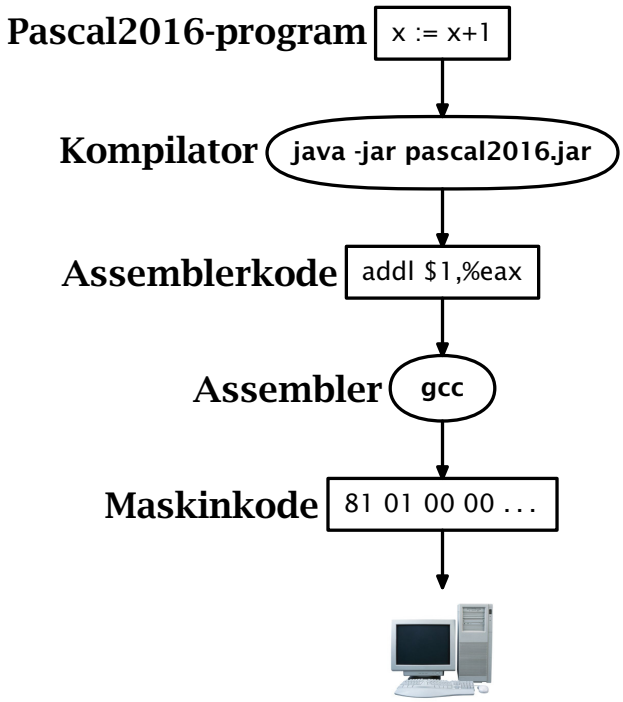


Figur 1.1: Sammenhengen mellom Pascal2016, kompilator, assembler og en x86-maskin

Del 1: Skanneren

Første skritt, del 1, består i å få Pascal2016s **skanner** til å virke. Skanneren er den modulen som fjerner kommentarer fra programmet, og så deler den gjenstående teksten i en veldefinert sekvens av såkalte **symboler** (på engelsk «tokens»). Symbolene er de «ordene» programmet er bygget opp av, så som *navn*, *tall*, *nøkkelord*, '+', '>=', '`:=`' og alle de andre tegnene og tegnkombinasjonene som har en bestemt betydning i Pascal2016-språket.

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

Del 2: Parseren

Del 2 vil ta imot den symbolsekvensen som blir produsert av del 1, og det sentrale arbeidet her vil være å sjekke at denne sekvensen har den formen et riktig Pascal2016-program skal ha (altså, at den følger Pascal2016s **syntaks**).

Om alt er i orden, skal del 2 bygge opp et **syntakstre**, en **trestruktur** av objekter som direkte representerer det aktuelle Pascal2016-programmet, altså hvordan det er satt sammen av «expression» inne i «statement» inne i «func decl» osv. Denne trestrukturen skal så leveres videre til del 3 som grunnlag for sjekking.

Del 3: Sjekking

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

Del 4: Kodegenerering

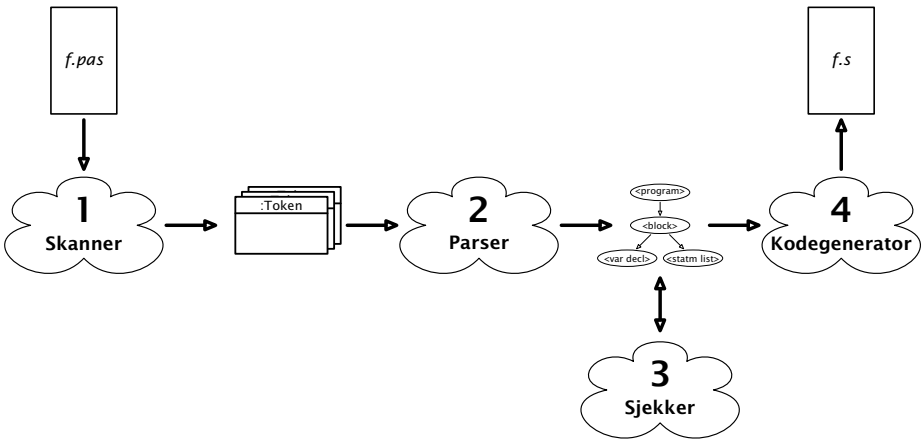
Til sist kan kompilatoren vår gjøre selve oversettelsen til x86-kode; da tar vi igjen utgangspunkt i den trestrukturen som del 2 produserte for det aktuelle Pascal2016-programmet. Koden skal legges på en fil og den skal være i såkalt x86 assemblerformat.

I avsnitt 4.5 på side 47 er det angitt hvilke sekvenser av x86-instruksjoner hver enkelt Pascal2016-konstruksjon skal oversettes til, og det er viktig å merke seg at disse skjemaene *skal* følges (selv om det i enkelte tilfeller er mulig å produsere lurere x86-kode; dette skal vi eventuelt se på i noen ukeoppgaver).

Kapittel 4

Prosjektet

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



Figur 4.1: Oversikt over prosjektet

4.1 Diverse informasjon om prosjektet

4.1.1 Basiskode

På emnets nettside ligger 2100-oblig-2016.zip som er nyttig kode å starte med. Lag en egen mappe til prosjektet og legg ZIP-filen der. Gjør så dette:

```
$ cd mappen
$ unzip inf2100-oblig-2016.zip
$ ant
```

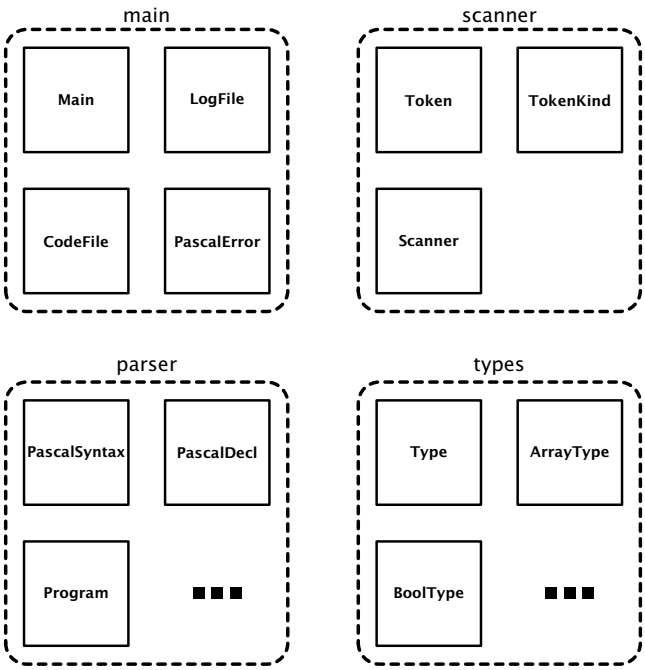
Dette vil resultere i en kjørbar fil `pascal2016.jar` som kan kjøres slik

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

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

4.1.2 Oppdeling i moduler

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



Figur 4.2: De fire modulene i kompilatoren

`main` innholder fire sentrale klasser som alle er ferdig programmert:

- `Main` er «hovedprogrammet» som styrer hele kompileringen.
- `LogFile` brukes til å opprette en loggfil (se avsnitt 4.1.4 på neste side).
- `CodeFile` brukes til å skrive kodefilen som skal være resultatet av kompileringen.
- `PascalError` benyttes til feilhåndteringen.

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

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

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

4.1.3 Selvidentifikasjon

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

I dette prosjektet skal vi la alle klassene ha en metode

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

som gir den informasjonen vi ønsker om objektet; se for eksempel på klassen `Token` i figur 4.4 på side 39.[6]

4.1.4 Logging

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

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

Tabell 4.1: Opsjoner for logging

4.1.5 Testprogrammer

Til hjelp under arbeidet finnes diverse testprogrammer:

- I mappen ~inf2100/oblig/test/ (som også er tilgjengelig fra en nettleser som http://inf2100.at.ifi.uio.no/oblig/test/) finnes noen Pascal2016-programmer som bør fungere i den forstand at de ikke gir feilmeldinger, men genererer riktig kode; resultatet av kjøringene skal dessuten gi resultatet vist i `.res`-filene.
- I mappen ~inf2100/oblig/feil/ (som også er tilgjengelig utenfor Ifi som http://inf2100.at.ifi.uio.no/oblig/feil/) finnes diverse småprogrammer som alle inneholder en feil eller en raritet. Kompilatoren din bør håndetere disse programmene på samme måte som referansekompilatoren.

4.1.6 På egen datamaskin

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

`ant` er en overbygning til Java-kompilatoren; den gjør det enkelt å kompilere et system med mange Java-filer. Programmet kan hentes ned fra http://ant.apache.org/bindownload.cgi.

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

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

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

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

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

4.1.7 Tegnsett

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

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

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

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

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

4.2 Del 1: Skanneren

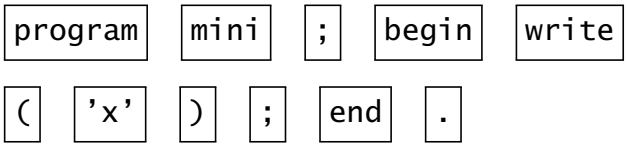
Skanneren leser programteksten fra en fil og deler den opp i **symboler** (på engelsk «tokens»), omtrent slik vi mennesker leser en tekst ord for ord.

`mini.pas`

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

Figur 4.3: Et minimalt Pascal2016-program `mini.pas`

Programmet vist i figur 4.3 inneholder for eksempel disse symbolene:



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

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

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

4.2.1 Representasjon av symboler

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

Token.java

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

Figur 4.4: Klassen `Token`

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

4.2.2 Skanneren

Selve skanneren er definert av klassen `Scanner`; se figur 4.6 på neste side. Legg merke til at den inneholder to symboler: `curToken` og `nextToken`, nemlig det nåværende og det neste symbolet. Grunnen til det er at vi av og til ønsker å se litt forover etter hva som kommer senere i teksten.

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

4.2.3 Logging

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

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

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

TokenKind.java

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

Figur 4.5: Enum-klassen TokenKind

Scanner.java

```
7
8   public class Scanner {
9       public Token curToken = null, nextToken = null;
10
11      private LineNumberReader sourceFile = null;
12      private String sourceFileName, sourceLine = "";
13      private int sourcePos = 0;
14
15      public Scanner(String fileName) {
16          sourceFileName = fileName;
17          try {
18              sourceFile = new LineNumberReader(new FileReader(fileName));
19          } catch (FileNotFoundException e) {
20              Main.error("Cannot read " + fileName + "!");
21          }
22
23          readNextToken();  readNextToken();
24      }
25
26
27      public String identify() {
28          return "Scanner reading " + sourceFileName;
29      }
              ⋮
239 }
```

Figur 4.6: Klassen Scanner

mini.pas

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

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

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

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

For å demonstrere hva som ønskes av testutskrift, har jeg laget både et minimalt og litt større Pascal-program; se figur 4.7 og figur 4.15 på side 55. Når kompilatoren vår kjøres med opsjonen `-testscanner`, skriver de ut logginformasjonen vist i henholdsvis figur 4.7 og figur 4.16 til 4.17 på side 55–56.

4.2.4 Mål for del 1

Mål for del 1

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

4.3 Del 2: Parsering

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

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

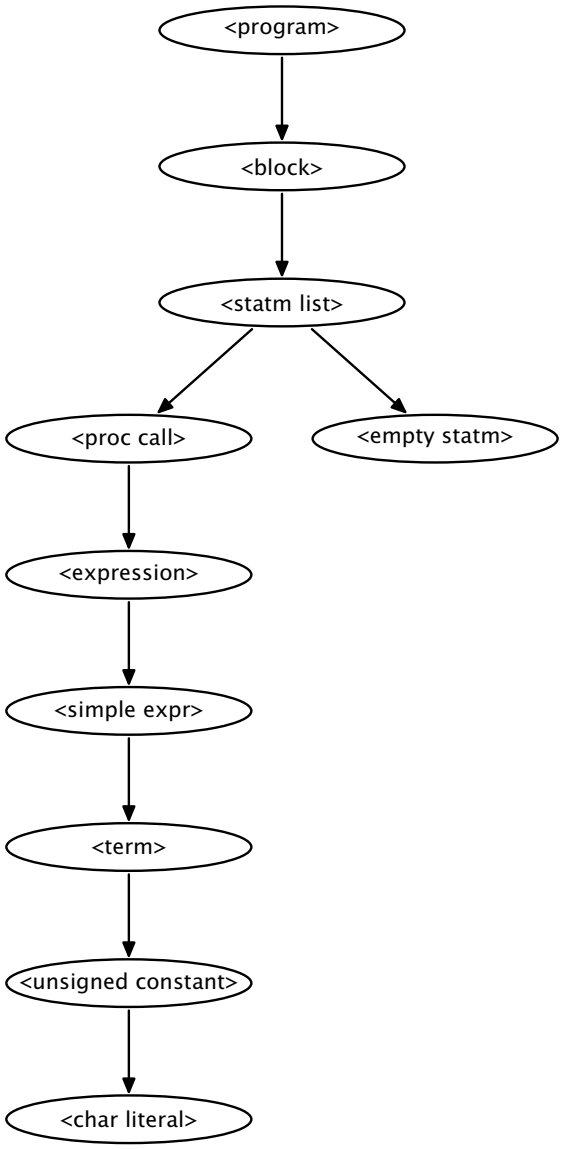
Testprogrammet `mini.pas` skal for eksempel gi treet vist i figur 4.8 på neste side.

4.3.1 Implementasjon

Aller først må det defineres en klasse per ikke-terminal («firkantene» i grammatikken), og alle disse må være subklasser av `PascalSyntax`. (Alle ikke-terminaler som representerer en deklarasjon, bør være subklasse av

mini.pas

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



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

`PascalDecl`, men dette kan ordnes under del 3.) Klassene må inneholde tilstrekkelige deklarasjoner til å kunne representere ikke-terminalen. Som et eksempel er vist klassen `WhileStatm` som representerer ⟨while-statm⟩; se figur 4.9 på neste side.

Et par ting verdt å merke seg:

- Ikke-terminalene ⟨letter a-z⟩, ⟨digit 0-9⟩ og ⟨char except '⟩ er allerede tatt hånd om av skanneren, så de kan vi se bort fra nå.
- ⟨name⟩ trenger ikke en egen klasse; en `String` er nok.
- Ikke-terminaler som kun er definert som et valgt mellom ulike andre ikke-terminaler (som f eks ⟨constant⟩ og ⟨type⟩) bør implementeres som en abstrakt klasse, og så bør alternativene være sub-klasser av denne abstrakte klassen.

WhileStatm.java

```
1  package parser;
2
3  import main.*;
4  import scanner.*;
5  import static scanner.TokenKind.*;
6
7  /* <while-statm> ::= 'while' <expression> 'do' <statement> */
8
9  class WhileStatm extends Statement {
10     Expression expr;
11     Statement body;
12
13     WhileStatm(int lNum) {
14         super(lNum);
15     }
16
17     @Override public String identify() {
18         return "<while-statm> on line " + lineNum;
19     }
           ⋮
40
41     @Override void prettyPrint() {
42         Main.log.prettyPrint("while ");  expr.prettyPrint();
43         Main.log.prettyPrintLn(" do");  Main.log.prettyIndent();
44         body.prettyPrint();  Main.log.prettyOutdent();
45     }
46
47     static WhileStatm parse(Scanner s) {
48         enterParser("while-statm");
49
50         WhileStatm ws = new WhileStatm(s.curLineNum());
51         s.skip(whileToken);
52
53         ws.expr = Expression.parse(s);
54         s.skip(doToken);
55         ws.body = Statement.parse(s);
56
57         leaveParser("while-statm");
58         return ws;
59     }
60 }
```

Figur 4.9: Klassen `WhileStatm`

4.3.2 Parsering

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

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

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

4.3.2.1 Tvetydigheter

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

```
1   v := a
```

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

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

kan `a` være tre forskjellige ting i henhold til syntaksen:

1) ⟨unsigned constant⟩
2) ⟨variable⟩
3) ⟨func call⟩ (uten paramtre)

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

4.3.3 Syntaksfeil

Ved å benytte denne parseringsmetoden er det enkelt å finne grammatikkfeil: Når det ikke finnes noe lovlig alternativ i jernbanediagrammene, har vi en feilsituasjon, og vi må kalle `PascalSyntax.error`. (Metodene `test` og `skip` gjør dette automatisk for oss.)

4.3.4 Logging

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

Våre to vanlige testprogram vist i henholdsvis figur 4.3 på side 38 og figur 4.15 på side 55 vil produsere loggfilene i figur 4.10 og figurene 4.18 til 4.22 på side 57 og etterfølgende når kompilatoren kjøres med opsjonen `-logP` eller `-testparser`.

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

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

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

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

i hver subklasse av `PascalSyntax`.

> ***Mål for del 2***
>
> *Programmet skal implementere parsering og også utskrift av det lagrede programmet; med andre ord skal opsjonen `-testparser` gi utskrift som vist i figurene 4.10–4.11 og 4.18–4.23.*

4.4 Del 3: Sjekking

Den tredje delen er å få sjekkingen på plass.

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

4.4.1 Sjekke navn ved deklarasjoner

Det må sjekkes at navn er deklarert riktig. I Pascal2016 er dette enkelt, for det er bare én mulig feil: å deklarere navn flere ganger i samme blokk.

4.4.2 Sjekke deklarasjoner

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

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

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

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

Symbol	Betydning
$\mathcal{T}_x$	Typen til x
$\mathcal{T}_f^{\mathcal{F}}$	Funksjonstypen til `f`
$\mathcal{T}_f^{\mathcal{P}_i}$	Typen til fs *formelle parameter*[8] nr i
$\{\mathcal{A}\}$	Mengden av alle array-typer
$\mathcal{T}_a^{\mathcal{E}}$	Typen til elementene i arrayen a
$\mathcal{T}_a^{\mathcal{I}}$	Typen til indeksen i arrayen a

Tabell 4.2: Notasjon for typesjekking

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

Tabell 4.3: Typesjekking av setninger

4.4.3 Sjekke navnebruk

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

4.4.4 Bestemme typer

For alle uttrykk og deluttrykk må vi finne hvilken type de har; dette settes inn i `Expression.type`, `Term.type` og alle andre klasser for deluttrykk. Ved å angi `-logT` kan brukeren få logget alle typesjekkene som foretas; se eksempel i figur 4.25 på side 62. (Vårt minimale testprogram `mini.pas` fra figur 4.3 på side 38 er så enkelt at det ikke produserer noen typesjekklogg.)

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

4.4.5 Beregne konstanter

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

[8] En **formell parameter** er en parameter i funksjons- eller prosedyre*deklarasjonen*, mens en **aktuell parameter** er en parameter i *kallet* på funksjonen eller prosedyren.

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

Tabell 4.4: Typesjekking av uttrykk

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

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

> ***Mål for del 3***
>
> *Kompilatoren skal foreta navnebindinger og sjekke typer, og når den kjøres med opsjonen* `-testchecker` *produsere data om dette som vist i figur 4.12 og 4.24.*

4.5 Del 4: Kodegenerering

4.5.1 Konvensjoner

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

4.5.2 Registre

Vi vil bruke disse registrene:

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

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

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

%ESP peker på toppen av kjørestakken.

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

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

Figur 4.14: Kodefil laget fra `mini.pas`

4.5.2.1 Navn

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

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

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

Variabler trenger heller ikke navn siden de også ligger på stakken. Nøyaktig hvor de ligger på stakken må kompilatoren vår regne seg frem til; dette avhenger av de andre lokale variablene i samme funksjon eller prosedyre.

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

4.5.3 Oversettelse av uttrykk

Hovedregelen når vi skal lage kode for å beregne uttrykk, er at resultatet av alle uttrykk og deluttrykk skal ende opp i %EAX. Dette gjør kodegenereringen svært mye enklere, men vi får ikke alltid den mest optimale koden.[9]

4.5.3.1 Operander i uttrykk

I tabell 4.5 på neste side er vist hvilken kode som må genereres for å hente en verdi ⟨n⟩, en enkel variabel $\langle v^{(b)o} \rangle$ (der b er blokknivået og o er variabelens «offset»), et arrayelement $\langle a^{(b)o} \rangle$`[`⟨e⟩`]` eller et uttrykk i parenteser (`(`⟨e⟩`)`) inn i register %EAX.

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

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

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

Boolean-verdier blir representert av heltall, nærmere bestemt:

`false` = 0, `true` = 1

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

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

⟨a$^{(b)o}$⟩`[`⟨e⟩`]`	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩
		`subl` `$`low`,%eax` (Dropp om low=0)
		`movl` $-4b$`(%ebp),%edx`
		`leal` o`(%edx),%edx`
		`movl (%edx,%eax,4),%eax`

`(`⟨e⟩`)`	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩

Tabell 4.5: Kode for å hente en verdi inn i %EAX

4.5.3.2 Operatorer i uttrykk

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

Unære operatorer Tabell 4.6 viser hvordan vi skal oversette de unære operatorene.

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

Tabell 4.6: Kode generert av unære operatorer i uttrykk

Binære operatorer I tabell 4.7 på neste side er vist hvordan de binære operatorene `+`, `div` (som trenger litt annen kode enn de andre regneoperatorene) og `=` skal oversettes. De øvrige finner du sikkert selv.

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

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

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

Tabell 4.7: Kode generert av binære operatorer i uttrykk

4.5.4 Oversettelse av setninger

4.5.4.1 Oversettelse av tomme setninger

Dette er den enkleste setningen å oversette, som vist i tabell 4.8.

	⇒	

Tabell 4.8: Kode generert av tom setning

4.5.4.2 Oversettelse av sammensatte setninger

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

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

Tabell 4.9: Kode generert av sammensatt setning

4.5.4.3 Oversettelse av tilordningssetninger

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

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

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

Tabell 4.10: Kode generert av tilordning

4.5.4.4 Oversettelse av kallsetninger

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

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

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

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

Tabell 4.11: Kode generert av prosedyrekall

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

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

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

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

Tabell 4.12: Kode generert av kall på `write`

4.5.4.5 Oversettelse av if-setninger

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

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

Tabell 4.13: Kode generert av if-setning

4.5.4.6 Oversettelse av while-setninger

Oversettelse av en while-setning innebærer å lage en løkke og en løkketest; dette er vist i tabell 4.14 på neste side.

4.5.5 Oversettelse av funksjoner og prosedyrer

Som vist i figur 4.15 på neste side legger vi inn litt fast kode i begynnelsen og slutten av funksjonen. Legg også merke til at:

- Parametrene resulterer ikke i noe kode siden de skal ligge på stakken når funksjonen kalles.

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

Tabell 4.14: Kode generert av while-setning

- Instruksjonen `enter` setter av plass til lokale variabler på stakken; for å finne ut hvor mange byte vi skal sette av, må vi summere hvor mange byte hver enkelt lokal variabel tar. Til denne summen skal vi addere **32** for systeminformasjon.

  Vi må også angi hvilket **blokknivå** funksjonen/prosedyren har.

- Vi må bruke `leave`-instruksjonen til å frigjøre plassen vi satte av til lokale variabler før vi hopper tilbake med en `ret`.

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

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

Tabell 4.15: Kode generert av funksjons- og prosedyredeklarasjon

4.5.5.1 Oversettelse av hovedprogrammet

Det enkleste er å late som om hoveprogrammet er en prosedyre som kalles av en minimal `main`;[11] se tabell 4.16 på neste side.

4.5.6 Deklarasjon av variabler

4.5.6.1 Deklarasjon av lokale variabler

Programmet/prosedyren/funksjonen sørger selv for å sette av plass til sine lokale variabler på stakken (se tabell 4.15).

4.5.6.2 Deklarasjon av parametre

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

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

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

Tabell 4.16: Kode generert av hovedprogrammet

Mål for del 4

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

4.6 Et litt større eksempel

gcd.pas

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

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

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

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

```
Scanner: name token on line 10: integer
Scanner: ; token on line 10
  11: begin
Scanner: begin token on line 11
  12:    if n = 0 then
Scanner: if token on line 12
Scanner: name token on line 12: n
Scanner: = token on line 12
Scanner: number token on line 12: 0
Scanner: then token on line 12
  13:       GCD := m
Scanner: name token on line 13: gcd
Scanner: := token on line 13
Scanner: name token on line 13: m
  14:    else
Scanner: else token on line 14
  15:       GCD := GCD(n, m mod n)
Scanner: name token on line 15: gcd
Scanner: := token on line 15
Scanner: name token on line 15: gcd
Scanner: ( token on line 15
Scanner: name token on line 15: n
Scanner: , token on line 15
Scanner: name token on line 15: m
Scanner: mod token on line 15
Scanner: name token on line 15: n
Scanner: ) token on line 15
  16: end; { GCD }
Scanner: end token on line 16
Scanner: ; token on line 16
  17:
  18: begin
Scanner: begin token on line 18
  19:    res := GCD(v1,v2);
Scanner: name token on line 19: res
Scanner: := token on line 19
Scanner: name token on line 19: gcd
Scanner: ( token on line 19
Scanner: name token on line 19: v1
Scanner: , token on line 19
Scanner: name token on line 19: v2
Scanner: ) token on line 19
Scanner: ; token on line 19
  20:    write('G', 'C', 'D', '(', v1, ',', v2, ')', '=', res, eol);
Scanner: name token on line 20: write
Scanner: ( token on line 20
Scanner: char token on line 20: 'G'
Scanner: , token on line 20
Scanner: char token on line 20: 'C'
Scanner: , token on line 20
Scanner: char token on line 20: 'D'
Scanner: , token on line 20
Scanner: char token on line 20: '('
Scanner: , token on line 20
Scanner: name token on line 20: v1
Scanner: , token on line 20
Scanner: char token on line 20: ','
Scanner: , token on line 20
Scanner: name token on line 20: v2
Scanner: , token on line 20
Scanner: char token on line 20: ')'
Scanner: , token on line 20
Scanner: char token on line 20: '='
Scanner: , token on line 20
Scanner: name token on line 20: res
Scanner: , token on line 20
Scanner: name token on line 20: eol
Scanner: ) token on line 20
Scanner: ; token on line 20
  21: end.
Scanner: end token on line 21
Scanner: . token on line 21
Scanner: e-o-f token
```

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

```
   1: program GCD;
Parser:   <program>
   2: /* A program to compute the {greatest common} of two numbers,
   3:    i.e., the biggest number by which the two original
   4:    numbers can be divided without a remainder. */
   5:
   6: const v1 = 1071; v2 = 462;
Parser:     <block>
Parser:       <const decl part>
Parser:         <const decl>
Parser:           <constant>
Parser:             <unsigned constant>
Parser:               <number literal>
Parser:               </number literal>
Parser:             </unsigned constant>
Parser:           </constant>
Parser:         </const decl>
Parser:         <const decl>
Parser:           <constant>
Parser:             <unsigned constant>
Parser:               <number literal>
   7:
   8: var res: integer;
Parser:               </number literal>
Parser:             </unsigned constant>
Parser:           </constant>
Parser:         </const decl>
Parser:       </const decl part>
Parser:       <var decl part>
Parser:         <var decl>
Parser:           <type>
Parser:             <type name>
   9:
  10: function GCD (m: integer; n: integer): integer;
Parser:             </type name>
Parser:           </type>
Parser:         </var decl>
Parser:       </var decl part>
Parser:       <func decl>
Parser:         <param decl list>
Parser:           <param decl>
Parser:             <type name>
Parser:             </type name>
Parser:           </param decl>
Parser:           <param decl>
Parser:             <type name>
Parser:             </type name>
Parser:           </param decl>
Parser:         </param decl list>
Parser:         <type name>
  11: begin
Parser:         </type name>
  12:    if n = 0 then
Parser:         <block>
Parser:           <statm list>
Parser:             <statement>
Parser:               <if-statm>
Parser:                 <expression>
Parser:                   <simple expr>
Parser:                     <term>
Parser:                       <factor>
Parser:                         <variable>
Parser:                         </variable>
Parser:                       </factor>
Parser:                     </term>
Parser:                   </simple expr>
Parser:                   <rel opr>
Parser:                   </rel opr>
Parser:                   <simple expr>
Parser:                     <term>
Parser:                       <factor>
Parser:                         <unsigned constant>
Parser:                           <number literal>
  13:       GCD := m
Parser:                           </number literal>
```

Figur 4.18: Loggfil som viser parsering av `gcd.pas` (del 1)

```
Parser:                          </unsigned constant>
Parser:                        </factor>
Parser:                      </term>
Parser:                    </simple expr>
Parser:                  </expression>
Parser:                  <statement>
Parser:                    <assign statm>
Parser:                      <variable>
Parser:                      </variable>
  14:    else
Parser:                      <expression>
Parser:                        <simple expr>
Parser:                          <term>
Parser:                            <factor>
Parser:                              <variable>
  15:       GCD := GCD(n, m mod n)
Parser:                              </variable>
Parser:                            </factor>
Parser:                          </term>
Parser:                        </simple expr>
Parser:                      </expression>
Parser:                    </assign statm>
Parser:                  </statement>
Parser:                  <statement>
Parser:                    <assign statm>
Parser:                      <variable>
Parser:                      </variable>
Parser:                      <expression>
Parser:                        <simple expr>
Parser:                          <term>
Parser:                            <factor>
Parser:                              <func call>
Parser:                                <expression>
Parser:                                  <simple expr>
Parser:                                    <term>
Parser:                                      <factor>
Parser:                                        <variable>
Parser:                                        </variable>
Parser:                                      </factor>
Parser:                                    </term>
Parser:                                  </simple expr>
Parser:                                </expression>
Parser:                                <expression>
Parser:                                  <simple expr>
Parser:                                    <term>
Parser:                                      <factor>
Parser:                                        <variable>
Parser:                                        </variable>
Parser:                                      </factor>
Parser:                                      <factor opr>
Parser:                                      </factor opr>
Parser:                                      <factor>
Parser:                                        <variable>
  16: end; { GCD }
Parser:                                        </variable>
Parser:                                      </factor>
Parser:                                    </term>
Parser:                                  </simple expr>
Parser:                                </expression>
Parser:                              </func call>
Parser:                            </factor>
Parser:                          </term>
Parser:                        </simple expr>
Parser:                      </expression>
Parser:                    </assign statm>
Parser:                  </statement>
Parser:                </if-statm>
Parser:              </statement>
Parser:            </statm list>
  17:
  18: begin
Parser:          </block>
  19:    res := GCD(v1,v2);
Parser:        </func decl>
Parser:        <statm list>
```

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

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

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

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

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

```
Parser:                 </term>
Parser:               </simple expr>
Parser:             </expression>
Parser:             <expression>
Parser:               <simple expr>
Parser:                 <term>
Parser:                   <factor>
Parser:                     <variable>
Parser:                     </variable>
Parser:                   </factor>
Parser:                 </term>
Parser:               </simple expr>
Parser:             </expression>
  21: end.
Parser:           </proc call>
Parser:         </statement>
Parser:         <statement>
Parser:           <empty statm>
Parser:           </empty statm>
Parser:         </statement>
Parser:       </statm list>
Parser:     </block>
Parser:   </program>
```

Figur 4.22: Loggfil som viser parsering av `gcd.pas` (del 5)

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

Figur 4.23: Loggfil med «skjønnskrift» av `gcd.pas`

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

Figur 4.24: Loggfil med navnebinding for `gcd.pas`

```
Type check = operands on line 12: type Integer vs type Integer
Type check if-test on line 12: type Boolean vs type Boolean
Type check := on line 13: type Integer vs type Integer
Type check param #1 on line 15: type Integer vs type Integer
Type check left mod operand on line 15: type Integer vs type Integer
Type check right mod operand on line 15: type Integer vs type Integer
Type check param #2 on line 15: type Integer vs type Integer
Type check := on line 15: type Integer vs type Integer
Type check param #1 on line 19: type Integer vs type Integer
Type check param #2 on line 19: type Integer vs type Integer
Type check := on line 19: type Integer vs type Integer
```

Figur 4.25: Loggfil med typesjekking for `gcd.pas`

```
# Code file created by Pascal2016 compiler 2016-07-29 11:47:15
        .globl  main
main:
        call    prog$gcd_1              # Start program
        movl    $0,%eax                 # Set status 0 and
        ret                             # terminate the program
func$gcd_2:
        enter   $32,$2                  # Start of gcd
                                        # Start if-statement
        movl    -8(%ebp),%edx
        movl    12(%edx),%eax           #   n
        pushl   %eax
        movl    $0,%eax                 #   0
        popl    %ecx
        cmpl    %eax,%ecx
        movl    $0,%eax
        sete    %al                     # Test =
        cmpl    $0,%eax
        je      .L0003
        movl    -8(%ebp),%edx
        movl    8(%edx),%eax            #   m
        movl    -8(%ebp),%edx
        movl    %eax,-32(%edx)          # gcd :=
        jmp     .L0004
.L0003:
        movl    -8(%ebp),%edx
        movl    8(%edx),%eax            #   m
        pushl   %eax
        movl    -8(%ebp),%edx
        movl    12(%edx),%eax           #   n
        movl    %eax,%ecx
        popl    %eax
        cdq
        idivl   %ecx
        movl    %edx,%eax               #   mod
```

Figur 4.26: Kodefil produsert fra `gcd.pas` (del 1)

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

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

Kapittel 5

Kompilering av blokkorienterte språk

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

5.1 Bakgrunn

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

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

5.1.1 Kontekstvektor

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

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

5.1.2 Et eksempel

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

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

Figur 5.1: En enkelt testprogram

5.2 Start av hovedprogrammet

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

1) Hovedprogrammet kalles med en `call`-instruksjon som legger returadressen (dvs adressen til instruksjonen etter `call`-instruksjonen) på stakken.

2) Den første instruksjonen i hovedprogrammet er **`enter`** som gjør flere ting:

   (a) Innholdet i `%EBP`-registeret gjemmes unna på stakken.

   (b) Det settes av plass til kontekstvektoren (28 byte),[12] returverdien (4 byte)[13] og 2 lokale variabler (2×4 = 8 byte); tilsammen 40 byte.

   (c) Kontekstvektoren fra blokken utenfor kopieres inn, men siden hovedprogrammet er på blokknivå 1, er det ingen ytre blokk.

   (d) Kontekstvektoren utvides med en peker til denne blokken.

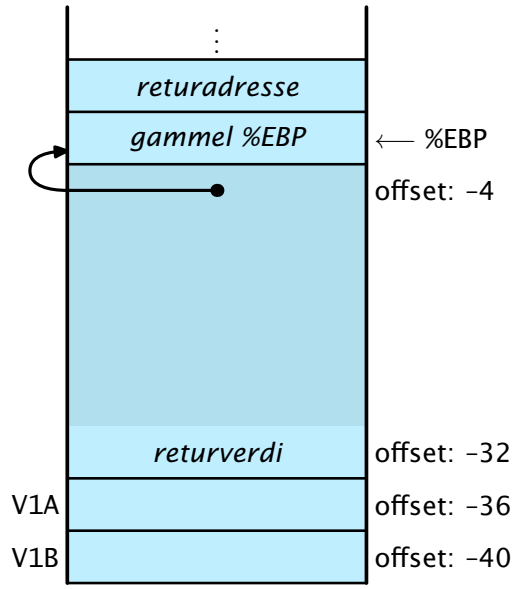
Vi får da situasjonen vist i figur 5.2 på neste side. Kontekstvektoren er markert med litt mørkere farge.

5.3 Start av en prosedyre

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

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

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



Figur 5.2: Stakken når hovedprogrammet starter

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

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

5.4 Start av en indre funksjon

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

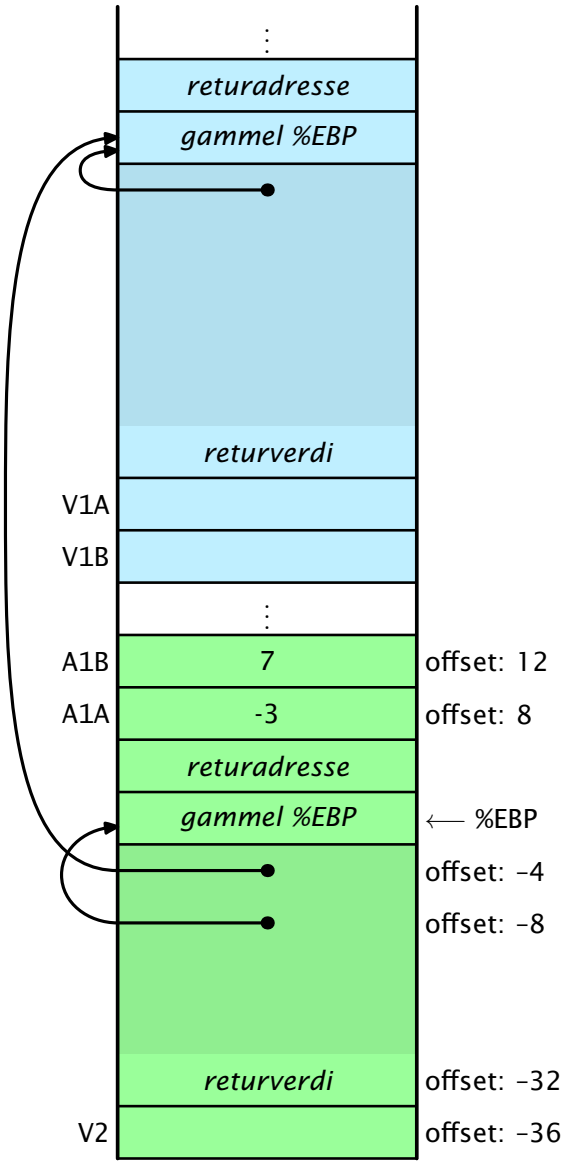
Vi ser nå at vi kan få tak i alle synlige variabler ved å gå via kontekstvektoren. For eksempel får vi tak i den lokale `V3` som ligger på blokknivå 3 ved først å gjøre følgende:

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

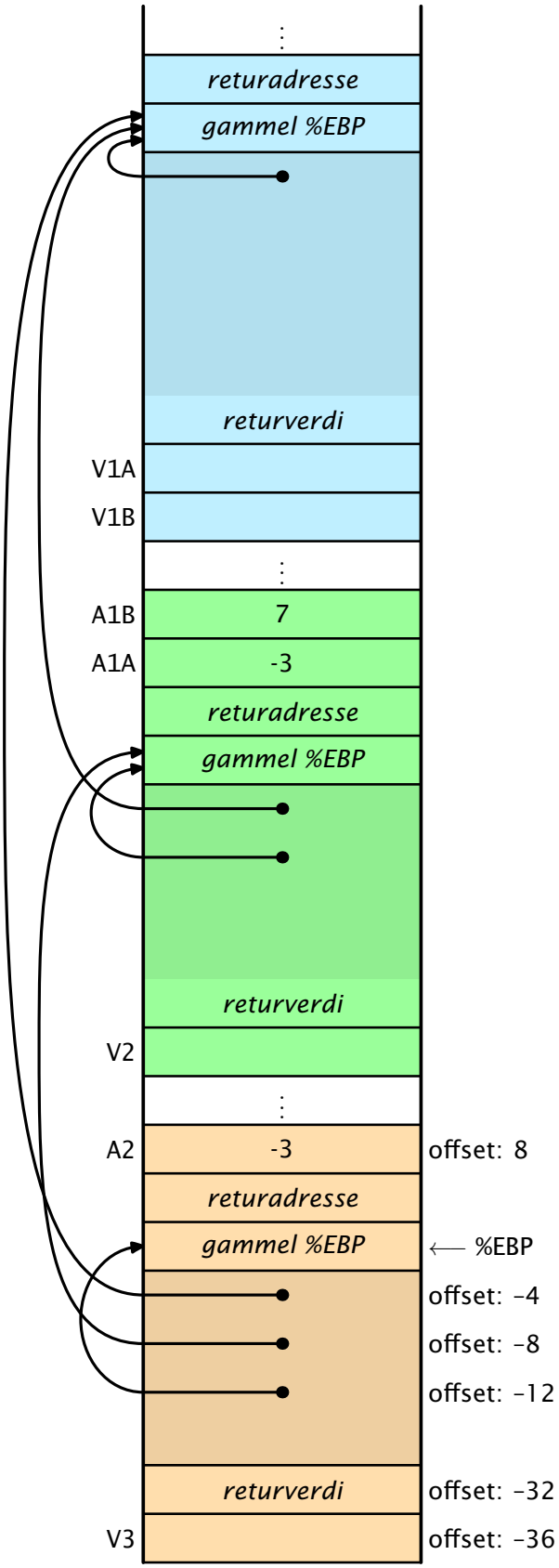
```
1         movl    -12(%ebp),%edx
2         movl    -36(%edx),%eax          #   v3
```

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

```
1         movl    -4(%ebp),%edx
2         movl    -40(%edx),%eax          #   v1b
```



Figur 5.3: Stakken når prosedyren har startet



Figur 5.4: Stakken når den indre funksjonen har startet

```
# Code file created by Pascal2016 compiler 2016-05-07 11:02:35
        .globl  main
main:
        call    prog$blokker_1          # Start program
        movl    $0,%eax                 # Set status 0 and
        ret                             # terminate the program
func$f2_3:
        enter   $36,$3                  # Start of f2
        movl    -12(%ebp),%edx
        movl    8(%edx),%eax            #   a2
        pushl   %eax
        movl    $1,%eax                 #   1
        movl    %eax,%ecx
        popl    %eax
        addl    %ecx,%eax               #   +
        movl    -12(%ebp),%edx
        movl    %eax,-36(%edx)          # v3 :=
        movl    -12(%ebp),%edx
        movl    -36(%edx),%eax          #   v3
        movl    %eax,-32(%ebp)          # f2 :=
        movl    -32(%ebp),%eax          # Fetch return value
        leave                           # End of f2
        ret
proc$p1_2:
        enter   $36,$2                  # Start of p1
        movl    -8(%ebp),%edx
        movl    8(%edx),%eax            #   a1a
        pushl   %eax                    # Push param #1
        call    func$f2_3
        addl    $4,%esp                 # Pop parameters
        movl    -8(%ebp),%edx
        movl    %eax,-36(%edx)          # v2 :=
        movl    -8(%ebp),%edx
        movl    -36(%edx),%eax          #   v2
        pushl   %eax
        movl    -8(%ebp),%edx
        movl    12(%edx),%eax           #   a1b
        movl    %eax,%ecx
        popl    %eax
        imull   %ecx,%eax               #   *
        movl    -4(%ebp),%edx
        movl    %eax,-36(%edx)          # v1a :=
        leave                           # End of p1
        ret
prog$blokker_1:
        enter   $40,$1                  # Start of blokker
        movl    $7,%eax                 #   7
        pushl   %eax                    # Push param #2.
        movl    $3,%eax                 #   3
        negl    %eax                    #   - (prefix)
        pushl   %eax                    # Push param #1.
        call    proc$p1_2
        addl    $8,%esp                 # Pop params.
        movl    -4(%ebp),%edx
        movl    -36(%edx),%eax          #   v1a
        pushl   %eax                    # Push next param.
        call    write_int
        addl    $4,%esp                 # Pop param.
        movl    $10,%eax                #   10
        pushl   %eax                    # Push next param.
        call    write_char
        addl    $4,%esp                 # Pop param.
        leave                           # End of blokker
        ret
```

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

Kapittel 6

Programmeringsstil

6.1 Suns anbefalte Java-stil

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

6.1.1 Klasser

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

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

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

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

2) Alle `import`-spesifikasjonene.

3) JavaDoc-kommentar for klassen. (JavaDoc er beskrevet i avsnitt 7.1 på side 75.)

4) Selve klassen.

6.1.2 Variabler

Variabler bør deklareres én og én på hver linje:

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

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

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

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

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

if (uttrykk) {
    setninger;
}

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

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

return uttrykk;

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

case xxx:
    setninger;
    break;

default:
    setninger;
    break;
}

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

while (uttrykk) {
    setninger;
}
```

Figur 6.1: Suns forslag til hvordan setninger bør skrives

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

6.1.3 Setninger

Enkle setninger bør stå én og én på hver linje:

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

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

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

Tabell 6.1: Suns forslag til navnevalg i Java-programmer

6.1.4 Navn

Navn bør velges slik det er angitt i tabell 6.1.

6.1.5 Utseende

6.1.5.1 Linjelengde og linjedeling

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

En linje som er for lang, bør deles

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

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

6.1.5.2 Blanke linjer

Sett inn doble blanke linjer

- mellom klasser.

Sett inn enkle blanke linjer

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

6.1.5.3 Mellomrom

Sett inn mellomrom

- etter kommaer i parameterlister,
- rundt binære operatorer:

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

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

- ved typekonvertering:

```
(int) x
```

Kapittel 7

Dokumentasjon

7.1 JavaDoc

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

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

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

7.1.1 Hvordan skrive JavaDoc-kommentarer

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

En JavaDoc-kommentarer for en klasse ser slik ut:

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

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

JavaDoc-kommentarer for metoder følger nesten samme oppsettet:

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

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

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

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

7.1.2 Eksempel

I figur 7.1 kan vi se en Java-metode med dokumentasjon.

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

Figur 7.1: Java-kode med JavaDoc-kommentarer

7.2 «Lesbar programmering»

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

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

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

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

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

7.2.1 Et eksempel

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

1) Skriv kildefilen `bubble.w0` (vist i figur 7.2 og 7.3). Dette gjøres med en vanlig tekstbehandler som for eksempel Emacs.
2) Bruk programmet `weave0`[14] til å lage det ferdige dokumentet som er vist i figur 7.4–7.7:

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

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

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

[14] Dette eksemplet bruker Dags versjon av lesbar programmering kalt web_0; for mer informasjon, se http://dag.at.ifi.uio.no/public/doc/web0.pdf.

bubble.w0 del 1

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

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

\begin{document}
\maketitle

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

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


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

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

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

Figur 7.2: «Lesbar programmering» — kildefilen bubble.w0 del 1

bubble.w0 del 2

```
Each pass through the array consists of looking at
every pair of adjacent elements;\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 7.3: «Lesbar programmering» — kildefilen bubble.w0 del 2

Bubble sort

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

July 29, 2016

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

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

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

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

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

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

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

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

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

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

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

Figur 7.4: «Lesbar programmering» — utskrift side 1

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

#5 ⟨*swap a[i] and a[i+1]*⟩ ≡

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

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

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

#4a ⟨*local variables* #4 (p.1)⟩ +≡

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

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

#6 ⟨*initialize*⟩ ≡

```
17  n_swaps = 0;
```

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

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

#7 ⟨*not sorted*⟩ ≡

```
18  n_swaps > 0
```

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

File: *bubble.w0* page 2

Figur 7.5: «Lesbar programmering» — utskrift side 2

Variables

A

a 1, 12, 14

I

i 11, 12, 13, 14

N

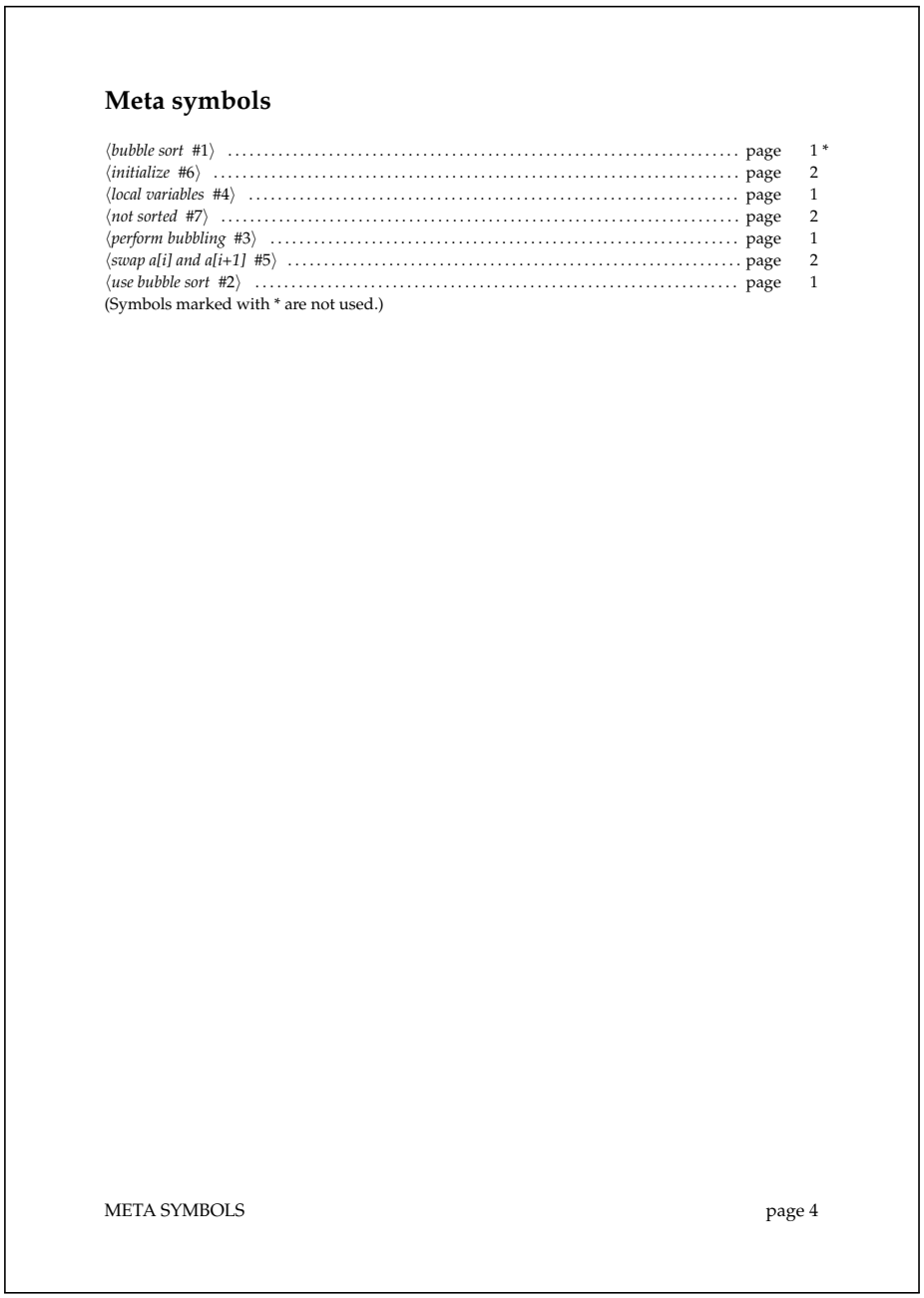
n 1, 11
n_swaps 15, 16, 17, 18

T

temp 14, 16

VARIABLES page 3

Figur 7.6: «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 7.7: «Lesbar programmering» — utskrift side 4