Et eksempel

```
program Mini;
begin
   write('x');
end.
```

```
# Code file created by Pascal2100 compiler 2015-11-03 23:27:24
        .extern write_char
        .extern write_int
        .extern write_string
        .globl  _main
        .globl  main
_main:
main:   call    prog$mini_1             # Start program
        movl    $0,%eax                 # Set status 0 and
        ret                             # terminate the program
prog$mini_1:
        enter   $32,$1                  # Start of mini
        movl    $120,%eax               #   char 120
        pushl   %eax                    # Push param #1.
        call    write_char
        addl    $4,%esp                 # Pop parameter.
        leave                           # End of mini
        ret
```

Klassen Main.CodeFile

```
package no.uio.ifi.pascal2100.main;

import java.io.*;
import java.text.SimpleDateFormat;
import java.util.Date;

public class CodeFile {
    private String codeFileName;
    private PrintWriter code;
    private int numLabels = 0;

    CodeFile(String fName) {
        codeFileName = fName;
        try {
            code = new PrintWriter(fName);
        } catch (FileNotFoundException e) {
            Main.error("Cannot create code file " + fName + "!");
        }
        code.println("# Code file created by Pascal2100 compiler " +
            new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date()));
    }
```

```
public void genInstr(String lab, String instr,
                     String arg, String comment) {
    printLabel(lab, (instr+arg+comment).equals(""));
    code.printf("%-7s %-23s ", instr, arg);
    if (comment.length() > 0) {
        code.print("# " + comment);
    }
    code.println();
}
```

Anta at vi har Pascal2100-koden **`v := 1 + 2;`**.

Disse x86-instruksjonene gjør dette:

```
movl    $1,%eax          # %EAX=1  %ECX=? %EDX=? stack=...
pushl   %eax             # %EAX=1  %ECX=? %EDX=? stack=1 ...
movl    $2,%eax          # %EAX=2  %ECX=? %EDX=? stack=1 ...
movl    %eax,%ecx        # %EAX=2  %ECX=2 %EDX=? stack=1 ...
popl    %eax             # %EAX=1  %ECX=2 %EDX=? stack=...
addl    %ecx,%eax        # %EAX=3  %ECX=2 %EDX=? stack=...
movl    -4(%ebp),%edx    # %EAX=3  %ECX=2 %EDX=ba stack=...
movl    %eax,-36(%edx)   # v = 3
```

En presisering

Det finnes mange mulige kodebiter som gjør det samme. I kompendiet står angitt ganske nøyaktig hvilke som skal brukes.

NB!

Det er viktigere at koden er riktig enn at den er rask!

Hvordan implementere kodegenerering

Det beste er å følge samme opplegg som for å sjekke programkoden:

Kodegenerering

Legg en metode `genCode` inn i alle klasser som representerer en del av Pascal2100-programmet (dvs er subklasse av `PascalSyntax`).

Metoden 'genCode'

```
class WhileStatm extends Statement {
    Expression test;
    Statement body;

    @Override void check(Block curScope, Library lib) {
        :
    }

    @Override void genCode(CodeFile f) {
        :
    }

    static WhileStatm parse(Scanner s) {
        :
    }

    @Override void prettyPrint() {
        :
    }
}
```

Konvensjoner

Kodegenerering blir mye enklere om vi setter opp noen fornuftige konvensjoner:

- Alle beregninger skal ende opp i %EAX.
- %ECX og %EDX er hjelperegistre.
- Hovedstakken (aksessert via %ESP) er til
  - variabler (neste uke)
  - mellomresultater
  - funksjons- og prosedyrekall (neste uke)

I kompendiet finnes kodeskjemaer for det meste i Pascal2100. Skjemaet for while-setningen ser slik ut:

```
while ⟨e⟩ do ⟨S⟩
```

⇒

```
⟨lab1⟩:
      ⟨Beregn ⟨e⟩ med svar i %EAX⟩
      cmpl    $0,%eax
      je      ⟨lab2⟩
      ⟨S⟩
      jmp     ⟨lab1⟩
⟨lab2⟩:
```

Lokale navnelapper

Når vi skal lage slike hopp, trenger vi stadig nye navnelapper. Dette kan vi få fra `CodeFile`-objektet:

```
public class CodeFile {

    private int numLabels = 0;


    public String getLocalLabel() {
        return String.format(".L%04d", ++numLabels);
    }
```

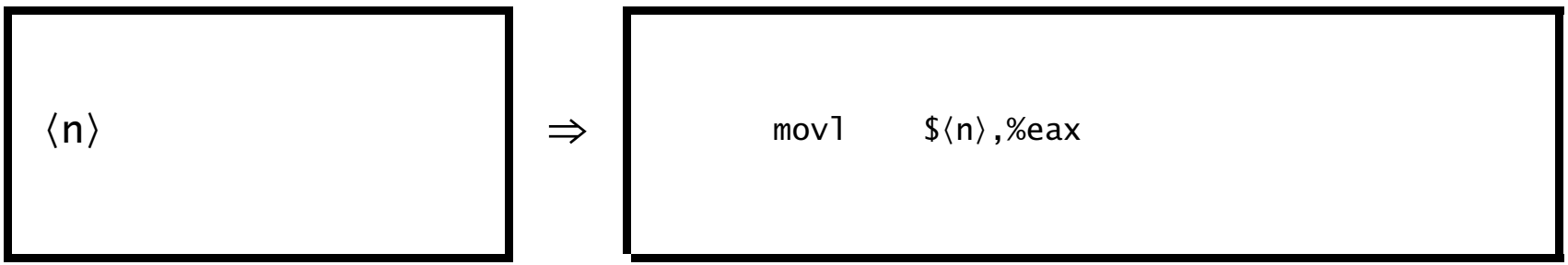
Hele koden for `WhileStatm`

```
class WhileStatm extends Statement {
    Expression expr;
    Statement body;


    @Override void genCode(CodeFile f) {
        String testLabel = f.getLocalLabel(),
               endLabel  = f.getLocalLabel();

        f.genInstr(testLabel, "", "", "Start while-statement");
        expr.genCode(f);
        f.genInstr("", "cmpl", "$0,%eax", "");
        f.genInstr("", "je", endLabel, "");
        body.genCode(f);
        f.genInstr("", "jmp", testLabel, "");
        f.genInstr(endLabel, "", "", "End while-statement");
    }
```

Uttrykk

Uttrykk (og alle deluttrykk) skal resultere i en verdi i %EAX-registeret.

Konstanter



Dette fungerer også for char-literaler; da bruker vi ASCII-verdien.

Enum-literaler

Hva med enum-literaler som i

```
type Sex = (Male, Female);
      :
... Female ...
```

Vi må bare gi dem en heltallsrepresentasjon: `Male`=0, `Female`=1.

NB!

Resten av koden forutsetter `False`=0 og `True`=1.

Operatorer

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

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

Neste uke

Til neste uke tar vi det som står igjen:

- blokker
- variabler
- funksjons- og prosedyrekall
- biblioteket
- hovedprogrammet