UNIVERSITETET I OSLO
Institutt for informatikk

RusC og Rask

Kompendium til
INF2100

Stein Krogdahl

Dag Langmyhr

Høsten 2008

Innhold

Figurer

Tabeller

Forord

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

Det opprinnelige kurset ble utviklet av *Stein Krogdahl* rundt 1980 og dreide seg om å skrive en kompilator som oversatte språket *Minila* til kode for en tenkt datamaskin *Flink*; implementasjonsspråket var Simula. I 1999 gikk man over til å bruke Java som implementasjonsspråk, og i 2007 ble kurset fullstendig renovert av *Dag Langmyhr*: Minila ble erstattet av en minimal variant av C kalt *RusC* og datamaskinen Flink ble avløst av *Rask*.

Forfatterne har lagt inn en trykkfeil i kompendiet og vil gi hederlig omtale til den første som finner den. ☺

Blindern, 8. august 2008

Stein Krogdahl *Dag Langmyhr*

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

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

— Forfatterne

Kapittel 1

Innledning

1.1 Hva er kurset INF2100?

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

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

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

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

1.2 Hvorfor oppgaven er å lage en kompilator

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

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

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 bieffekten gir øket forståelse av faget selv.
- Problemområdet bør ha så mange interessante variasjoner at det kan være en god kilde til øvelsesoppgaver som kan belyse hovedproblemstillingen.

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

Det å skrive en kompilator vil også for de fleste i utgangspunktet virke som en stor og uoversiktlig oppgave. Noe av poenget med kurset er å demonstrere at med en hensiktsmessig oppdeling 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 oversette til maskinspråket til en «ekte» datamaskin vil bli en alt for omfattende oppgave. Vi skal derfor forenkle oppgaven en del, for eksempel ved å lage vårt eget lille programmeringsspråk *RusC* og ved å definere vår egen maskin *Rask* som kompilatoren skal oversette til. Vi skal i det følgende se litt nærmere på disse og andre elementer som inngår i oppgaven.

1.3 Litt om kompilatorer og liknende verktøy

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

Grunnen til at man i det hele tatt har kompilatorer er at det er høyst upraktisk å bygge datamaskiner slik at de direkte ut fra sin elektronikk kan utføre et program skrevet i et høynivå programmeringsspråk som for eksempel Java, C, C++ eller Perl. I stedet er datamaskiner bygget slik at de kan utføre et begrenset repertoar av nokså enkle instruksjoner (og det blir derved en overkommelig oppgave å lage elektronikk som kan utføre disse). Til gjengjeld kan datamaskiner raskt utføre lange sekvenser av slike instruksjoner, grovt sett med en hastighet av 1000–3000 millioner instruksjoner per sekund.

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

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

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

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

1.3.1 Preprosessorer

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

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

Et eksempel på et språk der de første kompilatorene ble laget på denne måten, er C++. C++ var i utgangspunktet en utvidelse av språket C, og utvidelsen bestod i å legge til objektorienterte begreper (klasser, subklasser og objekter) hentet fra språket Simula. Denne utvidelsen ble i første

omgang implementert ved en preprosessor som oversatte alt til ren C. I dag er imidlertid de fleste kompilatorer for C++ skrevet som selvstendige kompilatorer som oversetter direkte til maskinkode.

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

1.3.2 Interpretering

Det er også en annen måte å utføre et program skrevet i et passelig programmeringsspråk på, og den kalles **interpretering**. I stedet for å skrive en kompilator som kan oversette programmer i det aktuelle programmeringsspråket til maskinspråk, skriver man en såkalt **interpret**. 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 interpret må lese (og utføre) hver linje på nytt hver eneste gang den skal utføres for eksempel i en løkke. Interpretering går derfor generelt en del tregere under utførelsen, men man slipper å gjøre noen kompilering. En del språk er (eller var opprinnelig) siktet spesielt inn på interpretering, det gjelder for eksempel Basic og Lisp. Det finnes imidlertid nå kompilatorer også for disse språkene.

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

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

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

1.3.3 Kompilering og kjøring av Java-programmer

En av de opprinnelige ideene ved Java var knyttet til datanett ved at et program skulle kunne kompileres på én maskin for så å kunne sendes over nettet til en hvilken som helst annen maskin (for eksempel som en såkalt *applet*) og bli utført der. For å få til dette definerte man en tenkt datamaskin kalt JVM (*Java Virtual Machine*), og lot kompilatorene produsere maskinkode (gjerne kalt *byte-kode*) for denne maskinen. Det er imidlertid ingen datamaskin som har elektronikk for direkte å utføre

slik byte-kode, og maskinen der programmet skal utføres må derfor ha et program som simulerer JVM-maskinen og dens utføring av byte-kode. Vi kan da gjerne si at et slikt simuleringsprogram interpreterer maskinkoden til JVM-maskinen. I dag har for eksempel de fleste nettlesere (Firefox, Opera, Explorer og andre) innebygget en slik JVM-interpret for å kunne utføre Java-applets når de får disse (ferdig kompilert) over nettet.

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

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

1.3.4 Forholdene i vår RusC-kompilator

I den oppgaven som skal løses i INF2100 skal vi i utgangspunktet lage en tradisjonell kompilator for språket *RusC*. Det å oversette til maskinkoden for en ekte maskin ville imidlertid føre for langt, og vi har derfor definert en litt enklere maskin som er kalt *Rask* som vi kommer tilbake til senere. For å kunne utføre programmer i Rask-kode må vi derfor ha et program som simulerer denne maskinen, altså en Rask-interpret.

Vi er altså på mange måter i samme situsjon som for tradisjonell Java (der vår Rask tilsvarer Javas JVM), men merk at begrunnelsene for en slik måte å gjøre det på er forskjellige: For JVM var begrunnelsen at man ønsket å kunne kjøre alle kompilerte Java-programmer på alle maskiner (portabilitet), mens vår begrunnelse for å definere maskinen Rask i INF2100 er å få en enkel datamaskin.

1.4 Språkene i oppgaven

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

1.4.1 Programmeringsspråket RusC

Det å lage en kompilator for til dømes Java ville fullstendig sprenge kursrammen på ti studiepoeng. I stedet har vi laget et språk spesielt for dette kurset med tanke på at det skal være overkommelig å oversette. Dette språket er kalt *RusC*. 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 RusC blir beskrevet i detalj i kapittel 2 på side 23.

1.4.2 Datamaskinen Rask og dens maskinspråk

Det en kompilator oversetter til, er vanligvis maskinspråket på den aktuelle maskin. Å forstå hovedideene for maskinspråket, for eksempel for en PowerPC- eller en Intel Pentium-maskin, ville ikke være veldig problematisk, men å bruke det som språk å oversette RusC til ville trekke med seg så mange detaljer at det ville bli uoverkommelig i et opplegg som dette.

For å få et språk å oversette til som er enkelt nok, og for samtidig å beholde et rimelig nært forhold til vanlige kompilatorer, vil vi altså her definere en egen forenklet datamaskin. Den vil ha en del typiske trekk fra vanlige datamaskiner, og vil gi en ganske god følelse for hvordan en datamaskin og dens maskinspråk vanligvis er utformet. En del detaljer er imidlertid fjernet.

Denne maskinen, med sitt maskinspråk og sine maskininstruksjoner er kalt *Rask*. Rask er en meget enkel maskin og kan for eksempel bare regne med heltall. Maskinen Rask er beskrevet i kapittel 3 på side 31.

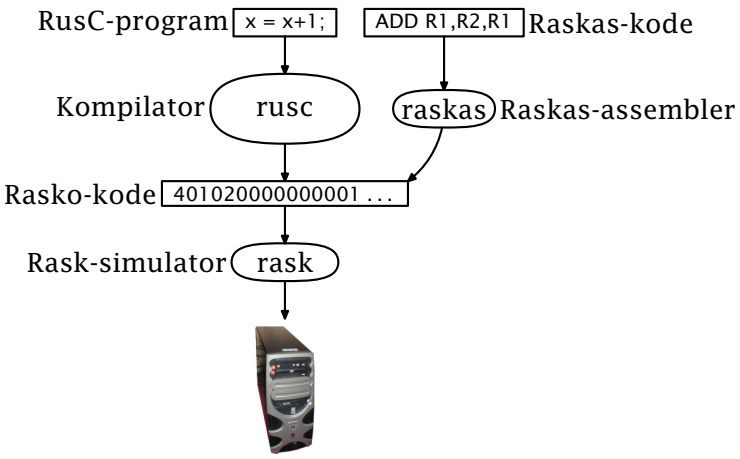
Det finnes altså ingen maskin som direkte – det vil si ved hjelp av elektronikk – kan utføre maskinspråket til Rask. At en slik maskin finnes er imidlertid den tenkte forutsetningen for den kompilatoren vi skal lage. For at denne forutsetningen skal være så nær som mulig oppfylt er det laget ferdig et program som *simulerer* maskinen Rask, både med hensyn til indre arkitektur og med hensyn til evne til å utføre maskininstruksjoner. Dette programmet starter med å lese et Rask-program fra en angitt fil. På filen må Rask-programmet da ligge i et spesielt format som kalles **Rasko**; denne koden er forklart nærmere i avsnitt 3.4 på side 32.

Slik vi framstiller det her er alt som lagres i Rask-maskinen tall. Dette kan gi næring til den vrangforestilling at datamaskiner dypest sett bare kan behandle tall, så la oss derfor straks se på hva som er tilfellet i en virkelig maskin. Den elementære lagringsenheten her kan bare ha to tilstander, som gjerne er kalt *av* og *på* eller 0 og 1. En slik lagringsenhet kalles gjerne et **bit** (som er en forkortelse for *binary digit*), og sekvenser av slike bit kan brukes like godt til å representere bokstaver som til å representere tall, bilder, eller hva annet man måtte ha behov for ved bare å bli enig om en passelig koding. Bit grupperes gjerne åtte og åtte til såkalte **byte**, og en datamaskin kan typisk inneholde flere milliarder slike byte i sitt interne lager. I det eksterne lager (vanligvis et platelager) kan det tilsvarende være plass til størrelsesorden hundre milliarder byte.

Når vi her lar lagerenhetene i Rask inneholde tall, er det fordi det å bruke bit ville bli for komplisert, og fordi tall gir riktige assosiasjoner i hvertfall et stykke på vei. Tall er også greie å behandle i Java og andre programmeringsspråk.

1.4.3 Assembleren Raskas

Vanligvis når man programmerer en datamaskin, beskriver man det man vil ha gjort i et passelig høynivå programmeringsspråk. En kompilator sørger så, slik som beskrevet over, for å produsere en tilsvarende sekvens av maskininstruksjoner og å legge denne ut på en angitt fil. Denne kan så senere lastes inn i maskinen og utføres. Vår RusC-kompilator skal altså gjøre nettopp dette.



Figur 1.1: Sammenhengen mellom RusC, Raskas, Rasko og Rask

Man har imidlertid også av og til behov for å skrive programmer (eller deler av programmer) direkte som en sekvens av maskininstruksjoner. Dette behovet oppstår vanligvis fordi man vil ha programmet spesielt effektivt, eller man ønsker å gjøre noe som høynivåspråket ikke tillater. Vi skal også programmere litt direkte i Rasks maskinspråk i dette kurset, men det er bare for å lære Rask-maskinen ordentlig å kjenne.

Når man skal programmere direkte i maskininstruksjoner, er det svært tungt å bruke tallkoder hele tiden, slik det fremtrer i Rasko-koden, og så godt som alle maskiner har derfor laget en tekstkode som er lettere å huske enn et tall. For eksempel kan man bruke «ADD» for en instruksjon som legger sammen heltall i stedet for til dømes tallet 4, som kan være instruksjonens egentlige kode. Man lager så et enkelt program som oversetter sekvenser av slike tekstlige instruksjoner til utførbar maskinkode-form, og slike oversetterprogrammer kalles tradisjonelt **assemblere**. Det oppsett eller format man må bruke for å angi et maskinprogram til assembleren, kalles gjerne **assemblerspråket**.

Assembler-språket for Rask kalles **Raskas**. Det er skrevet ferdig en assembler for dette språket, og denne ligger på Ifis datamaskiner. Raskas-assembleren tar som input en fil med Raskas-kode og oversetter denne til Rask-instruksjoner som blir lagt ut på en angitt fil i Rasko-format.

For treningens skyld skal vi i kurset skrive en del programmer i Raskas. Det nøyaktige formatet for assemblerspråket er beskrevet i kapittel 4 på side 35. Raskas-assembleren er altså bare skrevet for lettere å kunne sette seg inn i Rasks virkemåte og for å kunne trene seg i bruk av den. Den vil ikke inngå som del av den ferdige kompilator som skal skrives. Forholdet mellom RusC-kompilatoren, Raskas-assembleren, Rasko-koden og den utførende Rask-maskinen kan beskrives ved figur 1.1.

Både et RusC-program, et Raskas-program og et Rasko-program er altså et tekstlig program som ligger på en fil. Slike filer kan leses av RusC-kompilatoren, Raskas-assembleren og maskinen Rask. De to førstnevnte er programmer som begge er oversettere som produserer Rasko-kode, mens den sistnevnte er en simulator av en tenkt maskin.

Raskas-assembleren er tegnet litt mindre enn RusC-kompilatoren for å markere at den oversetter en «kortere» avstand, altså at et Raskas-program ligger «nærmere» maskinkode enn for eksempel RusC gjør. Ja, et Raskas-program er jo bare en tekstlig form for å angi Rask-instruksjon for Rask-instruksjon, mens RusC-kompilatoren jo må «finne på» en fornuftig sekvens av Rask-instruksjoner som gjør det som er foreskrevet i RusC-programmet. Man sier derfor gjerne at RusC er **høynivåspråk** mens for eksempel Raskas-kode er et **lavnivåspråk**. Det er selvfølgelig her snakk om en kontinuerlig skala og Java er for eksempel vesentlig «høyere» enn RusC.

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 fem språk med i bildet:

1) **Java**, som RusC-kompilatoren skal skrives i.
2) **RusC**, som kompilatoren skal oversette fra.
3) **Rasks maskinspråk**, som kompilatoren skal oversette til.
4) **Rasko-format**, som er et passelig tallformat for å lagre maskinprogrammer for Rask på fil.
5) **Raskas-kode** er en tekstlig form for maskininstruksjoner til Rask-maskinen. Raskas-assembleren kan oversette Raskas-koden til Rasko-kode.

I tillegg ligger altså maskinkoden til de aktuelle maskinene (som Pentium, PowerPC eller noe annet) under hele tiden, i og med at det er denne alt oversettes til før noe i det hele tatt kan utføres på disse maskinene. Hold tunga rett i munnen, og tenk litt på dette *hver kveld* så føler du deg snart hjemme i det hele!

1.5 Oppgaven og dens tre deler

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

Hele programmet kan grovt regnet bli på fra tusen til tre tusen Java-linjer, alt avhengig av hvor tett man skriver. Erfaringsmessig er det del 1 som innebærer mest arbeid. Vi gir her en rask oversikt over hva de tre delene vil inneholde, men vi kommer fyldig tilbake til hver av dem på forelesningene og i senere kapitler.

1.5.1 Del 0

Første skritt, del 0, består i å få **skanneren** 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 enhetene programmet er bygget opp

av, så som *navn*, *tall*, *nøkkelord*, '+', '>=', '=' og alle de andre tegn og tegnkombinasjoner som har en bestemt betydning i RusC-språket.

Denne «renskårene» sekvensen av symboler vil være det grunnlaget som resten av kompilatoren (del 1 og del 2) skal bruke til å arbeide videre med programmet. Mye av programmet til del 0 vil være ferdig laget eller skissert, og dette vil kunne hentes på angitt sted.

1.5.2 Del 1

Del 1 vil ta imot den symbolsekvensen som blir produsert av del 0, og det sentrale arbeidet her vil være å sjekke at denne sekvensen har den formen et riktig RusC-program skal ha (altså, at den følger **syntaksen** til RusC). Videre skal del 1 sjekke slike ting som at alle funksjoner og variable er deklarert.

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

1.5.3 Del 2

I del-2 skal man gjøre selve oversettelsen til Rask-kode ved å ta utgangspunkt i den trestrukturen som del 1 produserte for det aktuelle RusC-programmet. Koden skal legges på en fil angitt av brukeren og den skal være i såkalt Rasko-format.

I kapittel 5 på side 41 er det angitt hvilke sekvenser av Rask-instruksjoner hver enkelt RusC-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 Rask-kode; dette skal vi eventuelt se på i noen ukeoppgaver).

1.6 Programmering av lister, trær etc

Noe av hensikten med INF2100 er at man i størst mulig grad skal få en «hands on»-følelse med alle deler av programmeringen ned gjennom alle nivåer. Det er derfor et krav at gruppene selv programmerer all håndtering av lister og trær, og ikke bruker ferdiglagede bibliotekspakker og slikt til det. Med andre ord, det er ikke lov å importere andre Java-klasser enn `java.io.*`.

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

1.7 Krav til samarbeid og gruppetilhørighet

Normalt er det meningen at to personer skal samarbeide om å løse oppgaven. De som samarbeider bør være fra samme øvelsesgruppe 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.

1.8 Kontroll av innlevert arbeid

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

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

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

Et annet krav er at alle innleverte program er vesentlig ulike 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: Samarbeide er bra, men kopiering er galt!

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

1.9 Delta på gruppene

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

Selv om man ikke har forberedt seg er det imidlertid helt lov på gruppa å komme med et uartikulert:

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

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

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

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

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

Kapittel 2

Programmeringsspråket RusC

Programmeringsspråket RusC er en slags miniversjon av C; navnet er et bakronym[1] for «Rudimentary simple C». Syntaksen er gitt av jernbanediagrammene i figur 2.2 til 2.15 på side 25 og utover og bør være lett forståelig for alle som har programmert litt i C. Et eksempel på et RusC-program er vist i figur 2.1 på neste side.[2]

2.1 Kjøring

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

```
> rusc primes.rusc
> rask primes.rask
   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
```

[1] Et *akronym* er en forkortelse som kan uttales som et vanlig ord og derfor ofte oppfattes som en eget ord; eksempler er AIDS («acquired immune deficiency syndrome»), NATO («North Atlantic treaty organization») og RADAR («radio detection and ranging»).

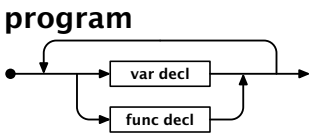
Et *bakronym* er et akronym der man starter med ordet og etterpå finner ut hva det skal være en forkortelse for; eksempler er programmeringsspråket PERL («Practical extraction and report language») og kommunikasjonsprotokollen TWAIN («Technology without an interesting name»).

[2] Du finner dette programmet og andre nyttige testprogrammer i mappen ~inf2100/oblig/test/ som også er tilgjengelig som http://www.ifi.uio.no/~inf2100/oblig/test/.

primes.rusc

```
/* Program 'primes'
   ----------------
   Finds all prime numbers up to 1000 (using the technique called
   "the sieve of Eratosthenes") and prints them nicely formatted.
*/

#include "/local/opt/inf2100/include/rusc.h"

int prime[1001];  /* The sieve */
int LF;           /* LF */

int find_primes ()
{
   /* Remove all non-primes from the sieve: */

   int i1, i2;

   i1 = 2;
   while (i1 <= 1000) {
      i2 = 2*i1;
      while (i2 <= 1000) {
         prime[i2] = 0;  i2 = i2+i1;
      }
      i1 = i1+1;
   }
}

int mod (int a, int b)
{
   /* Computes a%b. */

   int ax;

   ax = a/b;  ax = ax*b;
   return a - ax;
}

int p3 (int v)
{
   /* Does a 'printf("%3d", v)´;
      assumes 0<=v<=999. */

   if (v <= 9) {
      putchar(' ');   putchar(' ');
   } else {
      if (v <= 99) {
         putchar(' ');
      };
   }
   putint(v);
}

int print_primes ()
{
   /* Print the primes, 10 on each line. */

   int n_printed, i;

   n_printed = 0;  i = 1;
   while (i <= 1000) {
      if (prime[i]) {
         if (mod(n_printed,10) == 0 * n_printed) {
            putchar(LF);
         }
         putchar(' ');   p3(i);   n_printed = n_printed+1;
      }
      i = i+1;
   }
   putchar(LF);
}

int main ()
{
   int i;

   LF = 10;
   /* Initialize the sieve by assuming all numbers >1 to be primes: */
   prime[1] = 0;
   i = 2;
   while (i <= 1000) {
      prime[i] = 1;  i = i+1;
   }

   /* Find and print the primes: */
   find_primes();  print_primes();
}
```

Figur 2.1: Eksempel på et RusC-program



Figur 2.2: Jernbanediagram for ⟨program⟩

2.1.1 Kompilering med C-kompilatoren

Siden RusC er en nesten ekte undermengde av C, kan man bruke C-kompilatoren til å lage kjørbar kode. Det eneste man må sørge for, er at biblioteksfunksjonene kommer med; dette gjøres med en `#include`-spesifikasjon, se linje 7 i figur 2.1 på forrige side.

```
> gcc -x c -o primes primes.rusc
> ./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
```

2.2 RusC-program

Et RusC-program er rett og slett en samling funksjoner (kjent som «metoder» i Java). Før, mellom og etter disse funksjonene kan man deklarere globale variable.

Det må alltid eksistere en funksjon med navn `main` og programutførelsen starter alltid med å kalle denne funksjonen. (Funksjonen `main` kan ikke ha noen parametre.)

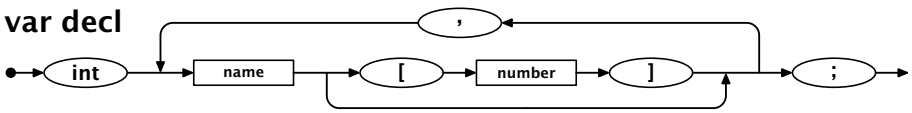
2.2.1 Variable

Brukeren kan deklarere enten globale variable eller variable som er lokale i en funksjon. Den eneste datatypen er `int`, men det er mulig å deklarere vektorer («array»-er) av `int`-er. Vektorer indekseres fra 0 (som i C og Java).

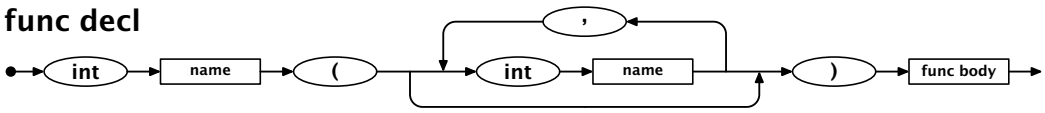
Det er ikke mulig å angi noen initialverdi for variable – de inneholder en ukjent verdi før de tilordnes en ny verdi av programmet.

2.2.2 Funksjoner

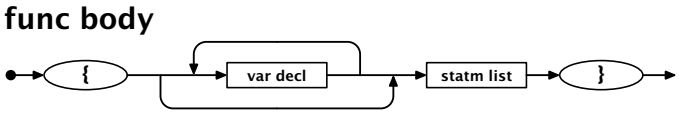
Brukeren kan deklarere funksjoner med inntil fire parametre; parametrene kan ikke være vektorer. Parameteroverføringen skjer ved kopiering (som i C og Java).



Figur 2.3: Jernbanediagram for ⟨var decl⟩



Figur 2.4: Jernbanediagram for ⟨func decl⟩



Figur 2.5: Jernbanediagram for ⟨func body⟩

Funksjon	Effekt
`int exit (int status)`	Gir dump om `status`≠0; avslutter programmet med angitt statusverdi.
`int getchar ()`	Leser neste tegn fra tastaturet.
`int getint ()`	Leser neste heltall fra tastaturet.
`int putchar (int c)`	Skriver et tegn på skjermen.
`int putint (int c)`	Skriver et heltall på skjermen.

Tabell 2.1: RusCs biblioteksfunksjoner

Alle funksjoner er `int`-funksjoner, dvs at de returnerer en `int`-verdi. Hvis ingen eksplisitt verdi er angitt (med en `return`-setning), returneres en tilfeldig verdi.

Det er ikke lov å kalle en funksjon rekursivt (dvs la en funksjon kalle seg selv). Det er også bare lov å kalle funksjoner som er definert tidligere i programkoden.

2.2.2.1 Biblioteket

RusC kjenner til de fem biblioteksfunksjonene som er vist i tabell 2.1. Disse kan brukes uten noen spesiell importering.

2.2.3 Setninger

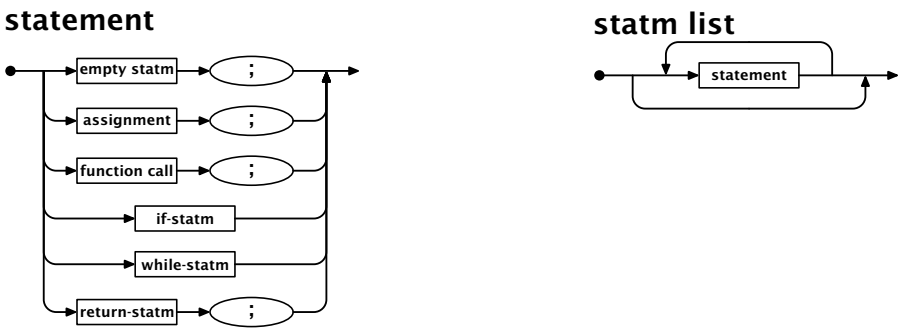
RusC kjenner til seks ulike setninger.

2.2.3.1 Funksjonskall

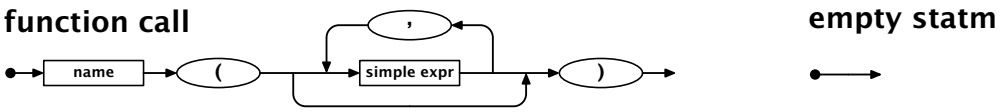
Et funksjonskall kan brukes som en egen setning.

2.2.3.2 Tomme setninger

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

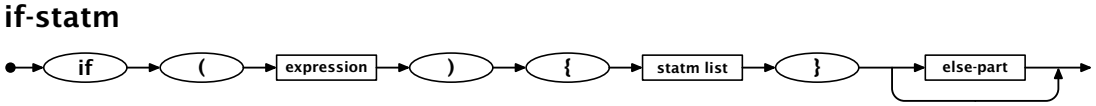


Figur 2.6: Jernbanediagram for ⟨statement⟩ og ⟨statm list⟩



Figur 2.7: Jernbanediagram for ⟨function call⟩ og ⟨empty statm⟩

Figur 2.8: Jernbanediagram for ⟨assignment⟩ og ⟨return-statm⟩



Figur 2.9: Jernbanediagram for ⟨if-statm⟩

2.2.3.3 Tilordninger

Det er tillatt å tilordne verdier til vanlige enkle variable eller vektorelementer.

2.2.3.4 `return`-setninger

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

2.2.3.5 `if`-setninger

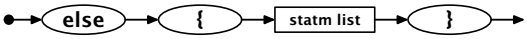
Disse setningene brukes til å velge om noen setninger skal utføres eller ikke. Selve testen er et uttrykk som beregnes: verdien 0 angir *usann* mens alle andre verdier angir *sann*.

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

2.2.3.6 `while`-setninger

En `while`-setning går i løkke inntil testuttrykket beregnes til 0.

else-part

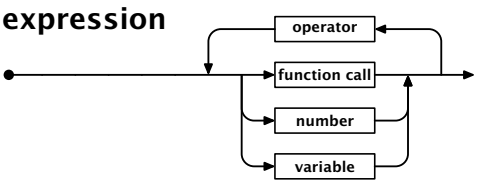


Figur 2.10: Jernbanediagram for ⟨else-part⟩

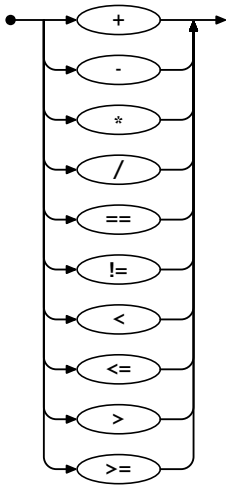
while-statm

Figur 2.11: Jernbanediagram for ⟨while-statm⟩

expression

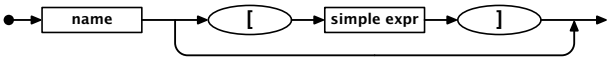


operator



Figur 2.12: Jernbanediagram for ⟨expression⟩ og ⟨operator⟩

variable



Figur 2.13: Jernbanediagram for ⟨variable⟩

2.2.4 Uttrykk

Uttrykk i RusC har ingen parenteser og ingen presedens[3] – alle uttrykk beregnes alltid rett frem fra venstre mot høyre.

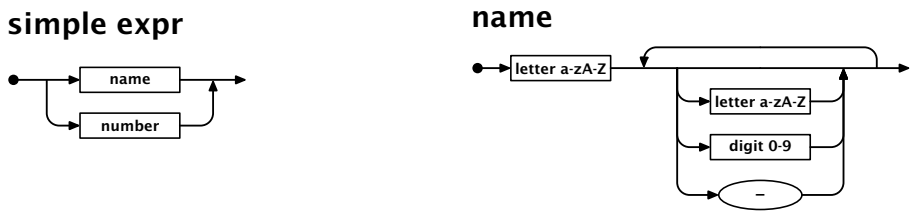
2.2.4.1 Enkle uttrykk

I noen sammenhenger (dvs som parametre eller som indekser i vektorer) er det bare lov å ha ganske enkle former for uttrykk, dvs bare en enkel variabel eller et tall.

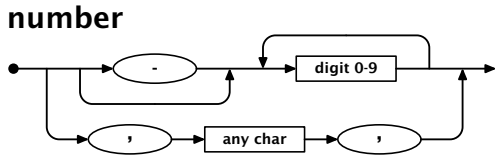
[3] Vanligvis har operatorer ulik *presedens*, dvs at noen operator binder sterkere enn andre. Når vi skriver for eksempel

$$a + b \times c$$

tolkes dette vanligvis som $a + (b \times c)$ fordi $\times$ normalt har høyere presedens enn $+$, dvs $\times$ binder sterkere enn $+$. Men slik er det altså ikke i RusC!



Figur 2.14: Jernbanediagram for ⟨simple expr⟩ og ⟨name⟩



Figur 2.15: Jernbanediagram for ⟨number⟩

2.2.4.2 Tall og tegn

Heltall brukes som vanlig i RusC. I tillegg er det lov å bruke tegnkonstanter (men altså ikke tekster på mer enn ett tegn). Slike tegnkonstanter betraktes som tall der verdien er tegnets representasjon i ISO 8859-1 som er standard tegnsett ved Ifi; se tabell 2.2 på neste side.

2.2.5 Andre ting

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

- Kommentarlinjer har et «#» som aller første tegn; da skal hele linjen ignoreres.
- Kommentarer kan også angis som «`/*...*/`» og kan da strekke seg over flere linjer.

2.3 Forskjeller til C

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

- Det er ingen parenteser i uttrykk.
- Det er bare én datatype: `int`. Det er altså ingen `struct`-er eller `union`-er og heller ingen `typedef`-er.
- Funksjoner kan bare ha inntil fire parametre, og vektorer kan ikke være parametre.
- Det er ingen `for`-setninger og heller ingen `do-while`- eller `switch`-setninger.
- Det er ingen preprosessordirektiver («#*xxx*-linjer»).[4]
- Funksjonsbiblioteket er minimalt sammenlignet med C.

[4] #-linjer er lov i RusC men betraktes kun som kommentarer.

ISO 8859–1

0	000 00	32	040 20	64	@	100 40	96	‘	140 60	128	200 80	160	240 A0	192	À	300 C0	224	à	340 E0
1	001 01	33	!	041 21	65	A	101 41	97	a	141 61	129	201 81	161	¡	241 A1	193	Á	301 C1	225	á	341 E1
2	002 02	34	"	042 22	66	B	102 42	98	b	142 62	130	202 82	162	¢	242 A2	194	Â	302 C2	226	â	342 E2
3	003 03	35	#	043 23	67	C	103 43	99	c	143 63	131	203 83	163	£	243 A3	195	Ã	303 C3	227	ã	343 E3
4	004 04	36	$	044 24	68	D	104 44	100	d	144 64	132	204 84	164	¤	244 A4	196	Ä	304 C4	228	ä	344 E4
5	005 05	37	%	045 25	69	E	105 45	101	e	145 65	133	205 85	165	¥	245 A5	197	Å	305 C5	229	å	345 E5
6	006 06	38	&	046 26	70	F	106 46	102	f	146 66	134	206 86	166	¦	246 A6	198	Æ	306 C6	230	æ	346 E6
7	007 07	39	’	047 27	71	G	107 47	103	g	147 67	135	207 87	167	§	247 A7	199	Ç	307 C7	231	ç	347 E7
8	010 08	40	(	050 28	72	H	110 48	104	h	150 68	136	210 88	168	¨	250 A8	200	È	310 C8	232	è	350 E8
9	011 09	41	)	051 29	73	I	111 49	105	i	151 69	137	211 89	169	©	251 A9	201	É	311 C9	233	é	351 E9
10	012 0A	42	*	052 2A	74	J	112 4A	106	j	152 6A	138	212 8A	170	ª	252 AA	202	Ê	312 CA	234	ê	352 EA
11	013 0B	43	+	053 2B	75	K	113 4B	107	k	153 6B	139	213 8B	171	«	253 AB	203	Ë	313 CB	235	ë	353 EB
12	014 0C	44	,	054 2C	76	L	114 4C	108	l	154 6C	140	214 8C	172	¬	254 AC	204	Ì	314 CC	236	ì	354 EC
13	015 0D	45	–	055 2D	77	M	115 4D	109	m	155 6D	141	215 8D	173	-	255 AD	205	Í	315 CD	237	í	355 ED
14	016 0E	46	.	056 2E	78	N	116 4E	110	n	156 6E	142	216 8E	174	®	256 AE	206	Î	316 CE	238	î	356 EE
15	017 0F	47	/	057 2F	79	O	117 4F	111	o	157 6F	143	217 8F	175	¯	257 AF	207	Ï	317 CF	239	ï	357 EF
16	020 10	48	0	060 30	80	P	120 50	112	p	160 70	144	220 90	176	°	260 B0	208	Ð	320 D0	240	ð	360 F0
17	021 11	49	1	061 31	81	Q	121 51	113	q	161 71	145	221 91	177	±	261 B1	209	Ñ	321 D1	241	ñ	361 F1
18	022 12	50	2	062 32	82	R	122 52	114	r	162 72	146	222 92	178	²	262 B2	210	Ò	322 D2	242	ò	362 F2
19	023 13	51	3	063 33	83	S	123 53	115	s	163 73	147	223 93	179	³	263 B3	211	Ó	323 D3	243	ó	363 F3
20	024 14	52	4	064 34	84	T	124 54	116	t	164 74	148	224 94	180	´	264 B4	212	Ô	324 D4	244	ô	364 F4
21	025 15	53	5	065 35	85	U	125 55	117	u	165 75	149	225 95	181	µ	265 B5	213	Õ	325 D5	245	õ	365 F5
22	026 16	54	6	066 36	86	V	126 56	118	v	166 76	150	226 96	182	¶	266 B6	214	Ö	326 D6	246	ö	366 F6
23	027 17	55	7	067 37	87	W	127 57	119	w	167 77	151	227 97	183	·	267 B7	215	×	327 D7	247	÷	367 F7
24	030 18	56	8	070 38	88	X	130 58	120	x	170 78	152	230 98	184	¸	270 B8	216	Ø	330 D8	248	ø	370 F8
25	031 19	57	9	071 39	89	Y	131 59	121	y	171 79	153	231 99	185	¹	271 B9	217	Ù	331 D9	249	ù	371 F9
26	032 1A	58	:	072 3A	90	Z	132 5A	122	z	172 7A	154	232 9A	186	º	272 BA	218	Ú	332 DA	250	ú	372 FA
27	033 1B	59	;	073 3B	91	[	133 5B	123	{	173 7B	155	233 9B	187	»	273 BB	219	Û	333 DB	251	û	373 FB
28	034 1C	60	<	074 3C	92	\	134 5C	124	|	174 7C	156	234 9C	188	¼	274 BC	220	Ü	334 DC	252	ü	374 FC
29	035 1D	61	=	075 3D	93	]	135 5D	125	}	175 7D	157	235 9D	189	½	275 BD	221	Ý	335 DD	253	ý	375 FD
30	036 1E	62	>	076 3E	94	^	136 5E	126	~	176 7E	158	236 9E	190	¾	276 BE	222	Þ	336 DE	254	þ	376 FE
31	037 1F	63	?	077 3F	95	_	137 5F	127		177 7F	159	237 9F	191	¿	277 BF	223	ß	337 DF	255	ÿ	377 FF

Tabell 2.2: Tegnsettet ISO 8859-1

Kapittel 3

Datamaskinen Rask

Rask er en enkel datamaskin inspirert av MIPS-prosessoren som var meget populær på 1990-tallet. Det er en typisk RISC[1]-maskin.

Maskinen Rask har

1) en prosessor,
2) 32 generelle registre,
3) et **PC**-register og
4) et minne med 10 000 lokasjoner.

3.1 Registrene

Registre er små datalagre inne i selve prosesoren i datamaskinen; hvert register kan inneholde ett tall. Datamaskiner har oftest 4–32 registre, og vår Rask-maskin har altså 32 med de fantasiløse navnene $\mathbf{R_0}$–$\mathbf{R_{31}}$. (I tillegg har Rask et **PC**-register som alle andre datamaskiner.)

3.1.1 Spesielle registre

De fleste av registrene våre er generelle, men to er ganske spesielle:

$\mathbf{R_0}$ inneholder alltid verdien 0.

$\mathbf{R_{31}}$ får automatisk returadressen når man foretar et funksjonskall (med instruksjonen CALL).

3.2 Instruksjonene

Rask har 17 *instruksjoner* vist i tabell 3.1 på neste side.

[1] RISC («Reduced instruction set computer») betegner datamaskiner med få og enkle instruksjoner som kan utføres meget raskt. Det motsatte er CISC («Complex instruction set computer»).

Nr	Navn	Operasjon
1	LOAD $\mathbf{R_A}$, $\mathbf{R_B}$, C	$\mathbf{R_A} \leftarrow \text{Mem}[\mathbf{R_B}+C]$
2	SET $\mathbf{R_A}$, $\mathbf{R_B}$, C	$\mathbf{R_A} \leftarrow \mathbf{R_B}+C$
3	STORE $\mathbf{R_A}$, $\mathbf{R_B}$, C	$\text{Mem}[\mathbf{R_B}+C] \leftarrow \mathbf{R_A}$
4	ADD $\mathbf{R_A}$, $\mathbf{R_B}$, $\mathbf{R_C}$	$\mathbf{R_A} \leftarrow \mathbf{R_B}+\mathbf{R_C}$
5	SUB $\mathbf{R_A}$, $\mathbf{R_B}$, $\mathbf{R_C}$	$\mathbf{R_A} \leftarrow \mathbf{R_B}-\mathbf{R_C}$
6	MUL $\mathbf{R_A}$, $\mathbf{R_B}$, $\mathbf{R_C}$	$\mathbf{R_A} \leftarrow \mathbf{R_B}\times\mathbf{R_C}$
7	DIV $\mathbf{R_A}$, $\mathbf{R_B}$, $\mathbf{R_C}$	$\mathbf{R_A} \leftarrow \mathbf{R_B}/\mathbf{R_C}$
8	EQ $\mathbf{R_A}$, $\mathbf{R_B}$, $\mathbf{R_C}$	$\mathbf{R_A} \leftarrow 1$ hvis $\mathbf{R_B}=\mathbf{R_C}$, ellers 0
9	NEQ $\mathbf{R_A}$, $\mathbf{R_B}$, $\mathbf{R_C}$	$\mathbf{R_A} \leftarrow 1$ hvis $\mathbf{R_B}\neq\mathbf{R_C}$, ellers 0
10	LESS $\mathbf{R_A}$, $\mathbf{R_B}$, $\mathbf{R_C}$	$\mathbf{R_A} \leftarrow 1$ hvis $\mathbf{R_B}<\mathbf{R_C}$, ellers 0
11	LESSEQ $\mathbf{R_A}$, $\mathbf{R_B}$, $\mathbf{R_C}$	$\mathbf{R_A} \leftarrow 1$ hvis $\mathbf{R_B}\leq\mathbf{R_C}$, ellers 0
12	GTR $\mathbf{R_A}$, $\mathbf{R_B}$, $\mathbf{R_C}$	$\mathbf{R_A} \leftarrow 1$ hvis $\mathbf{R_B}>\mathbf{R_C}$, ellers 0
13	GTREQ $\mathbf{R_A}$, $\mathbf{R_B}$, $\mathbf{R_C}$	$\mathbf{R_A} \leftarrow 1$ hvis $\mathbf{R_B}\geq\mathbf{R_C}$, ellers 0
14	JUMPEQ $\mathbf{R_A}$, $\mathbf{R_B}$, C	Hvis $\mathbf{R_A}=\mathbf{R_B}$: $\mathbf{PC} \leftarrow C$
15	JUMPNEQ $\mathbf{R_A}$, $\mathbf{R_B}$, C	Hvis $\mathbf{R_A}\neq\mathbf{R_B}$: $\mathbf{PC} \leftarrow C$
16	CALL $\mathbf{R_A}$, $\mathbf{R_B}$, C	$\mathbf{R_{31}} \leftarrow \mathbf{PC}$ og $\mathbf{PC} \leftarrow C$
17	RET	$\mathbf{PC} \leftarrow \mathbf{R_{31}}$

Tabell 3.1: Rask-instruksjoner

3.2.1 Koding av instruksjonene

For at et program skal kunne kjøres på Rask må instruksjonene lagres som tall. Alle Rask-instruksjonene består av fire deler:

Nr angir hvilken instruksjon det er snakk om; lovlige verdier er 1–17.

A angir et register 0–31.

B angir også et register 0–31.

C er enten et register 0–31, en minneadresse 0–9 999 eller positiv tallkonstant 0–9 999 999 999.

Instruksjonen lagres slik:

$$\text{Nr} \cdot 10^{14} + A \cdot 10^{12} + B \cdot 10^{10} + C$$

3.3 Operativsystemet

Rask har et ørlite operativsystem. Det gir tilgang til I/O-operasjoner og annet når vi foretar et CALL til de «magiske» adressene vist i tabell 3.2 på neste side.

3.4 Maskinkoden Rasko

Når Rask-maskinen starter, leser den en fil som forteller hva som skal ligge i minnecellene i maskinen når kjøringen starter; det dreier seg da om både

Adresse	Funksjon
9990	`exit(`$\mathbf{R}_{11}$`)`
9991	`getchar()`
9992	`getint()`
9993	`putchar(`$\mathbf{R}_{11}$`)`
9994	`putint(`$\mathbf{R}_{11}$`)`

Tabell 3.2: «Magiske» minnelokasjoner i Rask

instruksjoner og initialverdier for variable.[2] Denne filen er i et spesielt format kalt Rasko.

Rasko-formatet er meget enkelt:

- Tegnet # og alt som står etter på linjen, ignoreres.
- Ellers inneholder filen bare diverse heltall; det kan stå vilkårlig mange på hver linje. Første tall angir innholdet i celle nr. 0, det neste i celle nr. 1, osv.

Det er også vanlig at første linje i en Rasko-fil ser slik ut:

```
#! /local/bin/rask
```

slik at kodefilen kan gjøres eksekverbar i Unix, men dette er intet krav. Uansett blir denne linjen ignorert som kommentar (slik reglene ovenfor tilsier) av Rask-maskinen.

[2] Husk at Rask-maskinen i utgangspunktet ikke vet forskjell på instruksjoner og data – for den er alt tall.

Kapittel 4

Assemblerkode Raskas

Når man vil ha maskinen Rask til å utføre en bestemt oppgave, må man først og fremst fylle instruksjonslageret med en sekvens av instruksjoner som tilsammen løser oppgaven når maskinen blir satt i gang. For en del oppgaver er det også aktuelt å fylle bestemte tall i noen av cellene som skal brukes som data.

For at vi skal slippe å bruke tallkoder når vi skal «håndprogrammere» maskinen Rask, er det laget en bokstavkode (på tre til syv bokstaver) for hver instruksjon, og det finnes et program (kalt en *assembler*) som oversetter dette til riktig tallkode. Bokstavkoden er den samme som er brukt i beskrivelsen av Rask-maskinen (se tabell 3.1 på side 32). Assembleren (og assembler-språket) på Rask kalles Raskas.

Når vi skal skrive et program på denne måten skriver vi én instruksjon på hver linje, og bak bokstavkoden for instruksjonen kan man angi adressedelen som et tall (eller som et adressenavn; det omtales lenger ned). Av grunner vi skal se senere må instruksjonskoden ikke starte i første posisjon på linja.

En del av et program kan for eksempel se slik ut:

```
CALL    putchar    # putchar(   );

CALL    getint     # R1 = getint();
SET     R3,1       #          1
ADD     R11,R1,R3  # R11 = R1+ ;
```

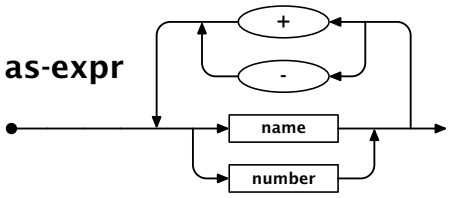
4.1 Kommentarer og blanke linjer

Blanke linjer vil Flass-assembleren bare hoppe over, og om man vil ha kommentarer på en linje kan man bare skrive '#' så vil *resten av linjen* da bli tatt som kommentar. Det er vanlig når man skriver slike programmer at alle instruksjonskodene og alle adressedelene skrives rett under hverandre.

4.2 Instruksjonslinjer

Hver instruksjonslinje ser slik ut:

⟨etikett⟩ ⟨instr⟩ ⟨param$_1$⟩ ⟨param$_2$⟩ ⟨param$_3$⟩ # ⟨kommentar⟩



Figur 4.1: Jernbanediagram for uttrykk i Raskas

De ulike feltene er:

⟨etikett⟩ omtales i avsnitt 4.3 på neste side.

⟨instr⟩ er én av følgende

- et instruksjonsnavn fra listen i tabell 3.1 på side 32,
- ordet «`INT`» (som omtales i avsnitt 4.4 på side 38),
- ordet «`RES`» (som også omtales i avsnitt 4.4 på side 38) eller
- ordet «`EQU`» (som du kan lese om i avsnitt 4.5 på side 38).

Disse navnene kan skrives med store eller små bokstaver, avhengig av personlig smak.

⟨param$_n$⟩ angir parametre til instruksjonene nevnt over. For `INT`, `RES` og `EQU` vil dette alltid være en verdi (se avsnitt 4.2.1), mens for instruksjonene i tabell 3.1 på side 32 vil det være

⟨param$_1$⟩ vil alltid være et register (altså `R0`–`R31`).

⟨param$_2$⟩ vil også alltid være et register.

⟨param$_3$⟩ vil enten være et register eller en verdi; dette avhenger av hvilken instruksjon det er snakk om og tabell 3.1 på side 32 viser dette klart.

Det er også verdt å merke seg at man kan droppe parametre om de er $\mathbf{R_0}$ eller 0. Dette skjer etter følgende regel:

- Om det bare er gitt to parametre, antas at ⟨param$_2$⟩ skal være $\mathbf{R_0}$.
- Om det bare er gitt én parameter, antas i tillegg at ⟨param$_1$⟩ skal være $\mathbf{R_0}$.
- Er det ikke gitt noen parametre, er dessuten ⟨param$_3$⟩ enten 0 eller $\mathbf{R_0}$ (avhengig av instruksjonen).

4.2.1 Verdier

En Raskas-verdi gis som et enkelt uttrykk etter grammatikken vist i figur 4.1. (Definisjonen av ⟨name⟩ og ⟨number⟩ finner du i figurene 2.14 og 2.15 på side 29.)

4.3 Etiketter

Assembleren vil legge instruksjonene den får oppgitt etter hverandre fra celle 0 og utover, og den har ingen som helst kontroll over om det er en rimelig sekvens av instruksjoner man oppgir. Hvis det var starten av programmet vi anga på side 35, ville altså '`CALL putchar`' komme som instruksjon nr 0, '`CALL getint`' som instruksjon nr 1 osv. De cellene i lageret man ikke sier noe om, vil ha en ukjent verdi når man (etter at assembleren er ferdig) starter programmet.

Om man skal lage en hoppinstruksjon skal adressedelen av denne være nummeret på den instruksjonscellen man vil hoppe til. Hvis dette er en del av et program:

```
        ADD     R1,R1,R2   # i celle 10
        ADD     R2,R2,R3   # i celle 11
        JUMPNEQ 10         # i celle 12
```

vil altså hoppet gå til instruksjonen '`ADD R1,R1,R2`', da den ligger i celle 10. Dette å holde greie på hvor de forskjellige instruksjonene havner, er imidlertid en tung affære, og om man etterpå finner ut at man skulle begynne programmet litt annerledes slik at ting flytter seg noen «hakk» i lageret, ville mange adressedeler måtte forandres.

Siden det er assembleren som har størst oversikt over hvor ting havner i lageret, tilbyr den hjelp for å komme ut av dette problemet. Man kan foran en instruksjon sette et selvvalgt navn (med vanlig navne-syntaks, som vist i figur 2.14 på side 29). Et slikt navn må starte i første posisjon på linjen. Dette navnet vil så av assembleren bli identifisert med nummeret på den cellen der den etterfølgende instruksjonen havner, og det kan derfor brukes i stedet for tall, som adressedel til en instruksjon. Hvilket tall et navn blir identifisert med er da ofte ikke så viktig, men det kan alltid leses ut av listingen fra assembleren om man er interessert.

Det trengs ikke noe mer deklarasjon av et slikt navn enn at det forekommer foran en instruksjon en gang, og det kan brukes som adressedel mange ganger i programmet, både foran og etter den instruksjonen der det er definert. Programmet over kunne da vært skrevet:

```
loop    ADD     R1,R1,R2   # i celle 10
        ADD     R2,R2,R3   # i celle 11
        JUMPNEQ loop       # i celle 12
```

Navnet `loop` blir altså her av assembleren identifisert med tallet 10. Legg merke til at man også godt kan bruke dette i instruksjonen

```
        SET     R1,loop
```

Utførelsen av denne fører altså til at Reg1-registeret får verdien 10, nøyaktig som om vi hadde skrevet '`SET R1,10`'.

Navnet `loop` kalles en *etikett* («label» på engelsk).[1] Det er også mulig å skrive en etikett alene på en linje. Den vil da referere til neste celle i instruksjonslageret, og man kan da oppnå å ha flere navn på samme instruksjon. I programbiten

[1] Mange liker å skrive et kolon bak slike etiketter der de defineres. Det kan man godt gjøre; slike kolon vil bare bli ignorert.

```
        JUMPEQ  R1,R0,Hit
         ⋮
Hit:
Her:    SET     R1,2
         ⋮
        JUMPEQ  R1,R0,Her
```

vil hoppene til `Hit` og `Her` havne samme sted, nemlig til instruksjonen '`SET R1,2`'.

Etikettene i tabell 3.2 på side 33 er forhåndsdefinert, så de kan man bruke uten videre.

4.4 Å reservere plass til variable

Assembleren tilbyr også hjelp når det gjelder å fylle opp og å administrere plassen i lageret for heltall. Om man midt inne i instrukssjonssekvensen finner ut at man vil fylle en celle i heltallslageret med et gitt tall, kan man bare skrive for eksempel:

```
Var1    INT     27
```

I og med at det i «funksjonsdelen» her står `INT`, forstår assembleren at du vil sette av en datacelle i minnet og at denne skal ha en gitt initialverdi.

Om man blir avansert, og vil sette av et større område i lageret, vil assembleren gjøre dette om man sier for eksempel:

```
Array   RES     28
```

Dette tilsvarer 28 linjer med «`INT 0`». Etiketten (med navnet `Array` her) vil da være adressen til den første cellen.

4.5 Å definere konstanter

En siste mulighet man har i en assembler er at man kan definere konstanter, altså navn for gitte tallverdier.[2]

```
Antall  EQU     8
         ⋮
Tab:    RES     Antall
```

Her settes det av 8 celler.

[2] En konstant er nesten det samme som man oppnår ved å bruke etiketter, men man kan gi konstanter nøyaktig hvilken tallverdi man ønsker – etiketter får alltid en tallverdi som er adressen til neste celle i minnet.

nexta.raskas

```
# Et minimalt testprogram:
# Det ber om et tall v og skriver så ut v+1.

main:   SET     R11,'?'     #         '?'
        CALL    putchar     # putchar(   );
        SET     R11,' '     #         ' '
        CALL    putchar     # putchar(   );

        CALL    getint      # R1 = getint();
        SET     R3,1        #          1
        ADD     R11,R1,R3   # R11 = R1+ ;
        CALL    putint      # putint(R11);
        SET     R11,10      #         LF
        CALL    putchar     # putchar(  );

        SET     R11,0       #      0
        CALL    exit        # exit( );
```

Figur 4.2: Et eksempelprogram skrevet i Raskas-kode

nexta.list

```
Raskas assembler version 1.0
Source file: nexta.raskas

                        # Et minimalt testprogram:
                        # Det ber om et tall v og skriver så ut v+1.

   0:  2 11  0      63  main:   SET     R11,'?'     #         '?'
   1: 16  0  0    9993          CALL    putchar     # putchar(   );
   2:  2 11  0      32          SET     R11,' '     #         ' '
   3: 16  0  0    9993          CALL    putchar     # putchar(   );

   4: 16  0  0    9992          CALL    getint      # R1 = getint();
   5:  2  3  0       1          SET     R3,1        #          1
   6:  4 11  1       3          ADD     R11,R1,R3   # R11 = R1+ ;
   7: 16  0  0    9994          CALL    putint      # putint(R11);
   8:  2 11  0      10          SET     R11,10      #         LF
   9: 16  0  0    9993          CALL    putchar     # putchar(  );

  10:  2 11  0       0          SET     R11,0       #      0
  11: 16  0  0    9990          CALL    exit        # exit( );

Program labels:
exit       9990
getchar    9991
getint     9992
main          0
putchar    9993
putint     9994
```

Figur 4.3: Listing produsert utifra programmet i figur 4.2

4.6 Et eksempel

I figur 4.2 er vist kildefilen til et lite program skrevet i Raskas-kode, og figurene 4.3 og 4.4 viser henholdsvis listingen og koden produsert ved å utføre kommandoen

```
> raskas nexta.raskas
```

Den genererte kodefilen kan selvfølgelig utføres:

```
> rask nexta.rask
? -754
-753
```

nexta.rask

```
#! /local/bin/rask

211000000000063
1600000000009993
211000000000032
1600000000009993
1600000000009992
203000000000001
411010000000003
1600000000009994
211000000000010
1600000000009993
211000000000000
1600000000009990
```

Figur 4.4: Kjørbar Rasko-kode produsert utifra programmet i figur 4.2

Kapittel 5

Kodegenerering

5.1 Konvensjoner

Når vi skal generere kode, er det en stor fordel å være enige om visse ting, for eksempel registerbruk. Derfor skal noen registre brukes slik det er angitt i tabell 5.1 på neste side.

5.2 Beregning av uttrykk

5.2.1 Operander i uttrykk

I tabell 5.2 på neste side er vist hvilkes kode som må genereres for å hente en verdi inn i register $\mathbf{R_1}$. Funksjonskall er ikke tatt med her – det er beskrevet i avsnitt 5.6.2 på side 44. Legg forøvrig spesielt merke til dette:

- Negative konstanter må spesialbehandles: først setter man inn den positive verdien og etterpå snur man fortegnet med en SUB-instruksjon.

5.2.2 Operatorer i uttrykk

I tabell 5.3 på neste side er vist hvorledes man oversetter en addisjon; de andre operasjonen gjøres helt tilsvarende.

5.3 Tilordning

Se tabell 5.4.

5.4 If-setninger

It-tester oversettes som vist i tabell 5.5.

5.5 While-setninger

While-løkker oversettes som vist i tabell 5.6 på side 43.

$\mathbf{R}_0$	Alltid 0	(gitt av Rask)
$\mathbf{R}_1$	Hovedregister for beregning av uttrykk	
$\mathbf{R}_2$	Hjelperegister ved vektoraksess	
$\mathbf{R}_3$	Hjelperegister ved beregning av uttrykk	
$\mathbf{R}_{11}$	Parameter 1 ved funksjonskall	
$\mathbf{R}_{12}$	Parameter 2 ved funksjonskall	
$\mathbf{R}_{13}$	Parameter 3 ved funksjonskall	
$\mathbf{R}_{14}$	Parameter 4 ved funksjonskall	
$\mathbf{R}_{31}$	Returadresse ved funksjonskall	(gitt av Rask)

Tabell 5.1: Vår konvensjon for bruk av Rask-registre

⟨n⟩	⇒	`SET R1,R0,`⟨n⟩
-⟨n⟩	⇒	`SET R1,R0,`⟨n⟩ `SUB R1,R0,R1`
⟨v⟩	⇒	`LOAD R1,R0,Adr(`⟨v⟩`)`
⟨v⟩`[`⟨n⟩`]`	⇒	`SET R2,R0,`⟨n⟩ `LOAD R1,R2,Adr(`⟨v⟩`)`
⟨v⟩`[`⟨v_2⟩`]`	⇒	`LOAD R2,R0,Adr(`⟨v_2⟩`)` `LOAD R1,R2,Adr(`⟨v⟩`)`

Tabell 5.2: Oversettelse av operand i uttrykk

⟨e_1⟩ + ⟨e_2⟩	⇒	⟨Beregn ⟨e_1⟩ med svaret i $\mathbf{R}_1$⟩
		`SET R3,R1,0`
		⟨Beregn ⟨e_2⟩ med svaret i $\mathbf{R}_1$⟩
		`ADD R1,R3,R1`

Tabell 5.3: Oversettelse av operatoren +

⟨v⟩ = ⟨e⟩;	⇒	⟨Beregn ⟨e⟩ med svaret i $\mathbf{R}_1$ ⟩ STORE R1,R0,Adr(⟨v⟩)
⟨v⟩[⟨n⟩] = ⟨e⟩;	⇒	⟨Beregn ⟨e⟩ med svaret i $\mathbf{R}_1$ ⟩ SET R2,R0,⟨n⟩ STORE R1,R2,Adr(⟨v⟩)
⟨v⟩[⟨v_2⟩] = ⟨e⟩;	⇒	⟨Beregn ⟨e⟩ med svaret i $\mathbf{R}_1$ ⟩ LOAD R2,R0,⟨v⟨2⟩⟩ STORE R1,R2,Adr(⟨v⟩)

Tabell 5.4: Oversettelse av tilordning

if (⟨e⟩) {
⟨S⟩
}
	⇒	⟨Beregn ⟨e⟩ med svaret i $\mathbf{R}_1$ ⟩
		JUMPEQ R1,R0,Lx
		⟨S⟩
		Lx:

if (⟨e⟩) {
⟨S_1⟩
} else {
⟨S_2⟩
}
	⇒	⟨Beregn ⟨e⟩ med svaret i $\mathbf{R}_1$ ⟩
		JUMPEQ R1,R0,Le
		⟨S_1⟩
		JUMPEQ R0,R0,Lx
		Le: ⟨S_2⟩
		Lx:

Tabell 5.5: Oversettelse av if-setning

while (⟨e⟩) {
⟨S⟩
}
	⇒	Lw: ⟨Beregn ⟨e⟩ med svaret i $\mathbf{R}_1$ ⟩
		JUMPEQ R1,R0,Lx
		⟨S⟩
		JUMPEQ R0,R0,Lw
		Lx:

Tabell 5.6: Oversettelse av while-setning

5.6 Funksjoner

Selve funksjonen oversettes som vist i tabell 5.7 på neste side. Legg spesielt merke til følgende:

- Returadressen må gjemmes unna i tilfelle denne funksjonen kaller en annen funksjon.
- Av samme grunn plasseres parametrene i minnet.
- Register $\mathbf{R}_3$ må også gjemmes unna fordi det kan bli ødelagt ved beregningen av et uttrykk i denne funksjonen.

Følgelig settes det alltid av 2–6 celler for hver funksjon som deklareres i tillegg til eventuelle lokale variable.

`int` ⟨f⟩ `(int` ⟨p_1⟩`, ...)`
`{`
`  int` ⟨v_1⟩`, ...;`
 	
`  `⟨S⟩
`}`
	⇒	`fRet: RES  1`
		`fR3:  RES  1`
		`fP1:  RES  1`
		`  ⋮`
		`fV1:  RES  1`
		`  ⋮`
		`f:  STORE  R31,R0,fRet`
		`  STORE  R3,R0,fR3`
		`  STORE  R11,R0,fP1`
		`  ⋮`
		`  `⟨S⟩
		`fx:  LOAD  R3,R0,fR3`
		`  LOAD  R31,R0,fRet`
		`  RET`

Tabell 5.7: Oversettelse av funksjon

`return` ⟨e⟩`;`	⇒	⟨Beregn ⟨e⟩ med svaret i $\mathbf{R}_1$⟩ `JUMPEQ  R0,R0,fx`

Tabell 5.8: Oversettelse av return-setning

`f()`	⇒	`CALL  `⟨f⟩
`f(`⟨e_1⟩`,`⟨e_2⟩`,`⟨e_3⟩`,`⟨e_4⟩`)`	⇒	⟨Legg ⟨e_1⟩ i $\mathbf{R}_{11}$⟩ ⟨Legg ⟨e_2⟩ i $\mathbf{R}_{12}$⟩ ⟨Legg ⟨e_3⟩ i $\mathbf{R}_{13}$⟩ ⟨Legg ⟨e_4⟩ i $\mathbf{R}_{14}$⟩ `CALL  `⟨f⟩

Tabell 5.9: Oversettelse av funksjonskall

5.6.1 Return-setningen

Return-setningen gir koden vist i tabell 5.8. Den beregner returverdien og hopper til utgangen av funksjonen.

5.6.2 Funksjonskall

Oversettelse av funksjonskall kan man se i tabell 5.9; den er den samme uavhengig av om kallet er en egen setning eller et element i et uttrykk. For enkelhets skyld er det bare vist funksjonskall med 0 og 4 parametre.

Kapittel 6

Implementasjonen

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

Vi skal dele prosjektet vårt i pakkene vist i figur 6.1 på neste side. Siden Java kun tillater klasser i pakkene sine, vil vi alltid legge legge inn en klasse med samme navn[1] som pakken og som inneholder data og metoder som «hører hjemme i» pakken, spesielt de to metodene[2] `init` og `finish` som benyttes for initiering og terminering av modulene. Hver pakke vil altså se ut som vist i figur 6.2 på neste side.

6.1 Modulen Rusc

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

6.2 Modulen CharGenerator

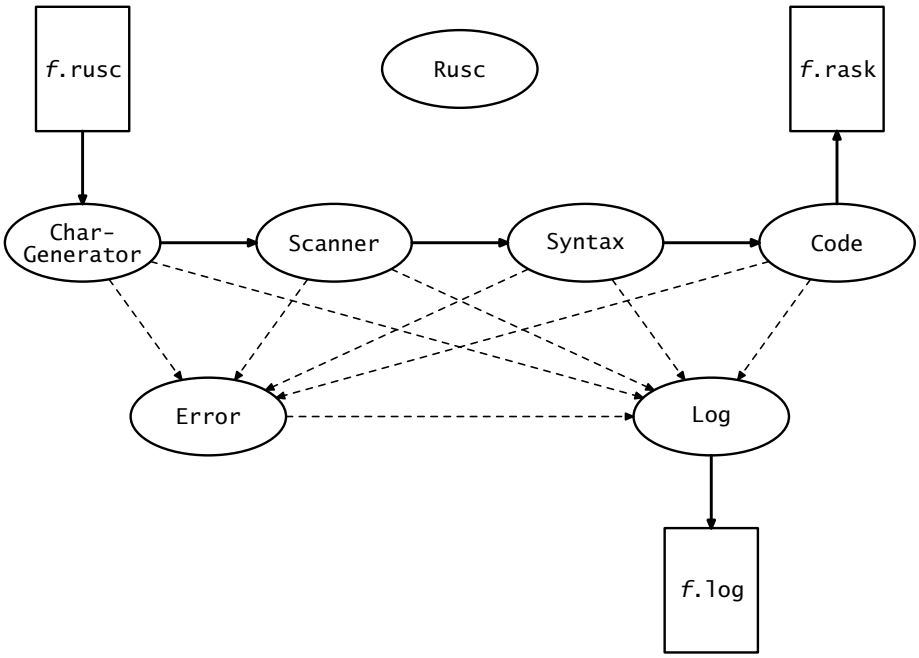
Denne modulen vil lese kildefilen linje for linje, ignorere #-linjer og sende den resterende kildekoden tegn for tegn videre i to variable: `curC` (med nåværende tegn) og `nextC` (med neste tegn). Metoden `readNext` vil klargjøre neste tegn.

6.3 Modulen Scanner

Denne modulen vil få tegn fra kildefilen (via `CharGenerator`) og levere fra seg symboler i variablene `curToken`, `nextToken` og `nextNextToken`; disse variablene er av klassen `Token`. `curToken` er det aktuelle symbolet vi

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

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



Figur 6.1: Modulene i kompilatoren

```
package no.uio.ifi.rusc.p;

public class P {
    public static ⟨data-deklasjon⟩;
    private static ⟨data-deklasjon⟩;

    public static void init() {
        ⋮
    }

    public static void finish() {
        ⋮
    }

    ⋮
}
```

Figur 6.2: Oppsett for de enkelte moduler

skal analysere, mens `nextToken` og `nextNextToken` er de to etterfølgende symbolene.

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

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

Når det ikke er flere symboler igjen på filen, vil de tre —`Token`-variablene få verdien `eofToken`.[3]

6.4 Modulen Syntax

Denne modulen tar seg av parseringen av RusC-programmet. Den inneholder klassen `SyntaxUnit` og diverse subklasser som brukes når parseringstreet skal bygges.

6.5 Modulen Code

Denne modulen tar seg av kodegenereringen. Den inneholder `genLoad`, `genSet` og tilsvarende metoder som genererer hver sin instruksjon. I tillegg har den `resMem` for å sette av plass til variable og `updateInstr` som oppdaterer adressedelen av en tidligere generert instruksjon.

Avslutningsmetoden `finish` skriver det ferdige programmet ut på en fil i Rasko-format.

6.6 Modulen Error

Denne modulen brukes for feilutskrifter. Den har én sentral metode `error` som skriver en feilmelding på skjermen og i loggfilen.

6.7 Modulen Log

Denne modulen tar seg av logging av informasjon. Den skriver data på filen øyeblikkelig og lukker filen etter hver linje slik at innholdet bevares om kompilatoren krasjer. Modulen har disse nyttige rutinene:

`logCode` logger informasjon om generert kode, men bare når `doLogCode` er satt.

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

`noteRes` lagrer data om reservering av plass til variable, men bare om `doLogCode` er satt.

`noteUpdate` legger til opplysninger om oppdateringer av tidligere genererte instruksjoner, men bare om `doLogCode` er satt.

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

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

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

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

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

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

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

`outdentTree` benyttes når et slikt nivå skal avsluttes.

Kapittel 7

Prosjektet

Som nevnt er RusC-kompilatoren et større program enn dere sannsynligvis har skrevet før, så prosjektet er delt i tre deler: en minimal introduksjonsdel og to omtrent like store restdeler, som vist i figur 7.1 på neste side.

På filen ~inf2100/oblig/base.zip (også tilgjengelig som http://www.ifi.uio.no/~inf2100/oblig/base.zip ligger rammen som *skal* brukes til løsningen. Lag en egen mappe til prosjektet deres og gjør så

```
> cd mappen
> copy ~inf2100/oblig/base.zip .
> unzip base.zip
> make
```

Som en hjelp under arbeidet, og for enkelt å sjekke om de ulike delene virker, skal de inneholde loggutskriftene vist i tabell 7.1 på neste side.

7.1 Del 0

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

```
> java -jar Rusc.jar -testscanner primes.rusc
```

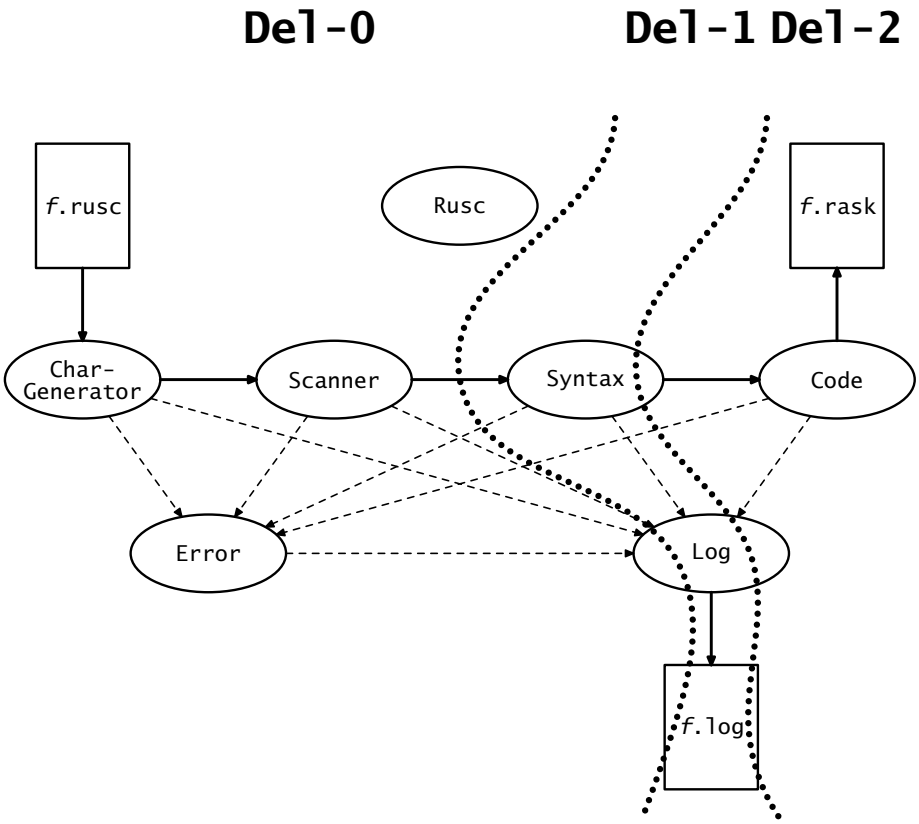
der testprogrammet er vist i figur 2.1 på side 24. Den resulterende filen `primes.log` skal da se ut som vist[1] i figurene 7.2 til 7.6 på side 52 og etterfølgende sider.

7.2 Del 1

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

```
> java -jar Rusc.jar -testparser primes.rusc
```

[1] Siden loggutskriften kommer fra to kilder (`CharGenerator` og `Scanner`), vil linjene fra disse kunne være blandet på en annen måte enn vist i dette kompendiet. Dette er helt normalt og klart akseptabelt.



Figur 7.1: De ulike delene i prosjektet

Opsjon	Del	Hva logges
`-logC`	Del 2	Hvilken Rask-kode genereres
`-logP`	Del 1	Hvilke parseringsmetoder kalles
`-logS`	Del 0	Hvilke symboler hentes fra skanneren
`-logT`	Del 1	Utskrift av parseringstreet

Tabell 7.1: Opsjoner for logging

vil loggfilen vise hvilke `parse`-rutiner som man er innom (opsjonen `-logP`, som settes automatisk av `-testparser`); som kontroll skrives den interne representasjonen av programmet ut (opsjonen `-logT`, som også settes av `-testparser`). Vårt vanlige testprogram vist i figur 2.1 på side 24 vil produsere loggfilen i figurene 7.7 til 7.15 på side 57 og etterfølgende.[2]

7.3 Del 2

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

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

- I mappen ~inf2100/oblig/test/ finnes fire RusC-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/ finnes diverse småprogrammer som alle inneholder en feil. kompilatoren din bør gi en tilsvarende feilemelding som referansekompilatoren.
- Når vi benytter opsjonen `-logC` på vårt standard testprogram (vist i figur 2.1 på side 24) skal gi loggfilen se ut som vist i figurene 7.16 til 7.18 på side 66 og deretter (med unntak av kommentarene på slutten av hver linje – de kan droppes eller se ut som du selv ønsker).

```
   1: /* Program 'primes'
   2:    ----------------
   3:    Finds all prime numbers up to 1000 (using the technique called
   4:    "the sieve of Eratosthenes") and prints them nicely formatted.
   5: */
   6:
   7: #include "/local/opt/inf2100/include/rusc.h"
   8:
   9: int prime[1001];  /* The sieve */
Scanner:  intToken
Scanner:  nameToken prime
Scanner:  leftBracketToken
Scanner:  numberToken 1001
Scanner:  rightBracketToken
Scanner:  semicolonToken
  10: int LF;           /* LF */
Scanner:  intToken
Scanner:  nameToken LF
Scanner:  semicolonToken
  11:
  12: int find_primes ()
Scanner:  intToken
Scanner:  nameToken find_primes
Scanner:  leftParToken
Scanner:  rightParToken
  13: {
Scanner:  leftCurlToken
  14:    /* Remove all non-primes from the sieve: */
  15:
  16:    int i1, i2;
Scanner:  intToken
Scanner:  nameToken i1
Scanner:  commaToken
Scanner:  nameToken i2
Scanner:  semicolonToken
  17:
  18:    i1 = 2;
Scanner:  nameToken i1
Scanner:  assignToken
Scanner:  numberToken 2
Scanner:  semicolonToken
  19:    while (i1 <= 1000) {
Scanner:  whileToken
Scanner:  leftParToken
Scanner:  nameToken i1
Scanner:  lessEqualToken
Scanner:  numberToken 1000
Scanner:  rightParToken
Scanner:  leftCurlToken
  20:       i2 = 2*i1;
Scanner:  nameToken i2
Scanner:  assignToken
Scanner:  numberToken 2
Scanner:  multiplyToken
Scanner:  nameToken i1
Scanner:  semicolonToken
  21:       while (i2 <= 1000) {
Scanner:  whileToken
Scanner:  leftParToken
Scanner:  nameToken i2
Scanner:  lessEqualToken
Scanner:  numberToken 1000
Scanner:  rightParToken
Scanner:  leftCurlToken
  22:    prime[i2] = 0;  i2 = i2+i1;
Scanner:  nameToken prime
Scanner:  leftBracketToken
Scanner:  nameToken i2
Scanner:  rightBracketToken
Scanner:  assignToken
Scanner:  numberToken 0
Scanner:  semicolonToken
Scanner:  nameToken i2
Scanner:  assignToken
Scanner:  nameToken i2
```

Figur 7.2: Loggfil som demonstrerer skanneren (del 1)

```
Scanner:  addToken
Scanner:  nameToken i1
Scanner:  semicolonToken
  23:       }
Scanner:  rightCurlToken
  24:       i1 = i1+1;
Scanner:  nameToken i1
Scanner:  assignToken
Scanner:  nameToken i1
Scanner:  addToken
Scanner:  numberToken 1
Scanner:  semicolonToken
  25:    }
Scanner:  rightCurlToken
  26: }
Scanner:  rightCurlToken
  27:
  28: int mod (int a, int b)
Scanner:  intToken
Scanner:  nameToken mod
Scanner:  leftParToken
Scanner:  intToken
Scanner:  nameToken a
Scanner:  commaToken
Scanner:  intToken
Scanner:  nameToken b
Scanner:  rightParToken
  29: {
Scanner:  leftCurlToken
  30:    /* Computes a%b. */
  31:
  32:    int ax;
Scanner:  intToken
Scanner:  nameToken ax
Scanner:  semicolonToken
  33:
  34:    ax = a/b;  ax = ax*b;
Scanner:  nameToken ax
Scanner:  assignToken
Scanner:  nameToken a
Scanner:  divideToken
Scanner:  nameToken b
Scanner:  semicolonToken
Scanner:  nameToken ax
Scanner:  assignToken
Scanner:  nameToken ax
Scanner:  multiplyToken
Scanner:  nameToken b
Scanner:  semicolonToken
  35:    return a - ax;
Scanner:  returnToken
Scanner:  nameToken a
Scanner:  subtractToken
Scanner:  nameToken ax
Scanner:  semicolonToken
  36: }
Scanner:  rightCurlToken
  37:
  38: int p3 (int v)
Scanner:  intToken
Scanner:  nameToken p3
Scanner:  leftParToken
Scanner:  intToken
Scanner:  nameToken v
Scanner:  rightParToken
  39: {
Scanner:  leftCurlToken
  40:    /* Does a ‘printf("%3d", v)´;
  41:       assumes 0<=v<=999. */
  42:
  43:    if (v <= 9) {
Scanner:  ifToken
Scanner:  leftParToken
Scanner:  nameToken v
Scanner:  lessEqualToken
```

Figur 7.3: Loggfil som demonstrerer skanneren (del 2)

```
151 Scanner:  numberToken 9
152 Scanner:  rightParToken
153 Scanner:  leftCurlToken
154   44:       putchar(' ');   putchar(' ');
155 Scanner:  nameToken putchar
156 Scanner:  leftParToken
157 Scanner:  numberToken 32
158 Scanner:  rightParToken
159 Scanner:  semicolonToken
160 Scanner:  nameToken putchar
161 Scanner:  leftParToken
162 Scanner:  numberToken 32
163 Scanner:  rightParToken
164 Scanner:  semicolonToken
165   45:    } else {
166 Scanner:  rightCurlToken
167 Scanner:  elseToken
168 Scanner:  leftCurlToken
169   46:       if (v <= 99) {
170 Scanner:  ifToken
171 Scanner:  leftParToken
172 Scanner:  nameToken v
173 Scanner:  lessEqualToken
174 Scanner:  numberToken 99
175 Scanner:  rightParToken
176 Scanner:  leftCurlToken
177   47:    putchar(' ');
178 Scanner:  nameToken putchar
179 Scanner:  leftParToken
180 Scanner:  numberToken 32
181 Scanner:  rightParToken
182 Scanner:  semicolonToken
183   48:       };
184 Scanner:  rightCurlToken
185 Scanner:  semicolonToken
186   49:    }
187 Scanner:  rightCurlToken
188   50:    putint(v);
189 Scanner:  nameToken putint
190 Scanner:  leftParToken
191 Scanner:  nameToken v
192 Scanner:  rightParToken
193 Scanner:  semicolonToken
194   51: }
195 Scanner:  rightCurlToken
196   52:
197   53: int print_primes ()
198 Scanner:  intToken
199 Scanner:  nameToken print_primes
200 Scanner:  leftParToken
201 Scanner:  rightParToken
202   54: {
203 Scanner:  leftCurlToken
204   55:    /* Print the primes, 10 on each line. */
205   56:
206   57:    int n_printed, i;
207 Scanner:  intToken
208 Scanner:  nameToken n_printed
209 Scanner:  commaToken
210 Scanner:  nameToken i
211 Scanner:  semicolonToken
212   58:
213   59:    n_printed = 0;  i = 1;
214 Scanner:  nameToken n_printed
215 Scanner:  assignToken
216 Scanner:  numberToken 0
217 Scanner:  semicolonToken
218 Scanner:  nameToken i
219 Scanner:  assignToken
220 Scanner:  numberToken 1
221 Scanner:  semicolonToken
222   60:    while (i <= 1000) {
223 Scanner:  whileToken
224 Scanner:  leftParToken
225 Scanner:  nameToken i
```

Figur 7.4: Loggfil som demonstrerer skanneren (del 3)

```
Scanner:  lessEqualToken
Scanner:  numberToken 1000
Scanner:  rightParToken
Scanner:  leftCurlToken
  61:       if (prime[i]) {
Scanner:  ifToken
Scanner:  leftParToken
Scanner:  nameToken prime
Scanner:  leftBracketToken
Scanner:  nameToken i
Scanner:  rightBracketToken
Scanner:  rightParToken
Scanner:  leftCurlToken
  62:    if (mod(n_printed,10) == 0 * n_printed) {
Scanner:  ifToken
Scanner:  leftParToken
Scanner:  nameToken mod
Scanner:  leftParToken
Scanner:  nameToken n_printed
Scanner:  commaToken
Scanner:  numberToken 10
Scanner:  rightParToken
Scanner:  equalToken
Scanner:  numberToken 0
Scanner:  multiplyToken
Scanner:  nameToken n_printed
Scanner:  rightParToken
Scanner:  leftCurlToken
  63:       putchar(LF);
Scanner:  nameToken putchar
Scanner:  leftParToken
Scanner:  nameToken LF
Scanner:  rightParToken
Scanner:  semicolonToken
  64:    }
Scanner:  rightCurlToken
  65:    putchar(' ');   p3(i);   n_printed = n_printed+1;
Scanner:  nameToken putchar
Scanner:  leftParToken
Scanner:  numberToken 32
Scanner:  rightParToken
Scanner:  semicolonToken
Scanner:  nameToken p3
Scanner:  leftParToken
Scanner:  nameToken i
Scanner:  rightParToken
Scanner:  semicolonToken
Scanner:  nameToken n_printed
Scanner:  assignToken
Scanner:  nameToken n_printed
Scanner:  addToken
Scanner:  numberToken 1
Scanner:  semicolonToken
  66:       }
Scanner:  rightCurlToken
  67:       i = i+1;
Scanner:  nameToken i
Scanner:  assignToken
Scanner:  nameToken i
Scanner:  addToken
Scanner:  numberToken 1
Scanner:  semicolonToken
  68:    }
Scanner:  rightCurlToken
  69:    putchar(LF);
Scanner:  nameToken putchar
Scanner:  leftParToken
Scanner:  nameToken LF
Scanner:  rightParToken
Scanner:  semicolonToken
  70: }
Scanner:  rightCurlToken
  71:
  72: int main ()
Scanner:  intToken
```

Figur 7.5: Loggfil som demonstrerer skanneren (del 4)

```
Scanner:  nameToken main
Scanner:  leftParToken
Scanner:  rightParToken
  73: {
Scanner:  leftCurlToken
  74:    int i;
Scanner:  intToken
Scanner:  nameToken i
Scanner:  semicolonToken
  75:
  76:    LF = 10;
Scanner:  nameToken LF
Scanner:  assignToken
Scanner:  numberToken 10
Scanner:  semicolonToken
  77:    /* Initialize the sieve by assuming all numbers >1 to be primes: */
  78:    prime[1] = 0;
Scanner:  nameToken prime
Scanner:  leftBracketToken
Scanner:  numberToken 1
Scanner:  rightBracketToken
Scanner:  assignToken
Scanner:  numberToken 0
Scanner:  semicolonToken
  79:    i = 2;
Scanner:  nameToken i
Scanner:  assignToken
Scanner:  numberToken 2
Scanner:  semicolonToken
  80:    while (i <= 1000) {
Scanner:  whileToken
Scanner:  leftParToken
Scanner:  nameToken i
Scanner:  lessEqualToken
Scanner:  numberToken 1000
Scanner:  rightParToken
Scanner:  leftCurlToken
  81:      prime[i] = 1;  i = i+1;
Scanner:  nameToken prime
Scanner:  leftBracketToken
Scanner:  nameToken i
Scanner:  rightBracketToken
Scanner:  assignToken
Scanner:  numberToken 1
Scanner:  semicolonToken
Scanner:  nameToken i
Scanner:  assignToken
Scanner:  nameToken i
Scanner:  addToken
Scanner:  numberToken 1
Scanner:  semicolonToken
  82:    }
Scanner:  rightCurlToken
  83:
  84:    /* Find and print the primes: */
  85:    find_primes();  print_primes();
Scanner:  nameToken find_primes
Scanner:  leftParToken
Scanner:  rightParToken
Scanner:  semicolonToken
Scanner:  nameToken print_primes
Scanner:  leftParToken
Scanner:  rightParToken
Scanner:  semicolonToken
  86: }
Scanner:  rightCurlToken
Scanner:  eofToken
Scanner:  eofToken
```

Figur 7.6: Loggfil som demonstrerer skanneren (del 5)

```
   1: /* Program 'primes'
   2:    ----------------
   3:    Finds all prime numbers up to 1000 (using the technique called
   4:    "the sieve of Eratosthenes") and prints them nicely formatted.
   5: */
   6:
   7: #include "/local/opt/inf2100/include/rusc.h"
   8:
   9: int prime[1001];  /* The sieve */
Parser:   <program>
Parser:     <var decl>
  10: int LF;           /* LF */
Parser:     </var decl>
Parser:     <var decl>
  11:
  12: int find_primes ()
Parser:     </var decl>
Parser:     <func decl>
  13: {
Parser:       <func params>
  14:    /* Remove all non-primes from the sieve: */
  15:
  16:    int i1, i2;
Parser:       </func params>
Parser:       <func body>
Parser:         <var decl>
  17:
  18:    i1 = 2;
Parser:         </var decl>
Parser:         <statm list>
Parser:           <statement>
Parser:             <assignment>
Parser:               <variable>
Parser:               </variable>
  19:    while (i1 <= 1000) {
Parser:               <expression>
Parser:                 <simple expr>
Parser:                   <number>
Parser:                   </number>
Parser:                 </simple expr>
Parser:               </expression>
Parser:             </assignment>
Parser:           </statement>
Parser:           <statement>
Parser:             <while-statm>
Parser:               <expression>
Parser:                 <simple expr>
Parser:                   <variable>
Parser:                   </variable>
Parser:                 </simple expr>
Parser:                 <simple expr>
Parser:                   <number>
  20:       i2 = 2*i1;
Parser:                   </number>
Parser:                 </simple expr>
Parser:               </expression>
Parser:               <statm list>
Parser:                 <statement>
Parser:                   <assignment>
Parser:                     <variable>
Parser:                     </variable>
Parser:                     <expression>
Parser:                       <simple expr>
Parser:                         <number>
Parser:                         </number>
Parser:                       </simple expr>
  21:       while (i2 <= 1000) {
Parser:                       <simple expr>
Parser:                         <variable>
Parser:                         </variable>
Parser:                       </simple expr>
Parser:                     </expression>
Parser:                   </assignment>
Parser:                 </statement>
Parser:                 <statement>
Parser:                   <while-statm>
Parser:                     <expression>
Parser:                       <simple expr>
Parser:                         <variable>
Parser:                         </variable>
```

Figur 7.7: Loggfil som demonstrerer parseren (del 1)

```
Parser:                        </simple expr>
Parser:                        <simple expr>
Parser:                          <number>
  22:    prime[i2] = 0;  i2 = i2+i1;
Parser:                          </number>
Parser:                        </simple expr>
Parser:                      </expression>
Parser:                      <statm list>
Parser:                        <statement>
Parser:                          <assignment>
Parser:                            <variable>
Parser:                              <simple expr>
Parser:                                <variable>
Parser:                                </variable>
Parser:                              </simple expr>
Parser:                            </variable>
Parser:                            <expression>
Parser:                              <simple expr>
Parser:                                <number>
Parser:                                </number>
Parser:                              </simple expr>
Parser:                            </expression>
Parser:                          </assignment>
Parser:                        </statement>
Parser:                        <statement>
Parser:                          <assignment>
Parser:                            <variable>
Parser:                            </variable>
Parser:                            <expression>
Parser:                              <simple expr>
Parser:                                <variable>
Parser:                                </variable>
Parser:                              </simple expr>
  23:       }
Parser:                              <simple expr>
Parser:                                <variable>
  24:       i1 = i1+1;
Parser:                                </variable>
Parser:                              </simple expr>
Parser:                            </expression>
Parser:                          </assignment>
Parser:                        </statement>
Parser:                      </statm list>
Parser:                    </while-statm>
Parser:                  </statement>
Parser:                  <statement>
Parser:                    <assignment>
Parser:                      <variable>
Parser:                      </variable>
Parser:                      <expression>
Parser:                        <simple expr>
Parser:                          <variable>
Parser:                          </variable>
Parser:                        </simple expr>
  25:    }
Parser:                        <simple expr>
Parser:                          <number>
  26: }
Parser:                          </number>
Parser:                        </simple expr>
Parser:                      </expression>
  27:
  28: int mod (int a, int b)
Parser:                    </assignment>
Parser:                  </statement>
Parser:                </statm list>
Parser:              </while-statm>
Parser:            </statement>
Parser:          </statm list>
Parser:        </func body>
Parser:      </func decl>
Parser:      <func decl>
Parser:        <func params>
  29: {
  30:    /* Computes a%b. */
  31:
  32:    int ax;
Parser:        </func params>
Parser:        <func body>
Parser:          <var decl>
```

Figur 7.8: Loggfil som demonstrerer parseren (del 2)

```
  33:
  34:    ax = a/b;  ax = ax*b;
Parser:         </var decl>
Parser:         <statm list>
Parser:           <statement>
Parser:             <assignment>
Parser:               <variable>
Parser:               </variable>
Parser:               <expression>
Parser:                 <simple expr>
Parser:                   <variable>
Parser:                   </variable>
Parser:                 </simple expr>
Parser:                 <simple expr>
Parser:                   <variable>
Parser:                   </variable>
Parser:                 </simple expr>
Parser:               </expression>
Parser:             </assignment>
Parser:           </statement>
Parser:           <statement>
Parser:             <assignment>
Parser:               <variable>
Parser:               </variable>
Parser:               <expression>
Parser:                 <simple expr>
Parser:                   <variable>
Parser:                   </variable>
Parser:                 </simple expr>
  35:    return a - ax;
Parser:                 <simple expr>
Parser:                   <variable>
Parser:                   </variable>
Parser:                 </simple expr>
Parser:               </expression>
Parser:             </assignment>
Parser:           </statement>
Parser:           <statement>
Parser:             <return statm>
Parser:               <expression>
Parser:                 <simple expr>
Parser:                   <variable>
Parser:                   </variable>
Parser:                 </simple expr>
  36: }
Parser:                 <simple expr>
Parser:                   <variable>
  37:
  38: int p3 (int v)
Parser:                   </variable>
Parser:                 </simple expr>
Parser:               </expression>
Parser:             </return statm>
Parser:           </statement>
Parser:         </statm list>
Parser:       </func body>
Parser:     </func decl>
Parser:     <func decl>
Parser:       <func params>
  39: {
  40:    /* Does a ‘printf("%3d", v)´;
  41:       assumes 0<=v<=999. */
  42:
  43:    if (v <= 9) {
Parser:       </func params>
Parser:       <func body>
Parser:         <statm list>
Parser:           <statement>
Parser:             <if-statm>
Parser:               <expression>
Parser:                 <simple expr>
Parser:                   <variable>
Parser:                   </variable>
Parser:                 </simple expr>
Parser:                 <simple expr>
Parser:                   <number>
  44:       putchar(’ ’);   putchar(’ ’);
Parser:                   </number>
Parser:                 </simple expr>
Parser:               </expression>
```

Figur 7.9: Loggfil som demonstrerer parseren (del 3)

```
241 Parser:                <statm list>
242 Parser:                  <statement>
243 Parser:                    <func call as statement>
244 Parser:                      <function call>
245 Parser:                        <simple expr>
246 Parser:                          <number>
247 Parser:                          </number>
248 Parser:                        </simple expr>
249 Parser:                      </function call>
250 Parser:                    </func call as statement>
251 Parser:                  </statement>
252 Parser:                  <statement>
253 Parser:                    <func call as statement>
254 Parser:                      <function call>
255 Parser:                        <simple expr>
256 Parser:                          <number>
257   45:    } else {
258 Parser:                          </number>
259 Parser:                        </simple expr>
260 Parser:                      </function call>
261 Parser:                    </func call as statement>
262 Parser:                  </statement>
263 Parser:                </statm list>
264   46:       if (v <= 99) {
265 Parser:                <else-part>
266 Parser:                  <statm list>
267 Parser:                    <statement>
268 Parser:                      <if-statm>
269 Parser:                        <expression>
270 Parser:                          <simple expr>
271 Parser:                            <variable>
272 Parser:                            </variable>
273 Parser:                          </simple expr>
274 Parser:                          <simple expr>
275 Parser:                            <number>
276   47:    putchar(' ');
277 Parser:                            </number>
278 Parser:                          </simple expr>
279 Parser:                        </expression>
280 Parser:                        <statm list>
281 Parser:                          <statement>
282 Parser:                            <func call as statement>
283 Parser:                              <function call>
284 Parser:                                <simple expr>
285 Parser:                                  <number>
286   48:       };
287 Parser:                                  </number>
288 Parser:                                </simple expr>
289 Parser:                              </function call>
290   49:    }
291 Parser:                            </func call as statement>
292 Parser:                          </statement>
293 Parser:                        </statm list>
294   50:    putint(v);
295 Parser:                      </if-statm>
296 Parser:                    </statement>
297 Parser:                    <statement>
298 Parser:                      <empty statm>
299 Parser:                      </empty statm>
300 Parser:                    </statement>
301 Parser:                  </statm list>
302 Parser:                </else-part>
303 Parser:              </if-statm>
304 Parser:            </statement>
305 Parser:            <statement>
306 Parser:              <func call as statement>
307 Parser:                <function call>
308 Parser:                  <simple expr>
309 Parser:                    <variable>
310   51: }
311 Parser:                    </variable>
312 Parser:                  </simple expr>
313   52:
314   53: int print_primes ()
315 Parser:                </function call>
316 Parser:              </func call as statement>
317 Parser:            </statement>
318 Parser:          </statm list>
319 Parser:        </func body>
320 Parser:      </func decl>
```

Figur 7.10: Loggfil som demonstrerer parseren (del 4)

```
Parser:     <func decl>
  54: {
Parser:       <func params>
  55:    /* Print the primes, 10 on each line. */
  56:
  57:    int n_printed, i;
Parser:       </func params>
Parser:       <func body>
Parser:         <var decl>
  58:
  59:    n_printed = 0;  i = 1;
Parser:         </var decl>
Parser:         <statm list>
Parser:           <statement>
Parser:             <assignment>
Parser:               <variable>
Parser:               </variable>
Parser:               <expression>
Parser:                 <simple expr>
Parser:                   <number>
Parser:                   </number>
Parser:                 </simple expr>
Parser:               </expression>
Parser:             </assignment>
Parser:           </statement>
Parser:           <statement>
Parser:             <assignment>
Parser:               <variable>
Parser:               </variable>
  60:    while (i <= 1000) {
Parser:               <expression>
Parser:                 <simple expr>
Parser:                   <number>
Parser:                   </number>
Parser:                 </simple expr>
Parser:               </expression>
Parser:             </assignment>
Parser:           </statement>
Parser:           <statement>
Parser:             <while-statm>
Parser:               <expression>
Parser:                 <simple expr>
Parser:                   <variable>
Parser:                   </variable>
Parser:                 </simple expr>
Parser:                 <simple expr>
Parser:                   <number>
  61:       if (prime[i]) {
Parser:                   </number>
Parser:                 </simple expr>
Parser:               </expression>
Parser:               <statm list>
Parser:                 <statement>
Parser:                   <if-statm>
Parser:                     <expression>
Parser:                       <variable>
Parser:                         <simple expr>
Parser:                           <variable>
Parser:                           </variable>
Parser:                         </simple expr>
  62:    if (mod(n_printed,10) == 0 * n_printed) {
Parser:                       </variable>
Parser:                     </expression>
Parser:                     <statm list>
Parser:                       <statement>
Parser:                         <if-statm>
Parser:                           <expression>
Parser:                             <function call>
Parser:                               <simple expr>
Parser:                                 <variable>
Parser:                                 </variable>
Parser:                               </simple expr>
Parser:                               <simple expr>
Parser:                                 <number>
Parser:                                 </number>
Parser:                               </simple expr>
Parser:                             </function call>
Parser:                             <simple expr>
Parser:                               <number>
Parser:                               </number>
```

Figur 7.11: Loggfil som demonstrerer parseren (del 5)

```
401 Parser:                              </simple expr>
402 Parser:                              <simple expr>
403 Parser:                                <variable>
404   63:       putchar(LF);
405 Parser:                                </variable>
406 Parser:                              </simple expr>
407 Parser:                            </expression>
408 Parser:                            <statm list>
409 Parser:                              <statement>
410 Parser:                                <func call as statement>
411 Parser:                                  <function call>
412 Parser:                                    <simple expr>
413 Parser:                                      <variable>
414   64:    }
415 Parser:                                      </variable>
416 Parser:                                    </simple expr>
417   65:    putchar(' ');   p3(i);   n_printed = n_printed+1;
418 Parser:                                  </function call>
419 Parser:                                </func call as statement>
420 Parser:                              </statement>
421 Parser:                            </statm list>
422 Parser:                          </if-statm>
423 Parser:                        </statement>
424 Parser:                        <statement>
425 Parser:                          <func call as statement>
426 Parser:                            <function call>
427 Parser:                              <simple expr>
428 Parser:                                <number>
429 Parser:                                </number>
430 Parser:                              </simple expr>
431 Parser:                            </function call>
432 Parser:                          </func call as statement>
433 Parser:                        </statement>
434 Parser:                        <statement>
435 Parser:                          <func call as statement>
436 Parser:                            <function call>
437 Parser:                              <simple expr>
438 Parser:                                <variable>
439 Parser:                                </variable>
440 Parser:                              </simple expr>
441 Parser:                            </function call>
442 Parser:                          </func call as statement>
443 Parser:                        </statement>
444 Parser:                        <statement>
445 Parser:                          <assignment>
446 Parser:                            <variable>
447 Parser:                            </variable>
448 Parser:                            <expression>
449 Parser:                              <simple expr>
450 Parser:                                <variable>
451 Parser:                                </variable>
452 Parser:                              </simple expr>
453   66:       }
454 Parser:                              <simple expr>
455 Parser:                                <number>
456   67:       i = i+1;
457 Parser:                                </number>
458 Parser:                              </simple expr>
459 Parser:                            </expression>
460 Parser:                          </assignment>
461 Parser:                        </statement>
462 Parser:                      </statm list>
463 Parser:                    </if-statm>
464 Parser:                  </statement>
465 Parser:                  <statement>
466 Parser:                    <assignment>
467 Parser:                      <variable>
468 Parser:                      </variable>
469 Parser:                      <expression>
470 Parser:                        <simple expr>
471 Parser:                          <variable>
472 Parser:                          </variable>
473 Parser:                        </simple expr>
474   68:    }
475 Parser:                        <simple expr>
476 Parser:                          <number>
477   69:    putchar(LF);
478 Parser:                          </number>
479 Parser:                        </simple expr>
480 Parser:                      </expression>
```

Figur 7.12: Loggfil som demonstrerer parseren (del 6)

```
481 Parser:                    </assignment>
482 Parser:                  </statement>
483 Parser:                </statm list>
484 Parser:              </while-statm>
485 Parser:            </statement>
486 Parser:            <statement>
487 Parser:              <func call as statement>
488 Parser:                <function call>
489 Parser:                  <simple expr>
490 Parser:                    <variable>
491   70: }
492 Parser:                    </variable>
493 Parser:                  </simple expr>
494   71:
495   72: int main ()
496 Parser:                </function call>
497 Parser:              </func call as statement>
498 Parser:            </statement>
499 Parser:          </statm list>
500 Parser:        </func body>
501 Parser:      </func decl>
502 Parser:      <func decl>
503   73: {
504 Parser:        <func params>
505   74:    int i;
506 Parser:        </func params>
507 Parser:        <func body>
508 Parser:          <var decl>
509   75:
510   76:    LF = 10;
511 Parser:          </var decl>
512 Parser:          <statm list>
513 Parser:            <statement>
514 Parser:              <assignment>
515 Parser:                <variable>
516 Parser:                </variable>
517   77:    /* Initialize the sieve by assuming all numbers >1 to be primes: */
518   78:    prime[1] = 0;
519 Parser:                <expression>
520 Parser:                  <simple expr>
521 Parser:                    <number>
522 Parser:                    </number>
523 Parser:                  </simple expr>
524 Parser:                </expression>
525 Parser:              </assignment>
526 Parser:            </statement>
527 Parser:            <statement>
528 Parser:              <assignment>
529 Parser:                <variable>
530 Parser:                  <simple expr>
531 Parser:                    <number>
532 Parser:                    </number>
533 Parser:                  </simple expr>
534 Parser:                </variable>
535   79:    i = 2;
536 Parser:                <expression>
537 Parser:                  <simple expr>
538 Parser:                    <number>
539 Parser:                    </number>
540 Parser:                  </simple expr>
541 Parser:                </expression>
542 Parser:              </assignment>
543 Parser:            </statement>
544 Parser:            <statement>
545 Parser:              <assignment>
546 Parser:                <variable>
547 Parser:                </variable>
548   80:    while (i <= 1000) {
549 Parser:                <expression>
550 Parser:                  <simple expr>
551 Parser:                    <number>
552 Parser:                    </number>
553 Parser:                  </simple expr>
554 Parser:                </expression>
555 Parser:              </assignment>
556 Parser:            </statement>
557 Parser:            <statement>
558 Parser:              <while-statm>
559 Parser:                <expression>
560 Parser:                  <simple expr>
```

Figur 7.13: Loggfil som demonstrerer parseren (del 7)

```
561 Parser:                    <variable>
562 Parser:                    </variable>
563 Parser:                  </simple expr>
564 Parser:                  <simple expr>
565 Parser:                    <number>
566   81:       prime[i] = 1;  i = i+1;
567 Parser:                    </number>
568 Parser:                  </simple expr>
569 Parser:                </expression>
570 Parser:                <statm list>
571 Parser:                  <statement>
572 Parser:                    <assignment>
573 Parser:                      <variable>
574 Parser:                        <simple expr>
575 Parser:                          <variable>
576 Parser:                          </variable>
577 Parser:                        </simple expr>
578 Parser:                      </variable>
579 Parser:                      <expression>
580 Parser:                        <simple expr>
581 Parser:                          <number>
582 Parser:                          </number>
583 Parser:                        </simple expr>
584 Parser:                      </expression>
585 Parser:                    </assignment>
586 Parser:                  </statement>
587 Parser:                  <statement>
588 Parser:                    <assignment>
589 Parser:                      <variable>
590 Parser:                      </variable>
591 Parser:                      <expression>
592 Parser:                        <simple expr>
593 Parser:                          <variable>
594 Parser:                          </variable>
595 Parser:                        </simple expr>
596   82:    }
597 Parser:                        <simple expr>
598 Parser:                          <number>
599   83:
600   84:    /* Find and print the primes: */
601   85:    find_primes();  print_primes();
602 Parser:                          </number>
603 Parser:                        </simple expr>
604 Parser:                      </expression>
605 Parser:                    </assignment>
606 Parser:                  </statement>
607 Parser:                </statm list>
608 Parser:              </while-statm>
609 Parser:            </statement>
610 Parser:            <statement>
611 Parser:              <func call as statement>
612 Parser:                <function call>
613 Parser:                </function call>
614 Parser:              </func call as statement>
615 Parser:            </statement>
616 Parser:            <statement>
617 Parser:              <func call as statement>
618 Parser:                <function call>
619   86: }
620 Parser:                </function call>
621 Parser:              </func call as statement>
622 Parser:            </statement>
623 Parser:          </statm list>
624 Parser:        </func body>
625 Parser:      </func decl>
626 Parser:    </program>
```

Figur 7.14: Loggfil som demonstrerer parseren (del 8)

```
627 Tree:     int prime[1001];
628 Tree:
629 Tree:     int LF;
630 Tree:
631 Tree:     func find_primes ()
632 Tree:     {
633 Tree:       int i1;
634 Tree:       int i2;
635 Tree:
636 Tree:       i1 = 2;
637 Tree:       while (i1 <= 1000) {
638 Tree:         i2 = 2 * i1;
639 Tree:         while (i2 <= 1000) {
640 Tree:           prime[i2] = 0;
641 Tree:           i2 = i2 + i1;
642 Tree:         }
643 Tree:         i1 = i1 + 1;
644 Tree:       }
645 Tree:     }
646 Tree:
647 Tree:     func mod (int a,int b)
648 Tree:     {
649 Tree:       int ax;
650 Tree:
651 Tree:       ax = a / b;
652 Tree:       ax = ax * b;
653 Tree:       return a - ax;
654 Tree:     }
655 Tree:
656 Tree:     func p3 (int v)
657 Tree:     {
658 Tree:       if (v <= 9) {
659 Tree:         putchar(32);
660 Tree:         putchar(32);
661 Tree:       } else {
662 Tree:         if (v <= 99) {
663 Tree:           putchar(32);
664 Tree:         }
665 Tree:         ;
666 Tree:       }
667 Tree:       putint(v);
668 Tree:     }
669 Tree:
670 Tree:     func print_primes ()
671 Tree:     {
672 Tree:       int n_printed;
673 Tree:       int i;
674 Tree:
675 Tree:       n_printed = 0;
676 Tree:       i = 1;
677 Tree:       while (i <= 1000) {
678 Tree:         if (prime[i]) {
679 Tree:           if (mod(n_printed,10) == 0 * n_printed) {
680 Tree:             putchar(LF);
681 Tree:           }
682 Tree:           putchar(32);
683 Tree:           p3(i);
684 Tree:           n_printed = n_printed + 1;
685 Tree:         }
686 Tree:         i = i + 1;
687 Tree:       }
688 Tree:       putchar(LF);
689 Tree:     }
690 Tree:
691 Tree:     func main ()
692 Tree:     {
693 Tree:       int i;
694 Tree:
695 Tree:       LF = 10;
696 Tree:       prime[1] = 0;
697 Tree:       i = 2;
698 Tree:       while (i <= 1000) {
699 Tree:         prime[i] = 1;
700 Tree:         i = i + 1;
701 Tree:       }
702 Tree:       find_primes();
703 Tree:       print_primes();
704 Tree:     }
```

Figur 7.15: Loggfil som demonstrerer utskrift av det parserte treet

```
Code    0: 1600000000000000  CALL      0  0          0  # <Dummy main program>
Code    1:  211000000000000  SET      11  0          0  #     (0)
Code    2: 1600000000009990  CALL      0  0       9990  # exit
Code    3:                   RES                   1001  # int prime[1001]
Code 1004:                   RES                      1  # int LF
Code 1005:                   RES                      1  # <return address> in find_primes
Code 1006:                   RES                      1  # <refuge for R3> in find_primes
Code 1007:                   RES                      1  # int i1
Code 1008:                   RES                      1  # int i2
Code 1009:  331000000001005  STORE    31  0       1005  # <save return address>
Code 1010:  303000000001006  STORE     3  0       1006  # <save R3>
Code 1011:  201000000000002  SET       1  0          2  # R1 = 2
Code 1012:  301000000001007  STORE     1  0       1007  # i1 =
Code 1013:  101000000001007  LOAD      1  0       1007  # R1 = i1
Code 1014:  203010000000000  SET       3  1          0  # <save operand>
Code 1015:  201000000001000  SET       1  0       1000  # R1 = 1000
Code 1016: 1101030000000001  LESSEQ    1  3          1  # <=
Code 1017: 1401000000000000  JUMPEQ    1  0          0  # break while if != 0
Code 1018:  201000000000002  SET       1  0          2  # R1 = 2
Code 1019:  203010000000000  SET       3  1          0  # <save operand>
Code 1020:  101000000001007  LOAD      1  0       1007  # R1 = i1
Code 1021:  601030000000001  MUL       1  3          1  # *
Code 1022:  301000000001008  STORE     1  0       1008  # i2 =
Code 1023:  101000000001008  LOAD      1  0       1008  # R1 = i2
Code 1024:  203010000000000  SET       3  1          0  # <save operand>
Code 1025:  201000000001000  SET       1  0       1000  # R1 = 1000
Code 1026: 1101030000000001  LESSEQ    1  3          1  # <=
Code 1027: 1401000000000000  JUMPEQ    1  0          0  # break while if != 0
Code 1028:  201000000000000  SET       1  0          0  # R1 = 0
Code 1029:  102000000001008  LOAD      2  0       1008  # R2 = i2
Code 1030:  301020000000003  STORE     1  2          3  # prime[...] =
Code 1031:  101000000001008  LOAD      1  0       1008  # R1 = i2
Code 1032:  203010000000000  SET       3  1          0  # <save operand>
Code 1033:  101000000001007  LOAD      1  0       1007  # R1 = i1
Code 1034:  401030000000001  ADD       1  3          1  # +
Code 1035:  301000000001008  STORE     1  0       1008  # i2 =
Code 1036: 1400000000001023  JUMPEQ    0  0       1023  # continue while
---> 1027: 1401000000001037                       1037  # <update break address>
Code 1037:  101000000001007  LOAD      1  0       1007  # R1 = i1
Code 1038:  203010000000000  SET       3  1          0  # <save operand>
Code 1039:  201000000000001  SET       1  0          1  # R1 = 1
Code 1040:  401030000000001  ADD       1  3          1  # +
Code 1041:  301000000001007  STORE     1  0       1007  # i1 =
Code 1042: 1400000000001013  JUMPEQ    0  0       1013  # continue while
---> 1017: 1401000000001043                       1043  # <update break address>
Code 1043:  103000000001006  LOAD      3  0       1006  # <restore R3>
Code 1044:  131000000001005  LOAD     31  0       1005  # <restore return address>
Code 1045: 1700000000000000  RET       0  0          0  # return (from find_primes)
Code 1046:                   RES                      1  # <return address> in mod
Code 1047:                   RES                      1  # <refuge for R3> in mod
Code 1048:                   RES                      1  # int a
Code 1049:                   RES                      1  # int b
Code 1050:                   RES                      1  # int ax
Code 1051:  331000000001046  STORE    31  0       1046  # <save return address>
Code 1052:  303000000001047  STORE     3  0       1047  # <save R3>
Code 1053:  311000000001048  STORE    11  0       1048  # <save parameter a>
Code 1054:  312000000001049  STORE    12  0       1049  # <save parameter b>
Code 1055:  101000000001048  LOAD      1  0       1048  # R1 = a
Code 1056:  203010000000000  SET       3  1          0  # <save operand>
Code 1057:  101000000001049  LOAD      1  0       1049  # R1 = b
Code 1058:  701030000000001  DIV       1  3          1  # /
Code 1059:  301000000001050  STORE     1  0       1050  # ax =
Code 1060:  101000000001050  LOAD      1  0       1050  # R1 = ax
Code 1061:  203010000000000  SET       3  1          0  # <save operand>
Code 1062:  101000000001049  LOAD      1  0       1049  # R1 = b
Code 1063:  601030000000001  MUL       1  3          1  # *
Code 1064:  301000000001050  STORE     1  0       1050  # ax =
Code 1065:  101000000001048  LOAD      1  0       1048  # R1 = a
Code 1066:  203010000000000  SET       3  1          0  # <save operand>
Code 1067:  101000000001050  LOAD      1  0       1050  # R1 = ax
Code 1068:  501030000000001  SUB       1  3          1  # -
Code 1069: 1400000000000000  JUMPEQ    0  0          0  # return
---> 1069: 1400000000001070                       1070  # <update address of return>
Code 1070:  103000000001047  LOAD      3  0       1047  # <restore R3>
Code 1071:  131000000001046  LOAD     31  0       1046  # <restore return address>
```

Figur 7.16: Loggfil som demonstrerer kodegenereringen (del 1)

```
Code 1072: 1700000000000000  RET       0  0          0  # return (from mod)
Code 1073:                   RES                     1  # <return address> in p3
Code 1074:                   RES                     1  # <refuge for R3> in p3
Code 1075:                   RES                     1  # int v
Code 1076:  331000000001073  STORE    31  0       1073  # <save return address>
Code 1077:  303000000001074  STORE     3  0       1074  # <save R3>
Code 1078:  311000000001075  STORE    11  0       1075  # <save parameter v>
Code 1079:  101000000001075  LOAD      1  0       1075  # R1 = v
Code 1080:  203010000000000  SET       3  1          0  # <save operand>
Code 1081:  201000000000009  SET       1  0          9  # R1 = 9
Code 1082: 1101030000000001  LESSEQ    1  3          1  # <=
Code 1083: 1401000000000000  JUMPEQ    1  0          0  # if false, jump
Code 1084:  211000000000032  SET      11  0         32  # R11 = 32
Code 1085: 1600000000009993  CALL      0  0       9993  # putchar(...)
Code 1086:  211000000000032  SET      11  0         32  # R11 = 32
Code 1087: 1600000000009993  CALL      0  0       9993  # putchar(...)
Code 1088: 1400000000000000  JUMPEQ    0  0          0  # if: end of true part
---> 1083: 1401000000001089                        1089  # <update if-address 1>
Code 1089:  101000000001075  LOAD      1  0       1075  # R1 = v
Code 1090:  203010000000000  SET       3  1          0  # <save operand>
Code 1091:  201000000000099  SET       1  0         99  # R1 = 99
Code 1092: 1101030000000001  LESSEQ    1  3          1  # <=
Code 1093: 1401000000000000  JUMPEQ    1  0          0  # if false, jump
Code 1094:  211000000000032  SET      11  0         32  # R11 = 32
Code 1095: 1600000000009993  CALL      0  0       9993  # putchar(...)
---> 1093: 1401000000001096                        1096  # <update if-address>
---> 1088: 1400000000001096                        1096  # <update if-address 2>
Code 1096:  111000000001075  LOAD     11  0       1075  # R11 = v
Code 1097: 1600000000009994  CALL      0  0       9994  # putint(...)
Code 1098:  103000000001074  LOAD      3  0       1074  # <restore R3>
Code 1099:  131000000001073  LOAD     31  0       1073  # <restore return address>
Code 1100: 1700000000000000  RET       0  0          0  # return (from p3)
Code 1101:                   RES                     1  # <return address> in print_primes
Code 1102:                   RES                     1  # <refuge for R3> in print_primes
Code 1103:                   RES                     1  # int n_printed
Code 1104:                   RES                     1  # int i
Code 1105:  331000000001101  STORE    31  0       1101  # <save return address>
Code 1106:  303000000001102  STORE     3  0       1102  # <save R3>
Code 1107:  201000000000000  SET       1  0          0  # R1 = 0
Code 1108:  301000000001103  STORE     1  0       1103  # n_printed =
Code 1109:  201000000000001  SET       1  0          1  # R1 = 1
Code 1110:  301000000001104  STORE     1  0       1104  # i =
Code 1111:  101000000001104  LOAD      1  0       1104  # R1 = i
Code 1112:  203010000000000  SET       3  1          0  # <save operand>
Code 1113:  201000000001000  SET       1  0       1000  # R1 = 1000
Code 1114: 1101030000000001  LESSEQ    1  3          1  # <=
Code 1115: 1401000000000000  JUMPEQ    1  0          0  # break while if != 0
Code 1116:  102000000001104  LOAD      2  0       1104  # R2 = i
Code 1117:  101020000000003  LOAD      1  2          3  # <load variable>
Code 1118: 1401000000000000  JUMPEQ    1  0          0  # if false, jump
Code 1119:  111000000001103  LOAD     11  0       1103  # R11 = n_printed
Code 1120:  212000000000010  SET      12  0         10  # R12 = 10
Code 1121: 1600000000001051  CALL      0  0       1051  # mod(...)
Code 1122:  203010000000000  SET       3  1          0  # <save operand>
Code 1123:  201000000000000  SET       1  0          0  # R1 = 0
Code 1124:  801030000000001  EQ        1  3          1  # ==
Code 1125:  203010000000000  SET       3  1          0  # <save operand>
Code 1126:  101000000001103  LOAD      1  0       1103  # R1 = n_printed
Code 1127:  601030000000001  MUL       1  3          1  # *
Code 1128: 1401000000000000  JUMPEQ    1  0          0  # if false, jump
Code 1129:  111000000001004  LOAD     11  0       1004  # R11 = LF
Code 1130: 1600000000009993  CALL      0  0       9993  # putchar(...)
---> 1128: 1401000000001131                        1131  # <update if-address>
Code 1131:  211000000000032  SET      11  0         32  # R11 = 32
Code 1132: 1600000000009993  CALL      0  0       9993  # putchar(...)
Code 1133:  111000000001104  LOAD     11  0       1104  # R11 = i
Code 1134: 1600000000001076  CALL      0  0       1076  # p3(...)
Code 1135:  101000000001103  LOAD      1  0       1103  # R1 = n_printed
Code 1136:  203010000000000  SET       3  1          0  # <save operand>
Code 1137:  201000000000001  SET       1  0          1  # R1 = 1
Code 1138:  401030000000001  ADD       1  3          1  # +
Code 1139:  301000000001103  STORE     1  0       1103  # n_printed =
---> 1118: 1401000000001140                        1140  # <update if-address>
Code 1140:  101000000001104  LOAD      1  0       1104  # R1 = i
Code 1141:  203010000000000  SET       3  1          0  # <save operand>
```

Figur 7.17: Loggfil som demonstrerer kodegenereringen (del 2)

```
Code 1142:  201000000000001  SET       1  0          1  # R1 = 1
Code 1143:  401030000000001  ADD       1  3          1  # +
Code 1144:  301000000001104  STORE     1  0       1104  # i =
Code 1145: 1400000000001111  JUMPEQ    0  0       1111  # continue while
---> 1115: 1401000000001146                       1146  # <update break address>
Code 1146:  111000000001004  LOAD     11  0       1004  # R11 = LF
Code 1147: 1600000000009993  CALL      0  0       9993  # putchar(...)
Code 1148:  103000000001102  LOAD      3  0       1102  # <restore R3>
Code 1149:  131000000001101  LOAD     31  0       1101  # <restore return address>
Code 1150: 1700000000000000  RET       0  0          0  # return (from print_primes)
Code 1151:                   RES                     1  # <return address> in main
Code 1152:                   RES                     1  # <refuge for R3> in main
Code 1153:                   RES                     1  # int i
Code 1154:  331000000001151  STORE    31  0       1151  # <save return address>
Code 1155:  303000000001152  STORE     3  0       1152  # <save R3>
Code 1156:  201000000000010  SET       1  0         10  # R1 = 10
Code 1157:  301000000001004  STORE     1  0       1004  # LF =
Code 1158:  201000000000000  SET       1  0          0  # R1 = 0
Code 1159:  202000000000001  SET       2  0          1  # R2 = 1
Code 1160:  301020000000003  STORE     1  2          3  # prime[...] =
Code 1161:  201000000000002  SET       1  0          2  # R1 = 2
Code 1162:  301000000001153  STORE     1  0       1153  # i =
Code 1163:  101000000001153  LOAD      1  0       1153  # R1 = i
Code 1164:  203010000000000  SET       3  1          0  # <save operand>
Code 1165:  201000000001000  SET       1  0       1000  # R1 = 1000
Code 1166: 1101030000000001  LESSEQ    1  3          1  # <=
Code 1167: 1401000000000000  JUMPEQ    1  0          0  # break while if != 0
Code 1168:  201000000000001  SET       1  0          1  # R1 = 1
Code 1169:  102000000001153  LOAD      2  0       1153  # R2 = i
Code 1170:  301020000000003  STORE     1  2          3  # prime[...] =
Code 1171:  101000000001153  LOAD      1  0       1153  # R1 = i
Code 1172:  203010000000000  SET       3  1          0  # <save operand>
Code 1173:  201000000000001  SET       1  0          1  # R1 = 1
Code 1174:  401030000000001  ADD       1  3          1  # +
Code 1175:  301000000001153  STORE     1  0       1153  # i =
Code 1176: 1400000000001163  JUMPEQ    0  0       1163  # continue while
---> 1167: 1401000000001177                       1177  # <update break address>
Code 1177: 1600000000001009  CALL      0  0       1009  # find_primes(...)
Code 1178: 1600000000001105  CALL      0  0       1105  # print_primes(...)
Code 1179:  103000000001152  LOAD      3  0       1152  # <restore R3>
Code 1180:  131000000001151  LOAD     31  0       1151  # <restore return address>
Code 1181: 1700000000000000  RET       0  0          0  # return (from main)
--->    0: 1600000000001154                       1154  # <address of 'main'>
```

Figur 7.18: Loggfil som demonstrerer kodegenereringen (del 3)

Kapittel 8

Koding

8.1 SUNs anbefalte Java-stil

Datafirmaet Sun, som utviklet Java, har også tanker om hvorledes 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.

8.1.1 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 9.1 på side 73.)

4) Selve klassen.

8.1.2 Variable

Variable bør deklareres én og én på hver linje:

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

De bør komme først i {}-blokken (dvs ikke etter noen setninger), men lokale `for`-indekser er helt OK:

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

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

Tabell 8.1: Suns forslag til navnevalg i Java-programmer

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

8.1.3 Setninger

Enkle setninger bør stå én og én på hver linje:

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

De ulike sammensatte setningene skal se ut som vist i figur 8.1 på neste side.

De skal alltid har {} rundt innmaten, og innmaten skal indenteres 4 posisjoner.

8.1.4 Navn

Navn bør velges slik det er angitt i tabell 8.1.

8.1.5 Utseende

8.1.5.1 Linjelengde og linjedeling

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

En linje som er for lang bør deles

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

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

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

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

if (uttrykk) {
    setninger;
}

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

if (uttykk) {
    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 8.1: Suns forslag til hvordan setninger bør skrives

8.1.5.2 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.

8.1.5.3 Blanke tegn

Sett inn blanke

- etter kommaer i parameterlister,
- rundt binære operatorer:

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

(men ikke etter unære operatorer: `-a`)

- ved typekonvertering:

```
(int) x
```

Kapittel 9

Dokumentasjon

9.1 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 bilbliotek: http://java.sun.com/javase/6/docs/api/.

9.1.1 Hvordan skrive JavaDoc-kommentarer

Det er ikke vanskelig å skrive JavaDoc-kommentarer. Her er en kort innføring; den fulle beskrivelsen finnes på nettsiden http://java.sun.com/j2se/javadoc/writingdoccomments/.

En JavaDoc-kommentarer for en klasse ser slik ut:

```
/**
 * kommentarer
 * kommentarer
 *      ⋮
 * @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 klassen/metoden
 * Ytterligere kommentarer
 *      ⋮
 * @param  navn1 Én-linjes beskrivelse av parameteren
 * @param  navn2 Én-linjes beskrivelse av parameteren
```

```
/**
 * 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 9.1: Java-kode med JavaDoc-kommentarer

```
 * @return Én-linjes 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.

9.1.2 Eksempel

I figur 9.1 kan vi se en Java-metode med dokumentasjon.

9.2 «Lesbar programmering»

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

- Programkoden og dokumentasjonen skrives som en enhet.
- Programmet deles opp i passende små navngitte enheter som legges inn i dokumentasjonen. Slike enheter kan referere til andre enheter.
- Programmet skrives i den rekkefølgen som er enklest for leseren å forstå.
- Dokumentasjonen skrives i et dokumentasjonsspråk (som LATEX) og kan benytte alle tilgjengelige typografiske hjelpemidler som figurer, matematiske formler, fotnoter, kapittelinndeling, fontskifte og annet.

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

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

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

9.2.1 Et eksempel

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

1) Skriv kildefilen `bubble.w0` (vist i figur 9.2 og 9.3). Dette gjøres med en vanlig tekstbehandler som for eksempel Emacs.
2) Bruk programmet `weave0`[1] til å lage det ferdige dokumentet som er vist i figur 9.4–9.7:

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

3) Bruke `tangle0` til å lage et kjørbart program:

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

[1] Dette eksemplet bruker Dags implementasjon av lesbar programmering kalt web_0; for mer informasjon, se /local/opt/web0/doc/web0.pdf.

bubble.w0 del 1

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

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

\begin{document}
\maketitle

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

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


<<bubble sort>>=
void bubble(int a[], int n)
{
  <<local variables>>

  <<use bubble sort>>
}
@
Bubble sorting is done by making several passes through
the array, each time letting the larger elements
‘‘bubble’’ up. This is repeated until the array is
completely sorted.

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

Figur 9.2: «Lesbar programmering» — kildefilen bubble.w0 del 1

bubble.w0 del 2

```
Each pass through the array consists of looking at
every pair of adjacent elements;\footnote{We could, on the
  average, double the execution speed of \texttt{bubble} by
  reducing the range of the \texttt{for}-loop by~1 each time.
  Since a simple implementation is the main issue, however,
  this improvement was omitted.} if the two are in
the wrong sorting order, they are swapped:
<<perform bubbling>>=
<<initialize>>
for (i=0; i<n-1; ++i)
  if (a[i]>a[i+1]) { <<swap a[i] and a[i+1]>> }
@
The \texttt{for}-loop needs an index variable
\texttt{i}:

<<local var...>>=
int i;
@
Swapping two array elements is done in the standard way
using an auxiliary variable \texttt{temp}. We also
increment a swap counter named \texttt{n\_swaps}.

<<swap ...>>=
temp = a[i];  a[i] = a[i+1];  a[i+1] = temp;
++n_swaps;
@
The variables \texttt{temp} and \texttt{n\_swaps}
must also be declared:

<<local var...>>=
int temp, n_swaps;
@
The variable \texttt{n\_swaps} counts the number of
swaps performed during one ``bubbling'' pass.
It must be initialized prior to each pass.

<<initialize>>=
n_swaps = 0;
@
If no swaps were made during the ``bubbling'' pass,
the array is sorted.

<<not sorted>>=
n_swaps > 0
@

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

Figur 9.3: «Lesbar programmering» — kildefilen bubble.w0 del 2

Bubble sort

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

August 8, 2008

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

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

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

#1 ⟨*bubble sort*⟩ ≡

```
void bubble(int a[], int n)
{
    ⟨local variables #4 (p.1)⟩

    ⟨use bubble sort #2 (p.1)⟩
}
```

(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*⟩ ≡

```
do {
    ⟨perform bubbling #3 (p.1)⟩
} 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*⟩ ≡

```
⟨initialize #6 (p.2)⟩
for (i=0; i<n-1; ++i)
  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*⟩ ≡

```
int i;
```

(This code is extended in #4$_a$ (p.2). It is used in #1 (p.1).)

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

File: *bubble.w0* page 1

Figur 9.4: «Lesbar programmering» — utskrift side 1

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

#5 ⟨*swap a[i] and a[i+1]*⟩ ≡

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

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

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

#4a ⟨*local variables* #4 (p.1)⟩ +≡

```
int temp, n_swaps;
```

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

#6 ⟨*initialize*⟩ ≡

```
n_swaps = 0;
```

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

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

#7 ⟨*not sorted*⟩ ≡

```
n_swaps > 0
```

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

File: *bubble.w0* page 2

Figur 9.5: «Lesbar programmering» — utskrift side 2

Variables

VARIABLES page 3

Figur 9.6: «Lesbar programmering» — utskrift side 3

Meta symbols

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

META SYMBOLS page 4

Figur 9.7: «Lesbar programmering» — utskrift side 4

Kapittel 10

Programredigering

Det finnes ulike verktøy for programmering. Vi skal her presentere to hovedgrupper, så får det bli opp til den enkelte å velge hva han eller hun vil bruke.

10.1 Spesialverktøy

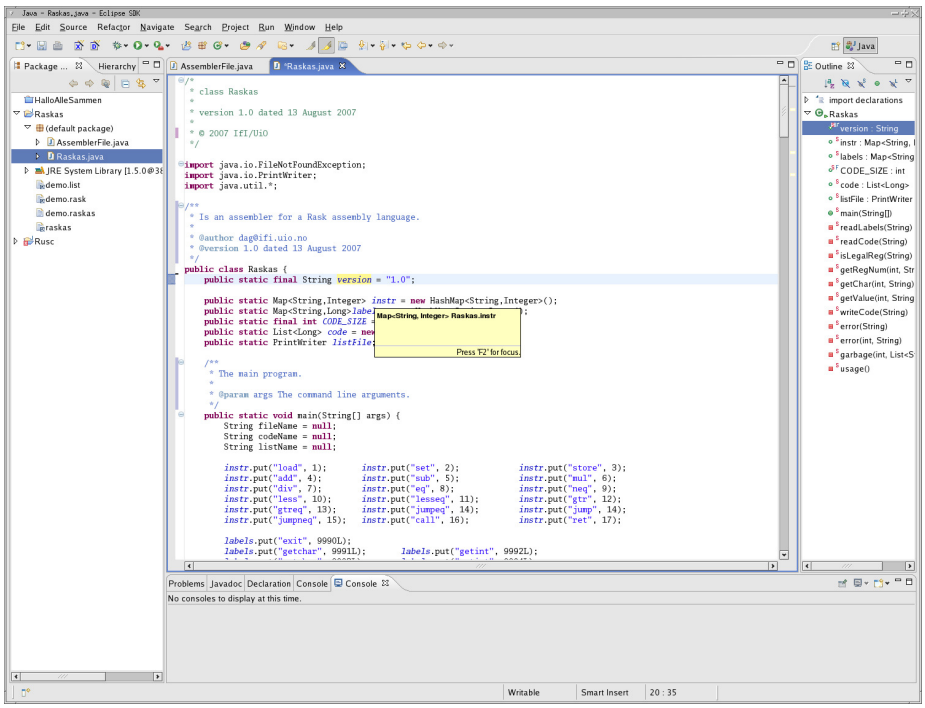
Et spesialverktøy er, som navnet sier, spesiallaget for ett spesielt språk. Det er typisk meget grafisk orientert; et eksempel til Java-programmering er Eclipse som er vist i figur 10.1 på neste side.

Det er mange og store fordeler med å bruke et slikt spesialverktøy som Eclipse eller NetBeans for Java:

- Mange brukere foretrekker «pek og klikk»-måten å jobbe på.
- Programmet har innebygget kunnskaper om Java og vil kunne vise feilmeldinger øyeblikkelig. (Man trenger ikke kompilere for å få se feilmeldingene.)
- Det går raskt å skrive kode fordi mye settes inn automatisk eller ved hjelp av menyer.
- Veldig mye informasjon om programmet gjøres lett tilgjengelig for brukeren. I figur 10.1 på neste side kan vi for eksempel se
  - Musen peker på variabelen `instr` og da kommer det frem et gult vindu med opplysninger om den, for eksempel hvilken tyupe den har.
  - Tidligere har brukeren klikket på variabelen `version` og da ble alle forekomster av den variabelen markert med gult.
- Hvis det oppstår feil under en testkjøring, får brukeren øyeblikkelig se hvor i programmet feilen oppsto.

10.2 Generelle verktøy

Andre verktøy er generelle verktøy i den forstand at samme program brukes til mange ulike programmeringsspråk. De kan allikevel ha innebygget



Figur 10.1: Eclipse i arbeid

litt kunnskap om språket de skal redigere; et typisk eksempel er Emacs i figur 10.2 på neste side som kan fargekode nøkkelord i Java.

Det finnes situasjoner der det kan være ganske så fornuftig å velge et generelt redigeringsverktøy som for eksempel Emacs:

- Det finnes ikke spesialverktøy for alle programmeringsspråk – når man jobber med et litt ukjent språk, må man ta til takke med hva man kan få.
- Spesialverktøy krever oftest en grafiske omgivelse; om man ikke har det (for eksempel fordi man jobber over nettet via et kommandovindu), vil et generelt verktøy være redningen.
- Det kan være ganske mye jobb å lære et nytt spesialverktøy.
- I mange generelle verktøy jobber man raskere fordi det meste kan gjøres med tastaturet i stedet for med musen.
- Noen nyttige operasjoner finnes sjelden i spesialverktøyene:
  - Bytt om to tegn.
  - Bytt om to linjer.
  - Finn alle variable som begynner med `runThis...` og omnavn dem til `runCurrent...`.

```
package no.uio.ifi.rusc.rusc;

/*
 * class Rusc
 *
 * version 1.02 dated 23 June 2008
 *
 * © 2008 Ifi/UiO
 */

import java.io.*;
import no.uio.ifi.rusc.chargenerator.CharGenerator;
import no.uio.ifi.rusc.code.Code;
import no.uio.ifi.rusc.error.Error;
import no.uio.ifi.rusc.log.Log;
import no.uio.ifi.rusc.scanner.Scanner;
import no.uio.ifi.rusc.scanner.Token;
import no.uio.ifi.rusc.syntax.Syntax;

/**
 * The "main program" of the Rusc compiler.

 * @author dag@ifi.uio.no
 */
public class Rusc {
    public static final String version = "1.02";

    public static String sourceName = null;

    /**
     * The actual "main program".
     * It will initialize the various modules and start the
     * compilation (or module testing, if requested); finally,
     * it will terminate the modules.
     *
     * @param args The command line arguments.
     */
    public static void main(String[] args) {
        boolean testParser = false, testScanner = false;

        for (int opt_no = 0;  opt_no < args.length;  ++opt_no) {
            String opt = args[opt_no];

            if (opt.equals("-logC")) {
                Log.doLogCode = true;
            } else if (opt.equals("-logP")) {
                Log.doLogParser = true;
            } else if (opt.equals("-logT")) {
                Log.doLogTree = true;
            } else if (opt.equals("-logS")) {
                Log.doLogScanner = true;
            } else if (opt.equals("-testparser")) {
                testParser = true;
                Log.doLogParser = Log.doLogTree = true;
            } else if (opt.equals("-testscanner")) {
                testScanner = true;
                Log.doLogScanner = true;
            } else if (opt.startsWith("-")) {
                Error.error("Unknown option: `" + opt + "'!");
            } else {
                if (sourceName != null) Error.giveUsage();
                sourceName = opt;
            }
        }
        if (sourceName == null) Error.giveUsage();

        Error.init();   Log.init();  Code.init();
-1:--  Rusc.java      Top (1,0)      (Java/l Abbrev)-----------------------------
 Loading cc-mode...done
```

Figur 10.2: Emacs i arbeid