# Dagens tema:

## Semantisk sjekking

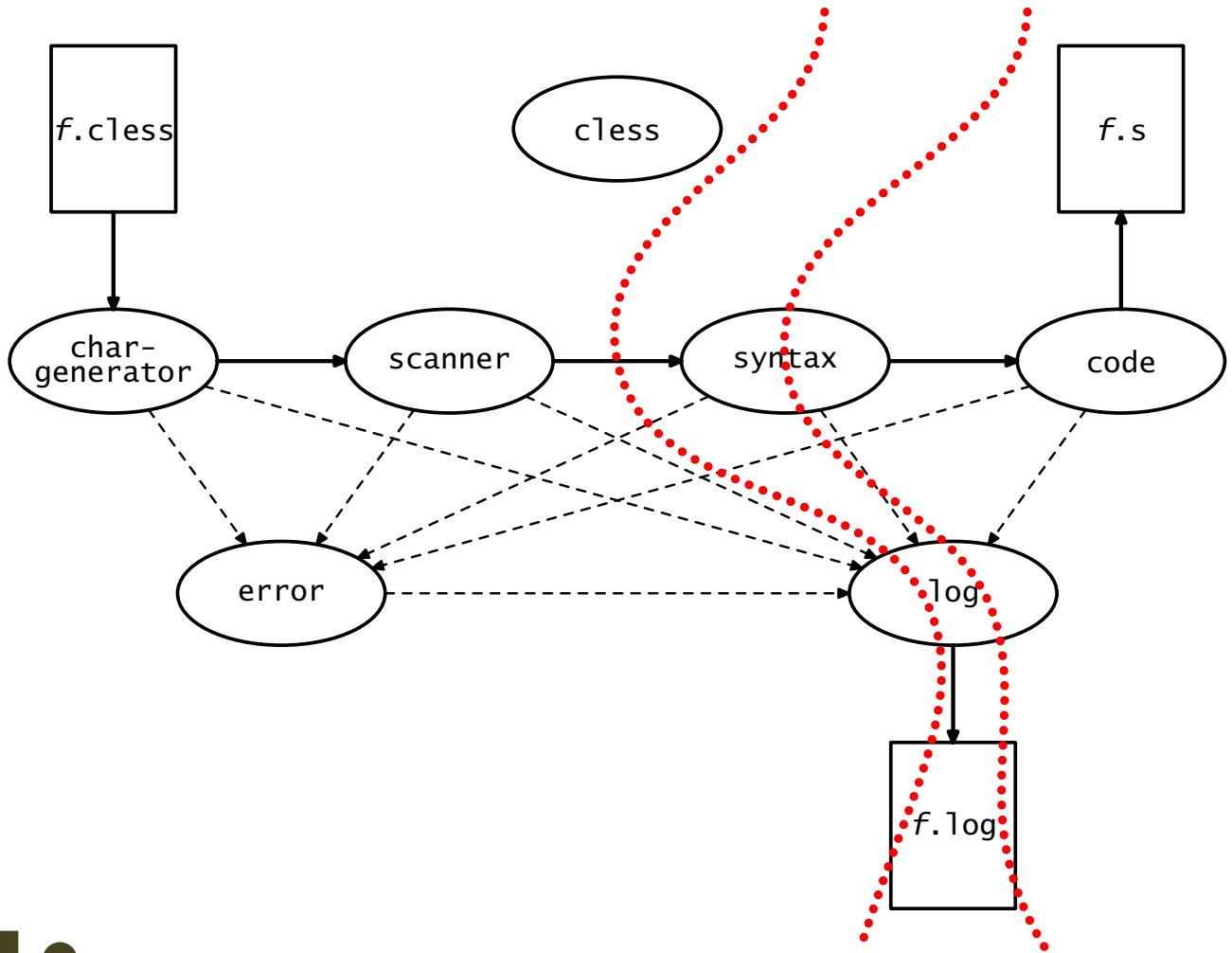
- Hvordan finne deklarasjoner?
- Typesjekking
- Hvordan programmere sjekking av riktig navnebruk?

## Maskinkode

- Litt datamaskinhistorie
- Hva er maskin- og assemblerkode?
- x86-prosessoren
- Programkode og variabler

## Prosjektoversikt

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



# Hva sjekkes ikke?

Noen kompilatorer er «snille» og forteller brukeren at han/hun har skrevet noe som er riktig men dumt.

```
int f (int x)
{
  return x+1;
  x = x+2;
  return x;
}
```

Vår kompilator skal ikke gjøre det. Hvorfor ikke?

- Prosjektet er stort nok allerede.
- Hvor mye dum kode skal man varsle om?

```
int f (int x)
{
  if (x<0) {
    return -1;
  } else if (x>0) {
    return 1;
  }
}
```

```
for (i=1; i<=9; i=i+2) {
  if (i==2) {
    ...
  }
}
```

# Sjekking av deklarasjoner

En kompilator må sjekke riktig navnebruk:

- Alle navn må være deklarert.
- Det må ikke forekomme dobbeltdeklarasjoner.
- Deklarerte navn må brukes riktig (enkel variabel kontra array kontra funksjon).

# Hvordan kople navn og deklarasjon?

```
int a;   int b;

int f ()
{
   int a;   a = b;   exit(0);
}

int c;

int g (int a)
{
   int b;   b = a + c;   f();
}
```

Alle deklarasjoner er objekter av en subklasse av Declaration:

```
abstract class Declaration extends SyntaxUnit {
    String name, assemblerName;
    Type type;
    boolean visible = false;
    Declaration nextDecl = null;
```

Alle deklarasjoner står i en deklarasjonsliste:

```
abstract class DeclList extends SyntaxUnit {
    Declaration firstDecl = null;
    DeclList outerScope;
```

Alle variabler har en peker til sin deklarasjon, og den må vi sette nå. Det skjer i en funksjon check.

```
class Variable extends Operand {
    String varName;
    VarDecl declRef = null;
    Expression index = null;
```

Det samme gjelder for funksjoner representert av et FunctionCall-objekt.

# Lokale variabler

Det enkleste er å sjekke om et navn refererer til en lokal variabel:

```
int f ()
{
   int c;   int v;
   c = 11;  v = c+1;
   return v;
}
```

Metoden check kan få med en parameter som peker til den lokale deklarasjonslisten.

# Globale navn

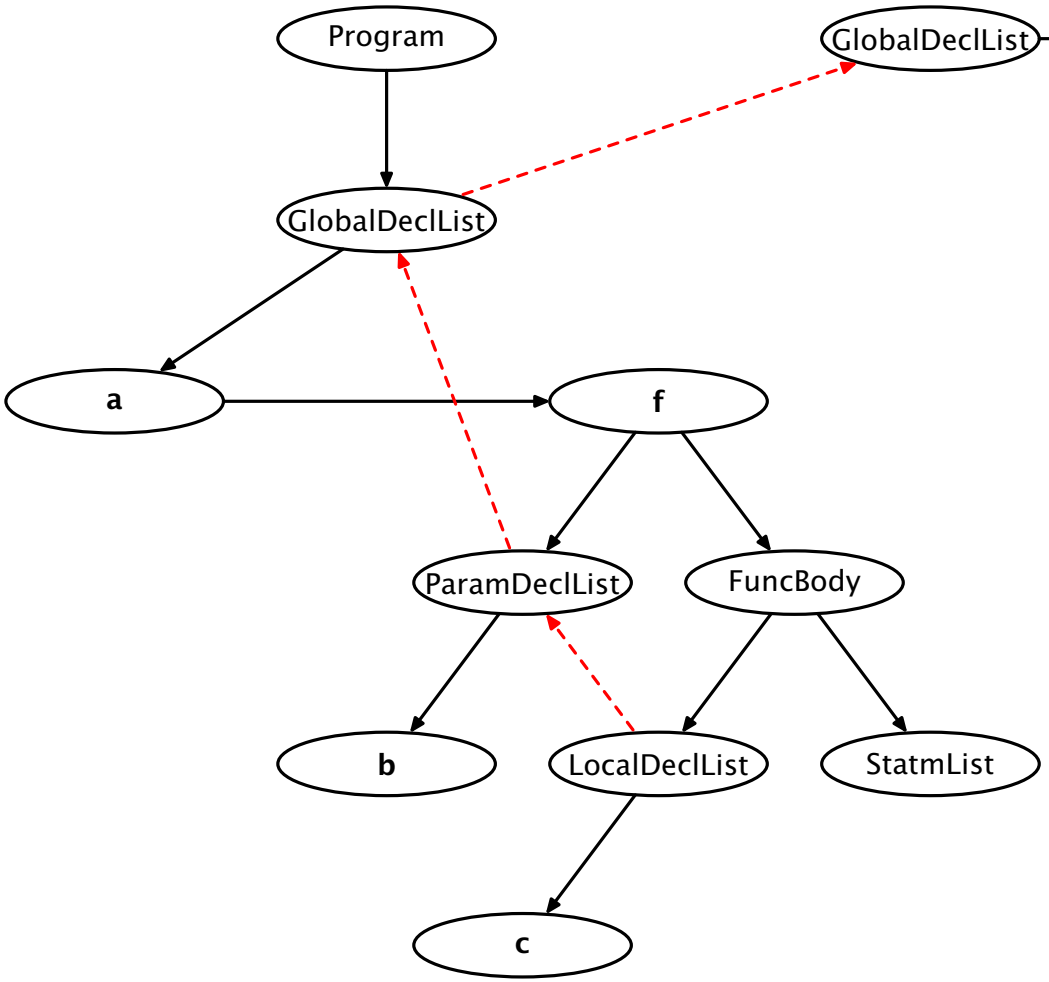
Navn kan imidlertid være deklarert «lenger ute». I C♭ kan vi ha opptil fire nivåer:

```
int a;

int f (int b)
{
   int c;

   ... c ... b ... a ... getchar() ...
```

## Hvordan finner globale navn?

Det enkleste er å la hvert DeclList-objekt ha en peker outerScope til sin nærmeste omliggende deklarasjonsliste.

## Hvordan finne navn?




```
int a;

int f (int b)
{
  int c;

  ... c ... b ... a
  ... getchar() ...
```

# Sjekke riktig bruk

```
abstract class Declaration extends SyntaxUnit {
    String name, assemblerName;
    Type type;
    boolean visible = false;
    Declaration nextDecl = null;

    abstract void checkWhetherArray(SyntaxUnit use);
    abstract void checkWhetherFunction(int nParamsUsed, SyntaxUnit use);
    abstract void checkWhetherSimpleVar(SyntaxUnit use);
}
```

For å sjekke at et navn er riktig brukt, kalles en checkWhether...-metode.

## Eksempel

Vi har funnet uttrykket a[2] som er en ⟨variable⟩. Navnet a må da være deklarert som en GlobalArrayDecl eller LocalArrayDecl. Vi sjekker dette slik i metoden Variable.check:

1. Finn Declaration fDecl; . . . fDecl = . . . ;
2. Kall **fDecl.checkWhetherArray(this);**

   Om a er galt deklarert, skrives det ut en feilmelding og kompileringen stopper.

```
class GlobalArrayDecl extends VarDecl {
        :
    @Override void checkWhetherArray(SyntaxUnit use) {
        /* OK */
    }

    @Override void checkWhetherSimpleVar(SyntaxUnit use) {
        Syntax.error(use,
                     name + " is an array and no simple variable!");
    }
```

checkWhetherFunction er felles for alle variabler:

```
abstract class VarDecl extends Declaration {
        :
    @Override void checkWhetherFunction(int nParams, SyntaxUnit use) {
        Syntax.error(use, name + " is a variable and no function!");
    }
```

# Dobbeltdeklarasjoner

Det er lurt å ha en metode addDecl i DeclList:

```
abstract class DeclList extends SyntaxUnit {
    Declaration firstDecl = null;
    DeclList outerScope;

    void addDecl(Declaration d) {
      :
```

**Hint**

Den enkleste måten å sjekke dobbeltdeklarasjoner på er å legge en test i addDecl.

# Skjuling av navn

Som i de fleste språk vil en indre deklarasjon skjule en ytre.

```
int a;

int f (int a)
{
   int a;    ... a ...
}
```

Dette håndteres automatisk av vårt opplegg.

# Deklarasjonsrekkefølgen

Husk at navn i C♭ først er kjent *etter* at de er deklarert.

```
int f1 ()
{
   outint(v);
}

int v;
    :
```

. . . så dette programmet skal gi feilmelding.

**Hint**

En variabel visible i Declaration gjør det enkelt å vite når en deklarasjon er blitt synlig.

# Testutskrift med –logB

For å sjekke navnebindingen må check fortelle om alle bindinger som gjøres, hvis brukeren angir opsjonen –logB.

- Testutskriftene forteller om check har vært innom alle delene av programmet der navn kan forekomme.
- Testutskriftene forteller om navnebindingen har vært gjort korrekt.

## Opsjonen -B

```
1   int a;   int b;
2
3   int f ()
4   {
5      int a;   a = b;   exit(0);
6   }
7
8   int c;
9
10  int g (int a)
11  {
12     int b;   b = a + c;   f();
13  }
```

```
Binding: Line 5: a refers to declaration in line 5
Binding: Line 5: b refers to declaration in line 1
Binding: Line 5: exit refers to declaration in the libr
Binding: Line 12: b refers to declaration in line 12
Binding: Line 12: a refers to declaration in line 10
Binding: Line 12: c refers to declaration in line 8
Binding: Line 12: f refers to declaration in line 3
```

# Typesjekking

Kompilatoren må også sjekke at typereglene følges.

- Alle deklarasjoner angir typen sin. (Funksjoner angir typen på returverdien.)
- Alle uttrykk og deluttrykk har en type:
  - Tallkonstanter er int.
  - Variabler har typen angitt i deklarasjonen.
  - Funksjonskall har funksjonens type (dvs typen til returverdien).
  - a == b etc har typen int.
  - a + b etc har typen til a og b.

## Hva kan gå galt?

**a==b** a og b har ulik type.

**a+b** a og b har ulik type.

**a[i]** i er ikke int.

**f(a)** a har annen type enn i deklarasjonen av f.

**return a;** a har gal type som returverdi.

## Hva kan ikke gå galt?

**a = b** kan ikke gi typefeil. (Kompilatoren vår vil automatiske generere kode for konvertering.)

# Noen nyttige funksjoner

```
abstract public class Type {
    abstract public int size();
    abstract public String typeName();

    abstract public void checkSameType(int lineNum, Type otherType, String what);
    abstract public void checkType(int lineNum, Type correctType, String what);
    public void genJumpIfZero(String jumpLabel) {}
}
```

---

```
abstract public class BasicType extends Type {
    @Override public void checkSameType(int lineNum, Type otherType, String what) {
        if (this != otherType)
            Error.error(lineNum,
                        what + " should have the same type, not " + typeName() +
                        " and " + otherType.typeName() + ".");
    }

    @Override public void checkType(int lineNum, Type correctType, String what) {
        if (this != correctType)
            Error.error(lineNum,
                        what + " is " + typeName() +
                        ", not " + correctType.typeName() + ".");
    }

}
```

# Hvordan programmere sjekkingen?

Alle programelementer (dvs objekter av en subklasse av SyntaxUnit) har en check-metode med lokal deklarasjonsliste som parameter.

```
class WhileStatm extends Statement {
    Expression test = new Expression();
    StatmList body = new StatmList();

    @Override void check(DeclList curDecls) {
        test.check(curDecls);
        body.check(curDecls);
    }
```

Med disse kan vi traversere hele syntakstreet.

## Hvordan programmere sjekkingen?

- På vei *nedover* i syntakstreet vil check
  - sørge for at alle navn koples til sin deklarasjon
  - sjekke at navnet er brukt riktig (vanlig variabel/array/funksjon)
  - sette visible=true (om aktuelt)
- På vei *oppover* vil den
  - sjekke typer
  - sette riktig type på «seg selv» (om aktuelt)

# Et eksempel til

```
class Variable extends Operand {
    String varName;
    VarDecl declRef = null;
    Expression index = null;

    @Override void check(DeclList curDecls) {
        Declaration d = curDecls.findDecl(varName,this);
        if (index == null) {
            d.checkWhetherSimpleVar(this);
            valType = d.type;
        } else {
            d.checkWhetherArray(this);
            index.check(curDecls);
            index.valType.checkType(lineNum, Types.intType, "Array index");
            valType = ((ArrayType)d.type).elemType;
        }
        declRef = (VarDecl)d;
    }
```

# Datamaskinenes historie

Menneskene har alltid prøvd å lage maskiner for å løse sine problemer.

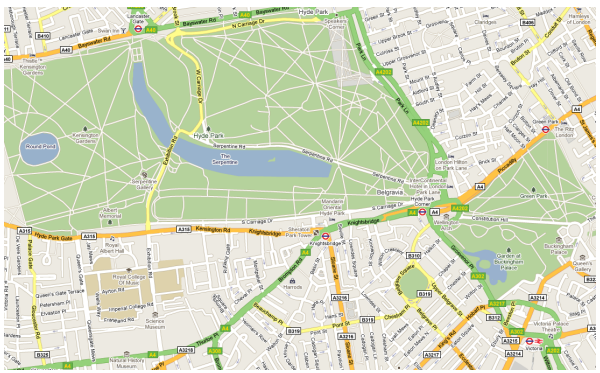
Midt på 1800-tallet var det store problemet *tabeller* med feil.

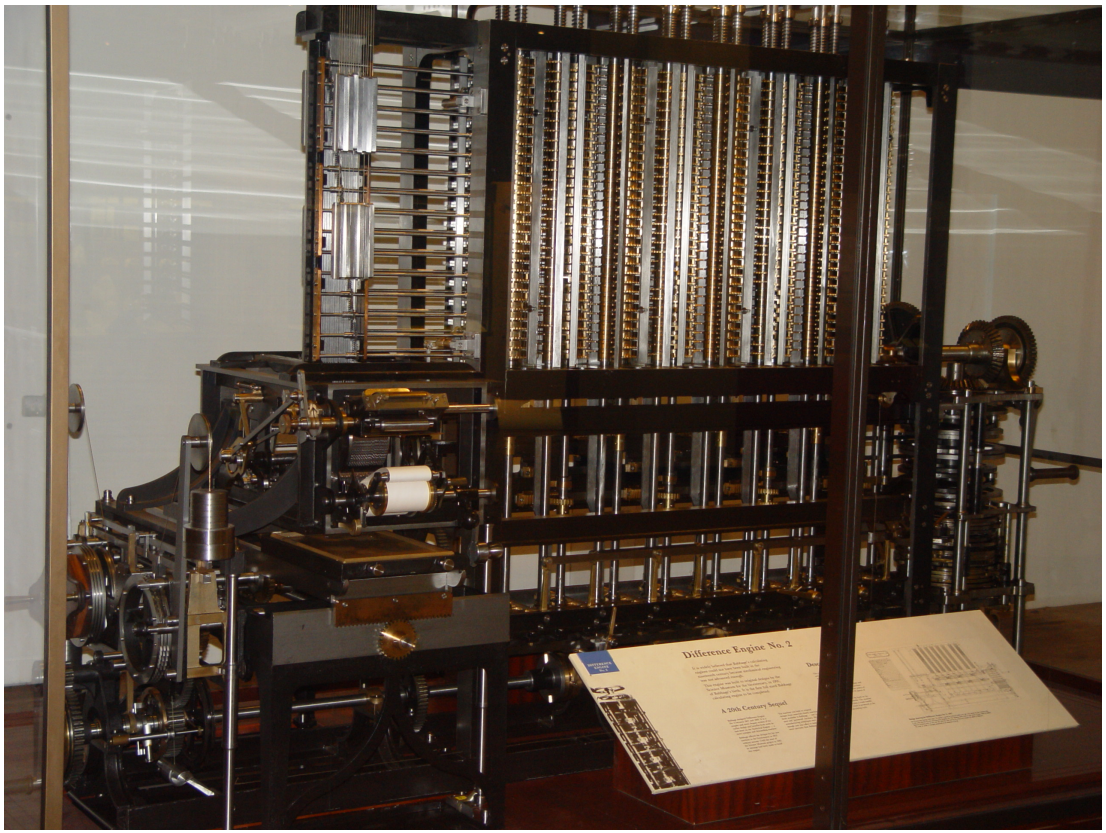
**Charles Babbage** konstruerte på 1830-tallet sin *Difference Engine* som kunne lage tabeller automatisk ved å løse differensligninger. (Den ble først ferdig i 1991.)

Han arbeidet også med en *Analytical Engine* som skulle bli en generell beregningsmaskin.

## Charles Babbage

*The difference engine* på *Science Museum* i London.





# De første moderne datamaskiner

Problemet i 1930-årene var kanoner. Det er mulig å beregne en prosjektilbane, men det er mye arbeid for en matematiker.

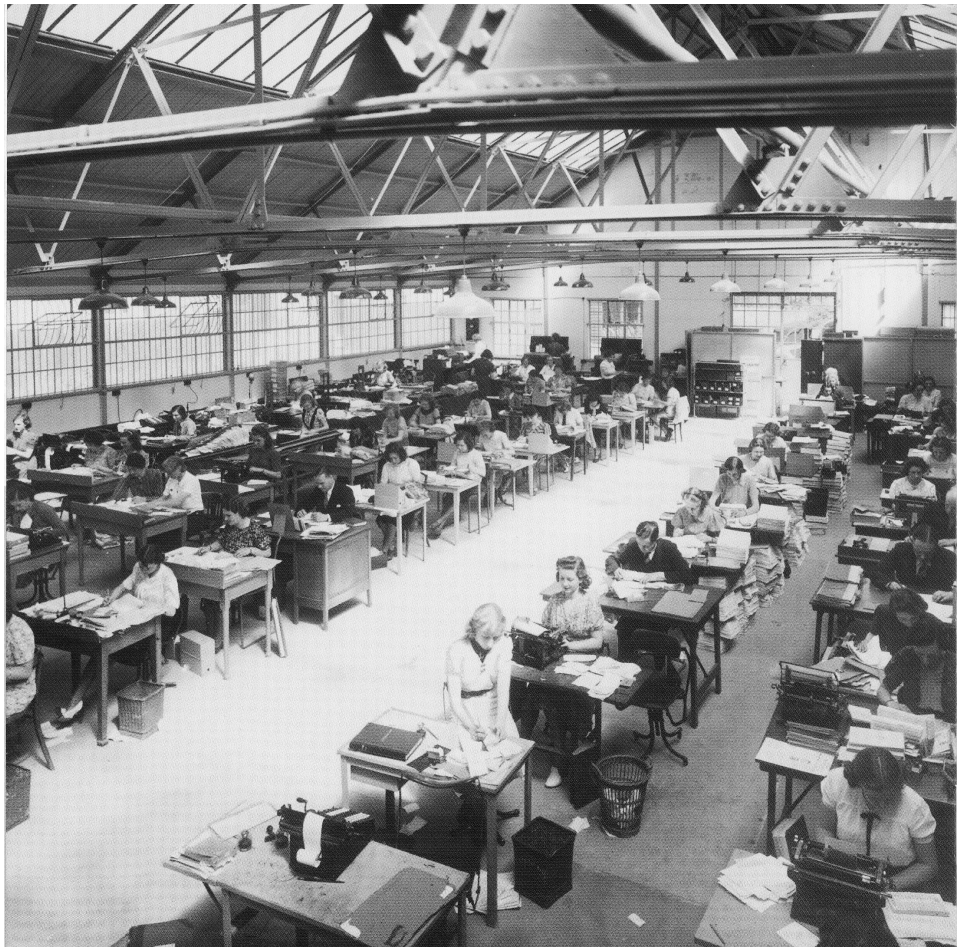
*U.S. Army Ordnance Department Ballistic Research Laboratory* trengte data for dusinvis av nye kanoner.

**Løsning**

Lag *arbeidsbeskrivelse*, og la egne «beregnere» gjøre jobben.

Fra en eldre utgave av *Webster's Dictionary*:

*computer* n, one that computes; *specif:* an automatic electronic machine for performing calculations

## Problem

Hver bane tok opptil 20 timer å beregne (selv med elektrisk bordregnemaskin), og man trengte 2–4000 ulike baner for hver kanon.

## Løsning

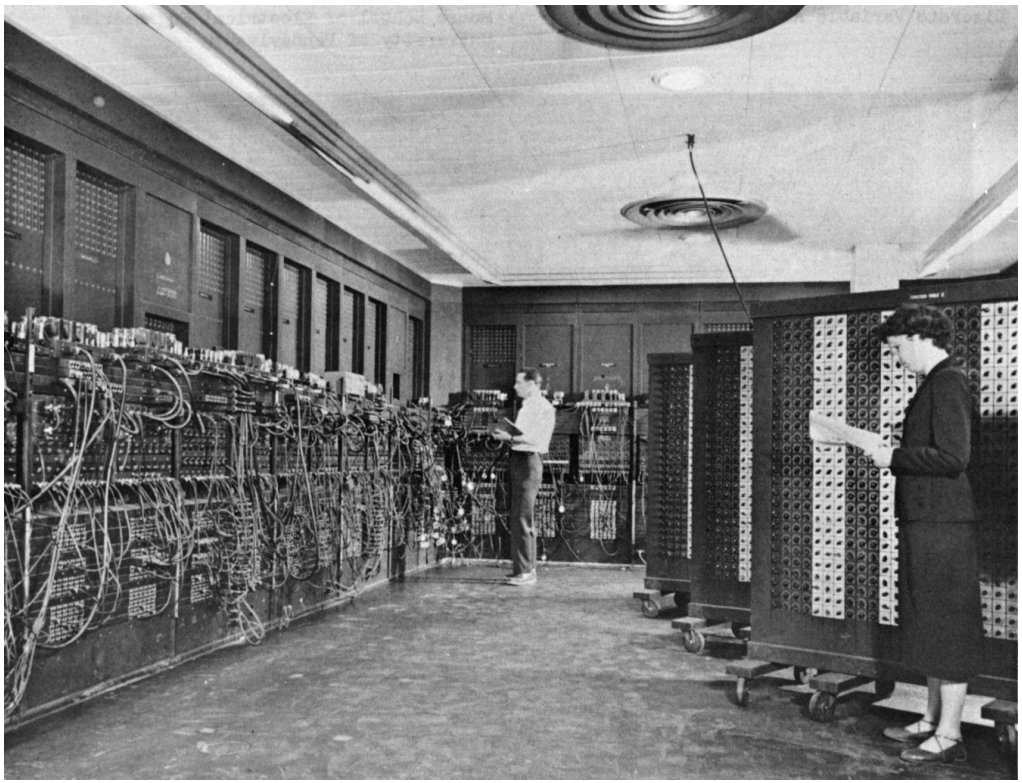
Lag en maskin som gjør dette automatisk.

*Moore School of Electrical Engineering* ved Universitetet i Pennsylvania gjorde det med penger fra *Ballistic Research Laboratory*. Resultatet ble **Eniac** som ble ferdig i 1945. Den kunne beregne en kulebane på drøyt 10 s.

## Eniac

Eniac målte
2½×1×30 m, veide
30 tonn og inneholdt
18 000 radiorør.

Den var i drift til 1955.



# Oppbyggingen av Eniac

Tanken var å kopiere en menneskelig beregner. Den har

**Aritmetisk enhet** («ALU») tilsvarte regnemaskinen med de fire regneartene:

$$+ \quad - \quad \times \quad :$$

Regnemaskinen har et tall for videre beregning; datamaskinen har et **register**.

**Minnet** tilsvarte et ark med mellomresultater. Datamaskinen kunne overføre innholdet av registeret til eller fra en celle i minnet.

**Programmet** tilsvarte beregnerens arbeidsbeskrivelse. Det skulle følges helt slavisk.

# Programmet

Et program for datamaskinen inneholdt de samme elementene som beregnerens arbeidsbeskrivelse:

**Aritmetiske operasjoner** var mulig i de fire regneartene; svaret kom i registeret.

**Mellomlagring av data** skjedde ved at registeret ble kopiert til en angitt celle i minnet.

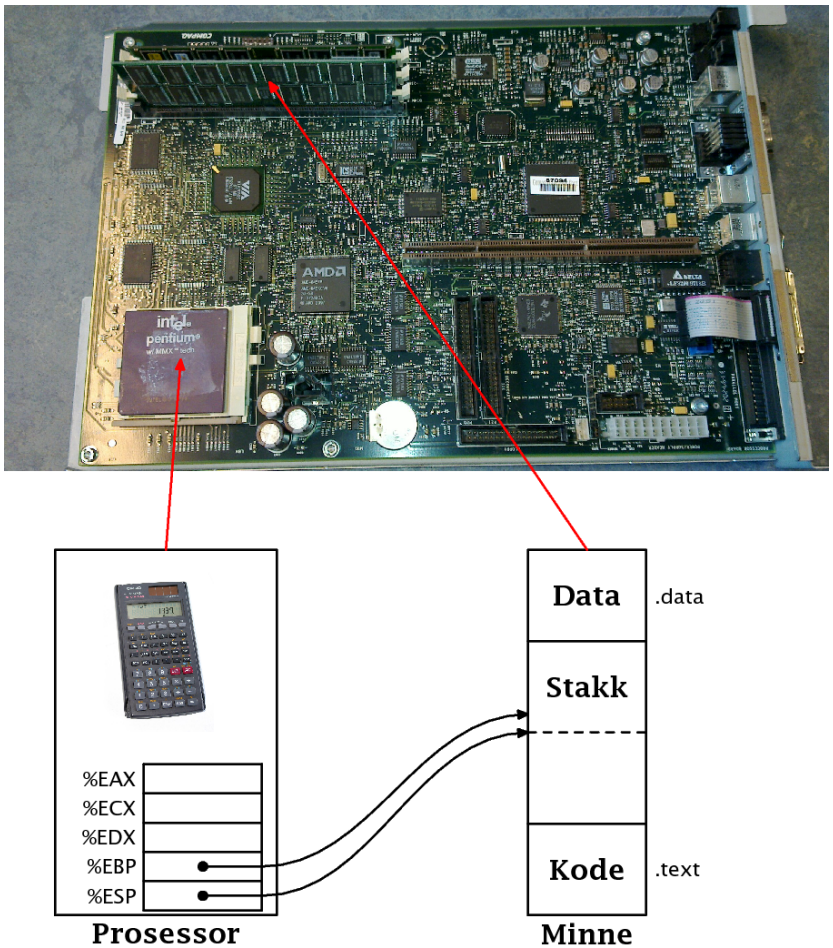
**Hopp** til en angitt instruksjon var nødvendig for å kunne gå i løkker.

**Tester** i forbindelse med hopp var typisk på om registeret var $< 0$, $= 0$ eller $> 0$.

Programmene ble etter hvert kodet som tall (mens Eniac ble kodet med kabler).

## En moderne datamaskin

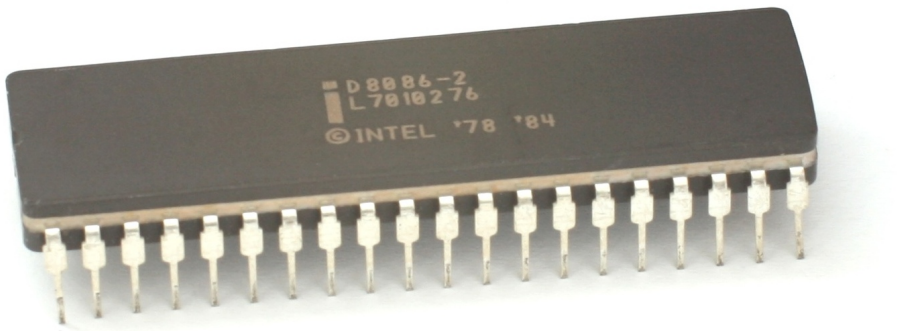
har grovt sett samme oppbygning den dag i dag.



# x86-prosessoren

Denne prosessoren er den mest suksessrike gjennom tidene.

- Introdusert i 1978 med **8086**.
- Produseres fremdeles (som x86-64).
- Brukes i de fleste PC-er.



Minnet inneholder tre former for data:

1. **Data** inneholder globale variabler.
2. **Stakken** inneholder parametre.
3. **Koden** er programmet (dvs instruksjonene i numerisk form).

# Prosessoren inneholder:

1. En regneenhet (kalt ALU = «Arithmetic Logic Unit») som kan
   1. utføre de fire regneartene ($+$, $-$, $\times$ og :)
   2. sammenligne to tall
   3. avgjøre om programmet skal hoppe
   4. flytte tall
2. Registre
   - %EAX, %ECX og %EDX benyttes til beregninger og sammenligninger. (%AL er en del av %EAX.)
   - %EBP («Extended Base Pointer») og %ESP («Extended Stack Pointer») peker på data i minnet.
3. En flyttallsenhet

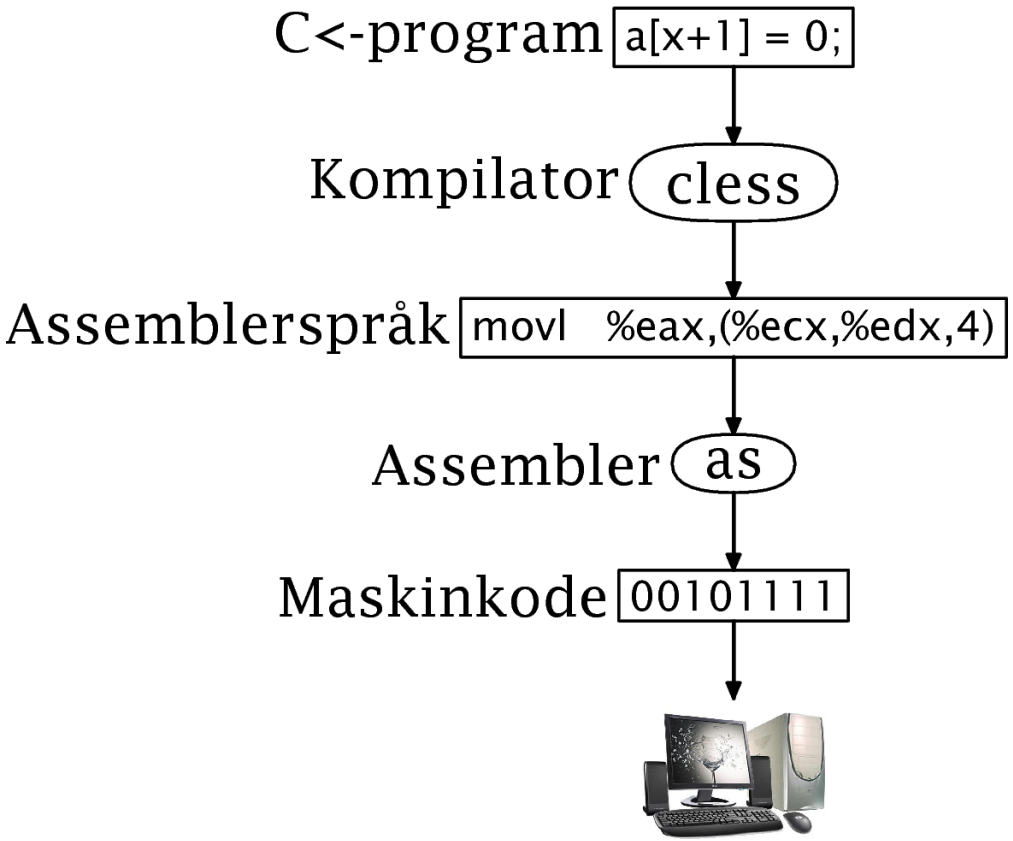
| | | |
|---|---|---|
| `movl` | $\langle v_1 \rangle$,$\langle v_2 \rangle$ | Flytt $\langle v_1 \rangle$ til $\langle v_2 \rangle$. |
| `cdq` | | Omform 32-bits %EAX til 64-bits %EDX:%EAX. |
| `leal` | $\langle v_1 \rangle$,$\langle v_2 \rangle$ | Flytt *adressen* til $\langle v_1 \rangle$ til $\langle v_2 \rangle$. |
| `pushl` | $\langle v \rangle$ | Legg $\langle v \rangle$ på stakken. |
| `popl` | $\langle v \rangle$ | Fjern toppen av stakken og legg verdien i $\langle v \rangle$. |
| `addl` | $\langle v_1 \rangle$,$\langle v_2 \rangle$ | Addér $\langle v_1 \rangle$ til $\langle v_2 \rangle$. |
| `subl` | $\langle v_1 \rangle$,$\langle v_2 \rangle$ | Subtraher $\langle v_1 \rangle$ fra $\langle v_2 \rangle$. |
| `imull` | $\langle v_1 \rangle$,$\langle v_2 \rangle$ | Multipliser $\langle v_1 \rangle$ med $\langle v_2 \rangle$. |
| `idivl` | $\langle v \rangle$ | Del %EDX:%EAX med $\langle v \rangle$; svar i %EAX. |
| `call` | $\langle$`lab`$\rangle$ | Kall funksjonen i $\langle$lab$\rangle$. |
| `ret` | | Returner fra funksjonen. |
| `cmpl` | $\langle v_1 \rangle$,$\langle v_2 \rangle$ | Sammenligning $\langle v_1 \rangle$ og $\langle v_2 \rangle$. |
| `jmp` | $\langle$`lab`$\rangle$ | Hopp til $\langle$lab$\rangle$. |
| `sete` | $\langle v \rangle$ | Sett $\langle v \rangle$=1 om $=$, ellers $\langle v \rangle$=0. |
| `setne` | $\langle v \rangle$ | Sett $\langle v \rangle$=1 om $\neq$, ellers $\langle v \rangle$=0. |
| `setl` | $\langle v \rangle$ | Sett $\langle v \rangle$=1 om $<$, ellers $\langle v \rangle$=0. |
| `setle` | $\langle v \rangle$ | Sett $\langle v \rangle$=1 om $\leq$, ellers $\langle v \rangle$=0. |
| `setg` | $\langle v \rangle$ | Sett $\langle v \rangle$=1 om $>$, ellers $\langle v \rangle$=0. |
| `setge` | $\langle v \rangle$ | Sett $\langle v \rangle$=1 om $\geq$, ellers $\langle v \rangle$=0. |

## Formålet

Vårt oppdrag er å lage en kompilator:

**Inndata** er trerepresentasjonen av C♭-programmet laget i del 1.

**Utdata** er en fil med x86 assemblerkode.



## Maskinkode

Dette er den *numeriske representasjonen* av instruksjonene; vi skal ikke benytte den i dette kurset.

## Assemblerkode

Dette er den *tekstlige representasjonen* av instruksjonene.

```
func:   movl   $0,%eax   # Initier til 0.
```

Navnelapp | Instruksjon | Parametre | Kommentar

## Noen eksempler

```
        .text
Start:  movl    $17,%eax    # Legg verdien 17 i %EAX
        movl    $-2,%edx    # Legg verdien -2 i %EDX
        addl    %edx,%eax   # Addér %EDX til %EAX
        jmp     Start       # Hopp til Start
```

**NB!** Direktivet **.text** forteller at vi skal legge instruksjonene i kodelageret. (Dette er standard.)

```
int v;

int main ()
{
    v = 65;
    putchar(v+1);
    putchar(10);
}
```

```
        .data
.tmp:   .fill   4                       # Temporary storage
        .globl  v
v:      .fill   4                       # int v;
        .text
        .globl  main
main:   pushl   %ebp                    # Start function main
        movl    %esp,%ebp
        movl    $65,%eax                # 65
        movl    %eax,v                  # v =
        movl    v,%eax                  # v
        pushl   %eax
        movl    $1,%eax                 # 1
        movl    %eax,%ecx
        popl    %eax
        addl    %ecx,%eax               # Compute +
        pushl   %eax                    # Push parameter #1
        call    putchar                 # Call putchar
        addl    $4,%esp                 # Remove parameters
        movl    $10,%eax                # 10
        pushl   %eax                    # Push parameter #1
        call    putchar                 # Call putchar
        addl    $4,%esp                 # Remove parameters
.exit$main:
        movl    %ebp,%esp
        popl    %ebp
        ret                             # End function main
```

# Modulen code

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


import java.io.*;
import no.uio.ifi.cflat.cflat.Cflat;
import no.uio.ifi.cflat.error.Error;
import no.uio.ifi.cflat.log.Log;


public class Code {
    private static PrintWriter codeFile;
    private static boolean generatingData = false;

    public static final String tmpLabel = ".tmp";
```

## Modulen 'code'

```
public static void init() {
    String codeFileName;

    if (Cflat.sourceBaseName == null) return;
    codeFileName = Cflat.sourceBaseName + ".s";
    try {
        codeFile = new PrintWriter(codeFileName);
    } catch (FileNotFoundException e) {
        Error.error("Cannot create code file " + codeFileName + "!");
    }

    genVar(tmpLabel, false, 4, "Temporary storage");
}

public static void finish() {
    codeFile.close();
}
```

## Modulen 'code'

```
private static void printLabel(String lab, boolean justALabel) {
    if (lab.length() > 6) {
        codeFile.print(lab + ":");
        if (! justALabel) codeFile.print("\n        ");
    } else if (lab.length() > 0) {
        codeFile.printf("%-8s", lab+":");
    } else {
        codeFile.print("        ");
    }
}

public static void genInstr(String lab, String instr,
                            String arg, String comment) {
    if (generatingData) {
        codeFile.println("        .text");
        generatingData = false;
    }

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

# Oppsummering

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!