vanlig tekstbehandler som for eksempel Emacs.

- 2) Bruk programmet `weave0`¹⁴ til å lage det ferdige dokumentet som er vist i figur 7.4–7.7:

```
$ 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
```

¹⁴ Dette eksemplet bruker Dags versjon av lesbar programmering kalt `web0`; for mer informasjon, se <http://dag.atifi.uio.no/public/doc/web0.pdf>.

bubble.w0 del 1

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

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

\begin{document}
\maketitle

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

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

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

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

Figur 7.2: «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; 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 7.3: «Lesbar programmering» — kildefilen `bubble.w0 del 2`

Bubble sort

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

July 29, 2016

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

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

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

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

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

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

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

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

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

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

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

Figur 7.4: «Lesbar programmering» — utskrift side 1

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

#5 $\langle \text{swap } a[i] \text{ and } a[i+1] \rangle \equiv$
14 `temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;`
15 `++n_swaps;`
(This code is used in #3 (p.1).)

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

#4a $\langle \text{local variables } \#4 \text{ (p.1)} \rangle + \equiv$
16 `int temp, n_swaps;`

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

#6 $\langle \text{initialize} \rangle \equiv$
17 `n_swaps = 0;`
(This code is used in #3 (p.1).)

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

#7 $\langle \text{not sorted} \rangle \equiv$
18 `n_swaps > 0`
(This code is used in #2 (p.1).)

File: *bubble.w0*

page 2

Figur 7.5: «Lesbar programmering» — utskrift side 2

Variables

A

a1, 12, 14

I

i 11, 12, 13, 14

N

n1, 11

n_swaps 15, 16, 17, 18

T

temp 14, 16

VARIABLES

page 3

Figur 7.6: «Lesbar programmering» — utskrift side 3

Meta symbols

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

(Symbols marked with * are not used.)

META SYMBOLS

page 4

Figur 7.7: «Lesbar programmering» — utskrift side 4

Register

- .L0001, 48
- Aktuell parameter, 46
- ant, 37
- Assembler, 16
- Assemblerspråk, 16
- Blokknivå, 53
- Blokkstrukturert, 23
- EoL, 30
- Formell parameter, 46
- Funksjoner, 25
- gas, 37
- gcc, 37
- Høynivå programmeringsspråk, 11
- Interpreter, 14
- java, 38
- javac, 38
- JavaDoc, 75
- Kodegenerering, 47
- Kommandospråk, 14
- Kompilator, 11
- Konstant, 23
- Konstanter, 23
- Kontekstvektor, 65
- Linux, 37
- Literal, 23
- Mac OS X, 37
- Maskinspråk, 11
- Moduler, 36
- Package, 36
- Parsering, 41
- Pascal, 21
- Pascal2016, 21
- Preprocessor, 13
- Presedens, 28
- Programmeringsstil, 71
- Prosedyrer, 25
- Return, 25
- Sjekking, 45
- Skanner, 17
- Symboler, 17, 38
- Syntaks, 17
- Syntakstre, 17
- Tokens, 17, 38
- Unicode, 38
- Windows, 37

