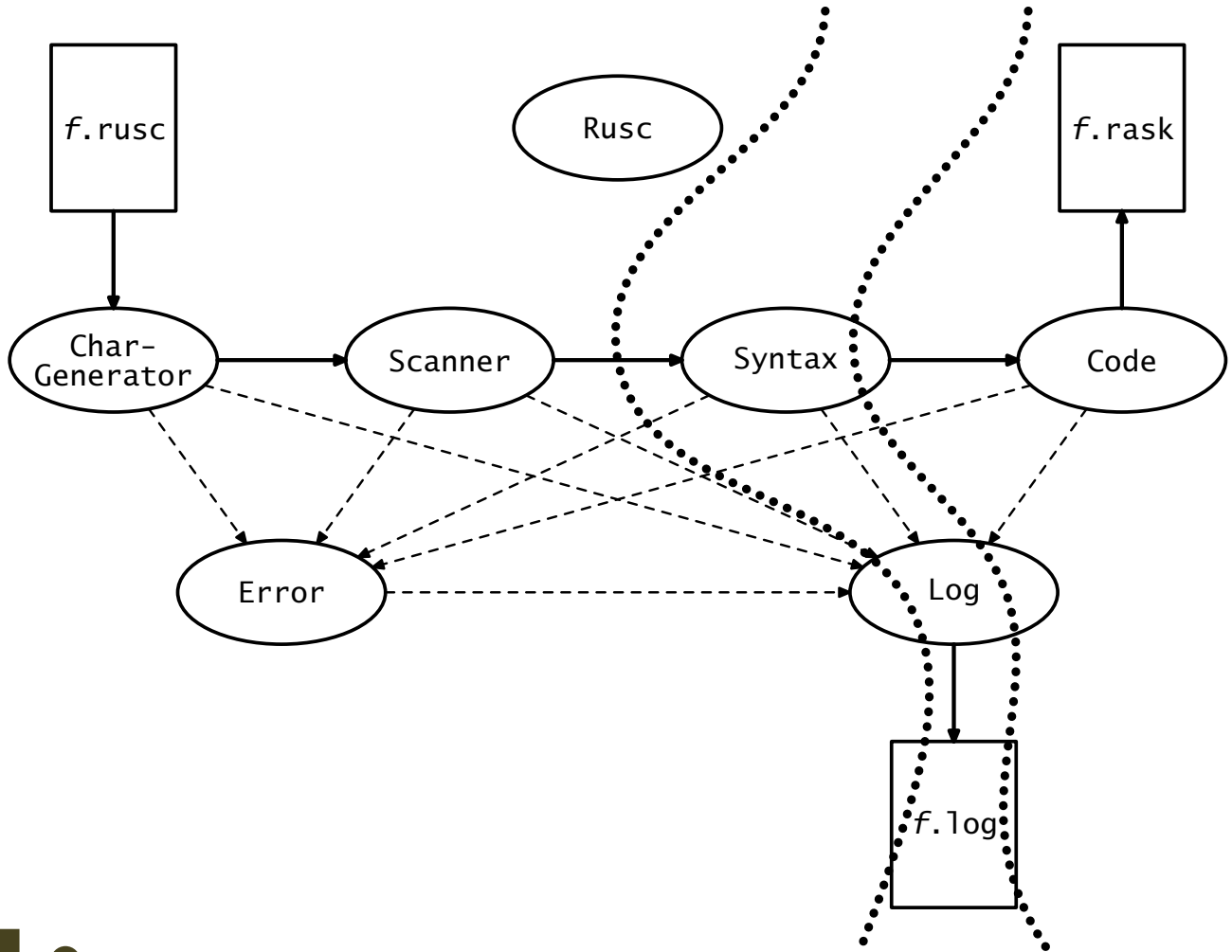
Dagens tema:

Kodegenerering

- Introduksjon
- Modulen Code
- Enkle variable
- Noen enkle setninger
- Uttrykk
- Litt mer kompliserte setninger med betingelser

(Alt unntatt program, funksjoner og array-er.)

Del-0 **Del-1** **Del-2**



Formålet

Formålet med kodegenereringen er å lage Rasko-kode som Rask-maskinen kan utføre:

Inndata er trerepresentasjonen av RusC-programmet laget i del 1.

Utdata er en fil med Rasko-kode.

Anta at vi har RusC-koden

```
putchar('?');
```

Disse Rask-instruksjonene gjør dette:

```
SET     R11,63
CALL    9993
```

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!

Rask-maskinen

Minne:

Inneholder 10 000 celler som kan inneholde enten data eller instruksjoner.

33 registre:

R_0 inneholder alltid 0.

R_1–R_{31} inneholder data.

PC gir adressen til neste instruksjon.

Instruksjonene

Nr	Navn	Operasjon
1	LOAD RA, RB, C	RA ← Mem[RB+C]
2	SET RA, RB, C	RA ← RB+C
3	STORE RA, RB, C	Mem[RB+C] ← RA
4	ADD RA, RB, RC	RA ← RB+RC
5	SUB RA, RB, RC	RA ← RB−RC
6	MUL RA, RB, RC	RA ← RB×RC
7	DIV RA, RB, RC	RA ← RB/RC

Nr	Navn	Operasjon
8	EQ RA, RB, RC	RA ← 1 hvis RB=RC, ellers 0
9	NEQ RA, RB, RC	RA ← 1 hvis RB≠RC, ellers 0
10	LESS RA, RB, RC	RA ← 1 hvis RB<RC, ellers 0
11	LESSEQ RA, RB, RC	RA ← 1 hvis RB≤RC, ellers 0
12	GTR RA, RB, RC	RA ← 1 hvis RB>RC, ellers 0
13	GTREQ RA, RB, RC	RA ← 1 hvis RB≥RC, ellers 0
14	JUMPEQ RA, RB, C	Hvis RA=RB, PC ← C
15	JUMPNEQ RA, RB, C	Hvis RA≠RB, PC ← C
16	CALL RA, RB, C	R31 ← PC og PC ← C
17	RET	PC ← R31

RusC-programmet

```
int pot2 (int x)
{
   int p2;   p2 = 1;
   while (2*p2 <= x) { p2 = 2*p2; }
   return p2;
}

int x;

int main ()
{
   int v;   v = getint();
   x = pot2(v);   putint(x);   putchar(10);
}
```

. . . skal bli Rask-koden

```
#! /local/bin/rask

1600000000000035 211000000000000 1600000000009990
0 0 0 331000000000003
303000000000004 311000000000005 201000000000001 3
201000000000002 203010000000000 101000000000006 6
203010000000000 101000000000005 1101030000000001
201000000000002 203010000000000 101000000000006 6
301000000000006 1400000000000012 101000000000006
103000000000004 131000000000003 1700000000000000
0 0 0 331000000000032
303000000000033 1600000000009992 301000000000034
1600000000000007 301000000000031 111000000000031
211000000000010 1600000000009993 103000000000033
1700000000000000
```

Genererte instruksjoner

Til hjelp under uttesting finnes opsjonen **-logC** som lar kompilatoren fortelle hvilken kode den lager:

```
Code    0: 1600000000000000  CALL      0  0          0  # main();
Code    1:  211000000000000  SET      11  0          0  #     (0)
Code    2: 1600000000009990  CALL      0  0       9990  # exit
Code    3:                   RES                    1  # Return address in pot2
Code    4:                   RES                    1  # Refuge for R3 in pot2
Code    5:                   RES                    1  # int x
Code    6:                   RES                    1  # int p2
Code    7:  331000000000003  STORE    31  0          3  # Save return address
Code    8:  303000000000004  STORE     3  0          4  # Save R3
Code    9:  311000000000005  STORE    11  0          5  # Save parameter x
Code   10:  201000000000001  SET       1  0          1  # R1 = 1
Code   11:  301000000000006  STORE     1  0          6  # p2 =
Code   12:  201000000000002  SET       1  0          2  # R1 = 2
Code   13:  203010000000000  SET       3  1          0  # Save operand in R3
Code   14:  101000000000006  LOAD      1  0          6  # R1 = p2
Code   15:  601030000000001  MUL       1  3          1  #   *
Code   16:  203010000000000  SET       3  1          0  # Save operand in R3
Code   17:  101000000000005  LOAD      1  0          5  # R1 = x
Code   18: 1101030000000001  LESSEQ    1  3          1  #   <=
Code   19: 1401000000000000  JUMPEQ    1  0          0  # Exit while-loop when test is fals
```

Opsjonen «-logC»

```
Code   20:  201000000000002  SET      1  0          2  # R1 = 2
Code   21:  203010000000000  SET      3  1          0  # Save operand in R3
Code   22:  101000000000006  LOAD     1  0          6  # R1 = p2
Code   23:  601030000000001  MUL      1  3          1  #   *
Code   24:  301000000000006  STORE    1  0          6  # p2 =
Code   25: 1400000000000012  JUMPEQ   0  0         12  # Repeat while-loop
--->   19: 1401000000000026                        26  # Update break address
Code   26:  101000000000006  LOAD     1  0          6  # R1 = p2
Code   27: 1400000000000000  JUMPEQ   0  0          0  # return
--->   27: 1400000000000028                        28  # Update address of return jump
Code   28:  103000000000004  LOAD     3  0          4  # Restore R3
Code   29:  131000000000003  LOAD    31  0          3  # Restore return address
Code   30: 1700000000000000  RET      0  0          0  # Return from pot2
Code   31:                   RES                    1  # int x
Code   32:                   RES                    1  # Return address in main
Code   33:                   RES                    1  # Refuge for R3 in main
Code   34:                   RES                    1  # int v
Code   35:  331000000000032  STORE   31  0         32  # Save return address
Code   36:  303000000000033  STORE    3  0         33  # Save R3
Code   37: 1600000000009992  CALL     0  0       9992  # Call getint(...)
```

Opsjonen «-logC»

```
Code    38:  3010000000000034  STORE     1  0          34  # v =
Code    39:  1110000000000034  LOAD     11  0          34  # R11 = v
Code    40: 1600000000000007   CALL      0  0           7  # Call pot2(...)
Code    41:  3010000000000031  STORE     1  0          31  # x =
Code    42:  1110000000000031  LOAD     11  0          31  # R11 = x
Code    43: 1600000000009994   CALL      0  0        9994  # Call putint(...)
Code    44:  2110000000000010  SET      11  0          10  # R11 = 10
Code    45: 1600000000009993   CALL      0  0        9993  # Call putchar(...)
Code    46:  1030000000000033  LOAD      3  0          33  # Restore R3
Code    47:  1310000000000032  LOAD     31  0          32  # Restore return address
Code    48: 1700000000000000   RET       0  0           0  # Return from main
--->     0: 1600000000000035                            35  # Fix call to main()
```

Å implementere denne opsjonen er en del av den obligatoriske oppgaven.

Hvordan implementere kodegenerering

Det beste er å følge samme opplegg som for å skrive ut programkoden i del 1:

Kodegenerering

Legg en metode genCode inn i alle klasser som representerer en del av RusC-programmet.

Metoden 'genCode'

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

    public static WhileStatm parse(DeclList decls) {
        ⋮
    }

    public void genCode() {
        ⋮
    }


    public void printTree() {
        ⋮
    }
}
```

Modulen Code

Modulen Code tar seg av kodegenereringen; all koden lagres først i vektoren mem:

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

import java.io.*;
import no.uio.ifi.rusc.error.Error;
import no.uio.ifi.rusc.log.Log;
import no.uio.ifi.rusc.rusc.Rusc;

public class Code {
    public static final int codeSize = 10000;
    public static int curAddr = 0;

    private static long mem[];

    public static void init() {
        //-- Må endres i del 2:
    }

    :
}
```

Metoden finish skriver ut den ferdige Rasko-kodefilen:

```
public static void finish() {
    String codeName = Rusc.sourceName;
    if (codeName == null) return;

    if (codeName.endsWith(".rusc"))
        codeName = codeName.substring(0,codeName.length()-5);
    codeName += ".rask";

    PrintWriter f = null;
    try {
        f = new PrintWriter(codeName);
    } catch (FileNotFoundException e) {
        Error.error("Cannot create code file " + codeName + "!");
    }
    f. println("#! /local/bin/rask");   f.println();

    for (int ix = 0;  ix < curAddr;  ++ix) {
        f.print(mem[ix]);
        if (ix%5 == 4) f.println();
        else           f.print(" ");
    }
    f.println();   f.close();
}
```

De enkelte instruksjonene lagres med genInstr:

```
    /**
     * Puts an instruction into Rask's memory.
     *
     * @param  instrName  The instruction's name (for the log file)
     * @param  instrNo    The instruction's code number
     * @param  op1        Instruction operand #1
     * @param  op2        Instruction operand #2
     * @param  op3        Instruction operand #3
     * @param  comment    A comment to add to the log file
     * @return            The memory address of the generated instructio
     */
private static int genInstr(String instrName, int instrNo,
                            int op1, int op2, int op3, String comment) {
    if (curAddr >= codeSize-10)
        Error.error("Memory overflow! More than " + (codeSize-10) +
                    " instructions/data words generated!");

    Log.noteCode(curAddr, instrName, instrNo, op1, op2, op3, comment);
    mem[curAddr++] = instrNo*1000000000000L + op1*10000000000L +
        op2*10000000000L + op3;
    return curAddr-1;
}
```

Så finnes det en metode for hver instruksjon:

```
public static int genSet(int op1, int op2, int op3, String comment) {
    return genInstr("SET", 2, op1, op2, op3, comment);
}

public static int genRet(int op1, int op2, int op3, String comment) {
    return genInstr("RET", 17, op1, op2, op3, comment);
}
```

Enkle variable

Det må settes av plass i minnet til alle enkle[1] variable.

Viktig!

- Data og instruksjoner ligger i det samme minnet.
- Datamaskinen Rask vet ikke hva som er hva – det må vi som genererer kode, passe på!

[1] Array-er er tema neste uke.

Reservasjon av variable løses med en metode i Code:

```
public class Code {
    public static final int codeSize = 10000;
    public static int curAddr = 0;

    private static long mem[];

    public static int resMem(int nWords, String comment) {
        if (curAddr+nWords > codeSize-10)
            Error.error("Memory overflow! More than " + (codeSize-10) +
                        " instructions/data words generated!");

        Log.noteRes(curAddr, nWords, comment);
        curAddr += nWords;
        return curAddr-nWords;
    }
    :
```

Setninger

I kompendiet finner vi kodeskjemaer for alle setningene.

NB!

Disse kodeskjemaene *skal* brukes, selv om de ikke gir optimal kode!

Assignment

⟨v⟩ = ⟨e⟩ ⇒

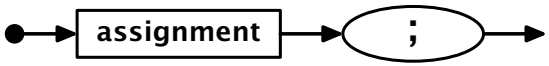
```
⟨Beregn ⟨e⟩, svar i R1⟩
STORE   R1,R0,Adr(⟨v⟩)
```

Registerbruk

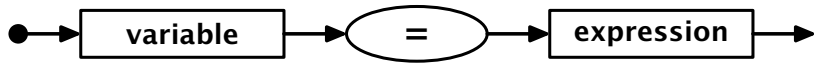
For å holde oversikt *skal* registrene brukes slik:

R_0	Alltid 0 (gitt av Rask)
R_1	Hovedregister for beregning av uttrykk
R_2	Hjelperegister ved vektoraksess
R_3	Hjelperegister ved beregning av uttrykk
R_{11}	Parameter 1 ved funksjonskall
R_{12}	Parameter 2 ved funksjonskall
R_{13}	Parameter 3 ved funksjonskall
R_{14}	Parameter 4 ved funksjonskall
R_{31}	Returadresse ved funksjonskall (gitt av Rask)

assign-statm



assignment



Kodegenereringsmetoden for AssignStatm blir:

```
class AssignStatm extends Statement {
    private Assignment assignment;

    public void genCode() {
        assignment.genCode();
    }

    ⋮
}
```

```
class Assignment extends SyntaxUnit {
    Variable lhs;       /* "Left hand side" */
    Expression rhs;     /* "Right hand side" */

    public void genCode() {
        rhs.genCode();
        if (⟨er et array-element⟩) {
          :
        } else {
            Code.genStore(1, 0, lhs.declRef.memAddr,
                          lhs.declRef.name+" = ");
        }
    }
    :
}
```

Uttrykk

Hvordan lager vi kode for et uttrykk som

*4 * a - 17?*

Første operand

Det er ganske enkelt å lage kode som legger en variabel i R_1-registeret:

⟨v⟩ ⇒ `LOAD R1,R0,Adr(⟨v⟩)`

Det er nesten like enkelt for konstanter (tall eller tegn); vi må bare spesialbehandle negative tall. (Hvorfor det?)

⟨n⟩	⇒	`SET R1,R0,⟨n⟩`
-⟨n⟩	⇒	`SET R1,R0,⟨n⟩` `SUB R1,R0,R1`

Resten av uttrykket

Resten av uttrykket består av 0 eller flere par av ⟨operator,konstant/variabel⟩ som vi kan ta etter tur. Slik oversetter vi +:

$\langle e_1 \rangle + \langle e_2 \rangle$	$\Rightarrow$	⟨Beregn ⟨e_1⟩, svar i R_1⟩ SET R3,R1,0 ⟨Beregn ⟨e_2⟩, svar i R_1⟩ ADD R1,R3,R1

Ved at alle operander er innom R_1 blir det enklere å skrive kodegenereringen (men koden blir ikke fullt så rask).

```
class Expression extends SyntaxUnit {
    ExprElement firstElem = null;

    public void genCode() {
        firstElem.genCode();
        ExprElement e = firstElem.nextElem;
        while (e != null) {
            Code.genSet(3, 1, 0, "Save operand in R3");
            e.nextElem.genCode();
            e.genCode();
            e = e.nextElem.nextElem;
        }
    }
    :
```

Setninger

Noen setninger er litt mer kompliserte å generere kode for.

While-setningen

```
while (⟨e⟩) {
    ⟨S⟩
}
```

⇒

```
Lw:   ⟨Beregn ⟨e⟩, svar i R1⟩
      JUMPEQ  R1,R0,Lx
      ⟨S⟩
      JUMPEQ  R0,R0,Lw
Lx:
```

RusC-koden

```
while (a < 10) {
   a = a + 1;
}
```

skal generere denne koden:

```
Code   10:  101000000000005  LOAD      1  0   5
Code   11:  203010000000000  SET       3  1   0
Code   12:  201000000000010  SET       1  0  10
Code   13: 1001030000000001  LESS      1  3   1
Code   14: 1401000000000021  JUMPEQ    1  0  21
Code   15:  101000000000005  LOAD      1  0   5
Code   16:  203010000000000  SET       3  1   0
Code   17:  201000000000001  SET       1  0   1
Code   18:  401030000000001  ADD       1  3   1
Code   19:  301000000000005  STORE     1  0   5
Code   20: 1400000000000010  JUMPEQ    0  0  10
```

Hoppadresser

Koden inneholder to hopp:

```
Code    10:   1010000000000005   LOAD      1   0    5
Code    11:   2030100000000000   SET       3   1    0
Code    12:   2010000000000010   SET       1   0   10
Code    13:   1001030000000001   LESS      1   3    1
Code    14:   1401000000000021   JUMPEQ    1   0   21
Code    15:   1010000000000005   LOAD      1   0    5
Code    16:   2030100000000000   SET       3   1    0
Code    17:   2010000000000001   SET       1   0    1
Code    18:   4010300000000001   ADD       1   3    1
Code    19:   3010000000000005   STORE     1   0    5
Code    20:   1400000000000010   JUMPEQ    0   0   10
```

Hopp til kjent adresse

Den siste hoppet er greit nok: Bare husk adressen da man begynte å lage kode for setningen.

Hopp til ukjent adresse

Hva med det første hoppet? Hvor skal vi hoppe?
Løsningen er:

1. Generér en hoppinstruksjon til adresse 0.
2. Når vi en stund senere vet den riktige adressen, kan vi sette inn denne i stedet.

Hele koden

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

    public void genCode() {
        int startAddr = Code.curAddr;

        test.genCode();
        int jumpAddr = Code.genJumpEq(1, 0, 0,
                "Exit while-loop when test is false");

        body.genCode();
        Code.genJumpEq(0, 0, startAddr,
                "Repeat while-loop");

        Code.updateInstr(jumpAddr, Code.curAddr,
            "Update break address");
    }

    :
```

I dag

- kodegenerering generelt
- kodegenerering for
  - uttrykk
  - tilordning
  - while-setninger

Neste uke

- Kodegenerering for
  - array-er
  - funksjoner og funksjonskall
  - hovedprogrammet