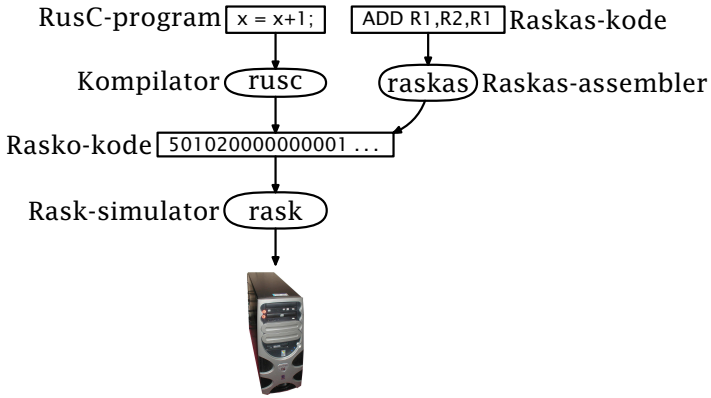
Dagens tema:

- Hvordan oversette kode
  - Hva skal kompilatoren gjøre?
  - Enkel oversettelse
  - Array-er
  - Funksjoner
- Prosjektet
  - Struktur
  - Javas pakker
  - Skanneren
  - Enum-klasser i Java
  - Testutskrifter

Hva skal gjøres?



Hva gjør en kompilator?

I mange tilfelle er det svært enkelt å oversette fra et høynivåspråk til maskinkode:

```
i = 5;
```

```
SET     R1,R0,5
STORE   R1,R0,i
```

Hva gjør en kompilator?

I mange tilfelle er det svært enkelt å oversette fra et høynivåspråk til maskinkode:

```
i = 5;
```

```
SET     R1,R0,5
STORE   R1,R0,i
```

Noen ganger det ikke fullt så innlysende:

```
a[i] = 2 * i + f(x);
```

```
SET     R1,R0,2
 ⋮
STORE   R1,R2,a
```

(Her vil min kompilator generere 11 instruksjoner.)

Forenkling

Eksemplet på forrige lysark *kunne* vært skrevet med færre instruksjoner. Det skal vi *ikke* gjøre i dette kurset!

- Det aller viktigste er at koden er riktig.
- Optimalisering i en kompilator er altfor komplisert for dette kurset.

Array-er

Hvordan kan vi bruke array-er?

1. Plassér indeksen i et register R_x.
2. Bruk så

```
LOAD    R1,Rx,adressen til array-en
STORE   R1,Rx,adressen til array-en
```


```
Code    28:  101000000000016  LOAD     1  0        16
Code    29:  203010000000000  SET      3  1         0
Code    30:  102000000000015  LOAD     2  0        15
Code    31:  101020000000003  LOAD     1  2         3
Code    32:  401030000000001  ADD      1  3         1
Code    33:  301000000000016  STORE    1  0        16
Code    34:  101000000000015  LOAD     1  0        15
Code    35:  203010000000000  SET      3  1         0
Code    36:  201000000000001  SET      1  0         1
Code    37:  401030000000001  ADD      1  3         1
Code    38:  301000000000015  STORE    1  0        15
```

Konvensjoner

Velvalgte konvensjoner gjør det mye enklere å programmere riktig.

- Uttrykk (og deluttrykk) plasserer resultatet i R_1.
- R_3 benyttes til mellomlagring.
- R_2 benyttes til array-oppslag.

Funksjoner

Hvordan programmerer man funksjoner? Man trenger:

1. En måte å hoppe til prosedyren
2. En måte å komme tilbake
3. Mulighet for å overføre parametre
4. Mulighet til å få tilbake returverdi


```
Code    17:   331000000000013  STORE    31  0          13
Code    18:   303000000000014  STORE     3  0          14

  ⋮

Code    42:   103000000000014  LOAD      3  0          14
Code    43:   131000000000013  LOAD     31  0          13
Code    44: 1700000000000000   RET       0  0           0
Code    45:                    RES                     1
Code    46:                    RES                     1
Code    47:   331000000000045  STORE    31  0          45
Code    48:   303000000000046  STORE     3  0          46
Code    49: 1600000000000017   CALL      0  0          17
Code    50:   103000000000046  LOAD      3  0          46
Code    51:   131000000000045  LOAD     31  0          45
Code    52: 1700000000000000   RET       0  0           0
```


Konvensjoner

I vår kompilator brukes følgende løsning:

- For å kalle en prosedyre, hopper man til den med instruksjonen CALL som legger PC-registeret i R_{31}-registeret før den hopper.
- Parametre overføres i R_{11}, R_{12}, R_{13} og R_{14}.
- Returverdien legges i R_1.
- Ved return hopper man tilbake med en RET.
- Verdiene i R_3- og R_{31}-registrene må gjemmes unna mens man utfører innmaten i prosedyren.

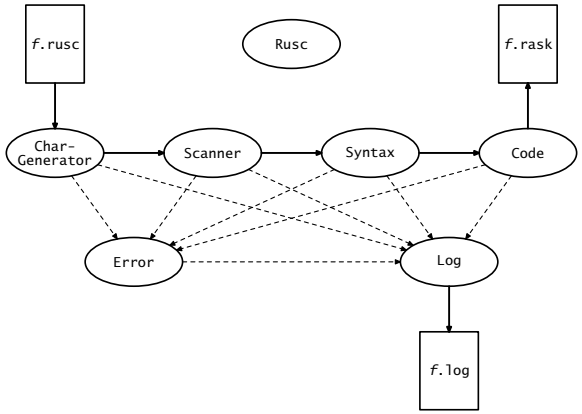
Prosjektet

Hvordan skriver man et større program som en kompilator?

Struktur

En fornuftig måte å dele opp problemet på er å se på hvor data flyter. Da er det enklere å kapsle inn deler av programmet.

Vår struktur



Alle filene som skal inngå i en Java-pakke starter med «package *navn*».

Pakkenavn bør bestå av opphavsstedets internettadresse (baklengs) og et lokalt navn. Vårt kompilatorprosjekt heter

no.uio.ifi.rusc

Noen programmeringsspråk har mekanismer for store moduler – men langt fra alle. Java har package.

Begrunnelse

Anta at vi skal utvikle et CAD-system. Et firma i India har laget en god GUI-modul.
Men, begge har en klasse Point.

Moduler kan løse dette problemet.

Eksempel

Filen P1/A.java:

```
package P1;

public class A {
    public static int x = 1;
}
```

(Under kompileringen må klassen ligge i et fil-tre som tilsvarer leddene i pakkenavnet; våre filer ligger i

no/uio/ifi/rusc/scanner/Scanner.java

og tilsvarende.)

Vi kan hente klasser fra alle pakker så lenge de finnes i CLASSPATH:

```
class B {
    public static void main (String arg[]) {
        System.out.println("P1.A.x = " + P1.A.x);
    }
}
```

Beskyttelse

Klasser kan beskyttes:

- **public** kan brukes fra andre pakker.
- — er usynlig utenfor pakken.

For klasseelementer gjelder:

- **private** er bare tilgjengelige i klassen.
- **protected** er for klassen og subklasser.
- — er bare for bruk innen pakken.
- **public** kan benyttes overalt.

For å unngå å skrive mange lange navn, kan vi importere klasser fra pakker:

```
import P1.A;

class B {
    public static void main (String arg[]) {
        System.out.println("P1.A.x = " + A.x);
    }
}
```

Vi kan også importere alle klassene fra en pakke:

```
import P1.*;

class B {
    public static void main (String arg[]) {
        System.out.println("P1.A.x = " + A.x);
    }
}
```

Hva kan en pakke inneholde?

En pakke kan bare inneholde klasser. Vi trenger også data og vedtar derfor:

- I alle pakker finnes en klasse med samme navn som pakken (men stor forbokstav); den inneholder data og metoder vi trenger.
- Alle disse klassene har disse to metodene:

  `init` benyttes til initiering av pakken.

  `finish` avslutter pakken.


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

 * @author dag@ifi.uio.no
 */
public class Rusc {
    public static String sourceName = null;

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

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


Hovedprogrammet

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

/*
 * class Rusc
 *
 * version 1.0 dated 13 August 2007
 *
 * © 2007 Ifi/UiO
 */

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



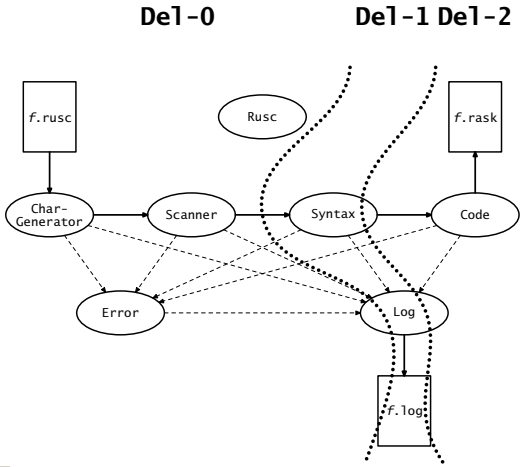
```
            if (opt.equals("-logC")) {
                Log.doLogCode = true;
            } else if (opt.equals("-logP")) {
                Log.doLogParser = true;
            } else if (opt.equals("-logT")) {
                Log.doLogTree = true;
            } else if (opt.equals("-logS")) {
   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

```
        Error.init();   Log.init();  Code.init();
        CharGenerator.init();  Scanner.init();  Syntax.init();

        if (testScanner) {
            while (Scanner.nextToken != Token.eofToken) Scanner.readNext();
        } else {
            Syntax.parseProgram();
            if (Log.doLogTree) Syntax.printProgram();
            if (! testParser) Syntax.genCode();
        }

        Syntax.finish();  Scanner.finish();  CharGenerator.finish();
        Code.finish();  Log.finish();   Error.finish();
    }
}
```

Oppdelingen av prosjektet



Skanner

En kompilator *kan* lese og tolke en program tegn for tegn, men det er mye lettere om det kan gjøres *symbol for symbol*. Dette ordner en **skanner**.

CharGenerator leser programkoden linje for linje og deler linjen opp i enkelt-tegn.

Scanner setter tegnene sammen til symboler (og fjerne kommentarer).


```
/* Program som leser et tall v
   og skriver ut v+1. */

func main ()
{
   int v;

   /* Les data: */
   v = getint() + 1;
   /* Skriv svaret: */
   putint(v);   putchar(10);
}
```

har disse symbolene:

func	main	(	)	{	int	v	;	v	=			
getint	(	)	+	1	;	putint	(	v	)	;		
putchar	(	10	)	;	}							

Enum-klasser

Noen ganger har man data som kun kan ha et begrenset antall fast definerte verdier:

Kortfarge Kløver, ruter, hjerter, spar

Tippetegn Hjemmeseier, uavgjort, borteseier

Å representere disse med heltall er en halvgod løsning.

Slik brukes denne klassen:

```
class Tipping {
    public static void main (String arg[]) {
        Tippetegn rekke[] = new Tippetegn[12+1];

        rekke[1] = Tippetegn.Hjemmeseier;
        rekke[2] = Tippetegn.Borteseier;
        rekke[3] = Tippetegn.Borteseier;

        for (int i = 1;  i <= 3;  ++i)
            System.out.print(rekke[i].Tegn());
        System.out.println();
    }
}
```

Java tilbyr **enum-klasser**:

```
enum Tippetegn {
    Hjemmeseier, Uavgjort, Borteseier;

    char Tegn () {
        return toString().charAt(0);
    }
}
```

Man oppgir alle verdiene og kan legge inn ekstra metoder og data.

I vår skanner

Vår skanner kan levere følgende **token**:

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

/*
 * class Token
 */

/**
 * The different kinds of tokens read by Scanner.
 */
public enum Token { addToken, assignToken, commaToken, divideToken, elseToken,
    eofToken, equalToken, funcToken, greaterEqualToken, greaterToken, ifToken,
    intToken, leftBracketToken, leftCurlToken, leftParToken, lessEqualToken,
    lessToken, multiplyToken, nameToken, notEqualToken, noToken, numberToken,
    rightBracketToken, rightCurlToken, rightParToken, returnToken,
    semicolonToken, subtractToken, whileToken;

    public static boolean isOperator(Token t) {
        //-- Må endres i del 0:
        return false;
    }
}
```

Testutskrifter

Alle vil gjøre feil under arbeidet med kompilatoren. For enklere å oppdage feilene når de skjer, skal vi bygge inn ulike testutskrifter som brukeren enkelt kan slå på:

Opsjon	Hva dumpes?	Del
-logC	Kodegenereringen	2
-logP	Parseringen	1
-logS	Skanneren	0
-logT	Lagret parseringstre	1

> rusc -testscanner mini.rusc

```
  1: /* Program som leser et tall v
  2:    og skriver ut v+1. */
  3:
  4: func main ()
Scanner:  funcToken
Scanner:  nameToken main
Scanner:  leftParToken
Scanner:  rightParToken
  5: {
Scanner:  leftCurlToken
  6:    int v;
Scanner:  intToken
Scanner:  nameToken v
Scanner:  semicolonToken
  7:
  8:    /* Les data: */
  9:    v = getint() + 1;
Scanner:  nameToken v
Scanner:  assignToken
Scanner:  nameToken getint
Scanner:  leftParToken
Scanner:  rightParToken
Scanner:  addToken
Scanner:  numberToken 1
Scanner:  semicolonToken
 10:    /* Skriv svaret: */
 11:    putint(v);   putchar(10);
Scanner:  nameToken putint
Scanner:  leftParToken
Scanner:  nameToken v
Scanner:  rightParToken
Scanner:  semicolonToken
Scanner:  nameToken putchar
Scanner:  leftParToken
Scanner:  numberToken 10
Scanner:  rightParToken
Scanner:  semicolonToken
 12: }
Scanner:  rightCurlToken
Scanner:  eofToken
```