Kompilatorkonstruktion
(CTH/GU)
vt 2001

Thomas Hallgren (kursansvarig)

Josef Svenningsson (handledare)

Dagens föreläsning

Snabb genomgång av kursen för att ge en överblick. Det betyder att:

- mycket nytt (förhoppningsvis) kommer presenteras,
- ni förväntas inte komma ihåg allt som sägs.

Försök istället få en överblick av ämnet.

Vad är en kompilator?

Med en “kompilator” brukar man vanligtvis mena ett program som översätter från ett högnivåspråk (C/Ada/Haskell/etc) till maskinkod för en viss processor (pentium/sparc/mips/etc).

Men det finns långt fler “snarlika” problem/program där man

- analyserar struktureren på indata,
- översätter till något

t ex

- textformattering “markup-languages”
- hårdvarubeskrivningspråk

Vad gör en kompilator?

Läser in ett program, vanligtvis som text.

Kontrollerar om programmet är korrekt.
(Analys)

Funderar lite.

Skriver ut maskinkod.
(Syntes)

Programspråk

Hur beskriver man programspråk?

Syntax

Utseende

Semantik

- Statisk Semantik: egentligen syntax
- Dynamisk Semantik: betydelse

```
j=0;
for i in 0 to 10-j do

        j=j+1;

end for
```

Syntax

Hur beskriver man syntax?

Lexikal struktur

Vilka “ord” språket är uppbyggt av.

Syntaktisk struktur

Hierarkisk struktur

Exempel: lexikal struktur

Nyckelord

while return if then else

Symboler

; = ( ) { }

Identifierare

i j k person1 person2 resultat

Går inte räkna upp, istället beskrivs de:

Identifierare börjar med en bokstav och följs av en eller flera bokstäver eller siffror.

Konstanter

0 123 1.35 true false

Exempel: Ett litet program 1

```
int sumto (int n)
{
        int i,sum;
        i = 0;
        sum = 0;
        while ( i < n ) {
                i = i + 1;
                sum = sum + i;
        }
        return sum;
}
```

Lexikalanalys

Det första som görs är att stycka upp texten.

```
int ID sumto ( int ID n )
{ int ID i , ID sum ;
  ID i = INT 0 ; ID sum = INT 0 ;
  while ( ID i < ID n ) {
    ID i = ID i + INT 1 ;
    ID sum = ID sum + ID i ;
  }
  return ID sum ;
}
```

Lexikalanalys, del 2

Vissa "ord" har värden, t ex ID *i* *INT 1*

Andra saknar värden, t ex ***int*** = ;

"Orden" kallas vanligen **tokens** (ibland **terminaler**), och:

- beskrivs mha av reguljära uttryck, t ex Identifierare = [a-zA-Z][a-zA-Z0-9]*,
- känns igen med "Deterministic Finite Automata", DFA.

Syntaxkontroll

Nästa steg är att kontrollera om programmets utseende är korrekt, dvs om det stämmer överens med syntaxen för programspråket.

Ett programsspråks syntax anges lämpligen med en grammatik. Kompilatorn kontrollerar om det givna programmet uppfyller gramatiken.

sats ::= ***while (*** *uttryck* ***)*** *sats*
| *variable* ***=*** *uttryck* ***;***
| ***return*** *uttryck* ***;***
| ***{*** *satser* ***}***
...

Denna kontroll görs normalt inte som en separat del av kompilatorn utan tillsammans med efterföljande steg, parsning.

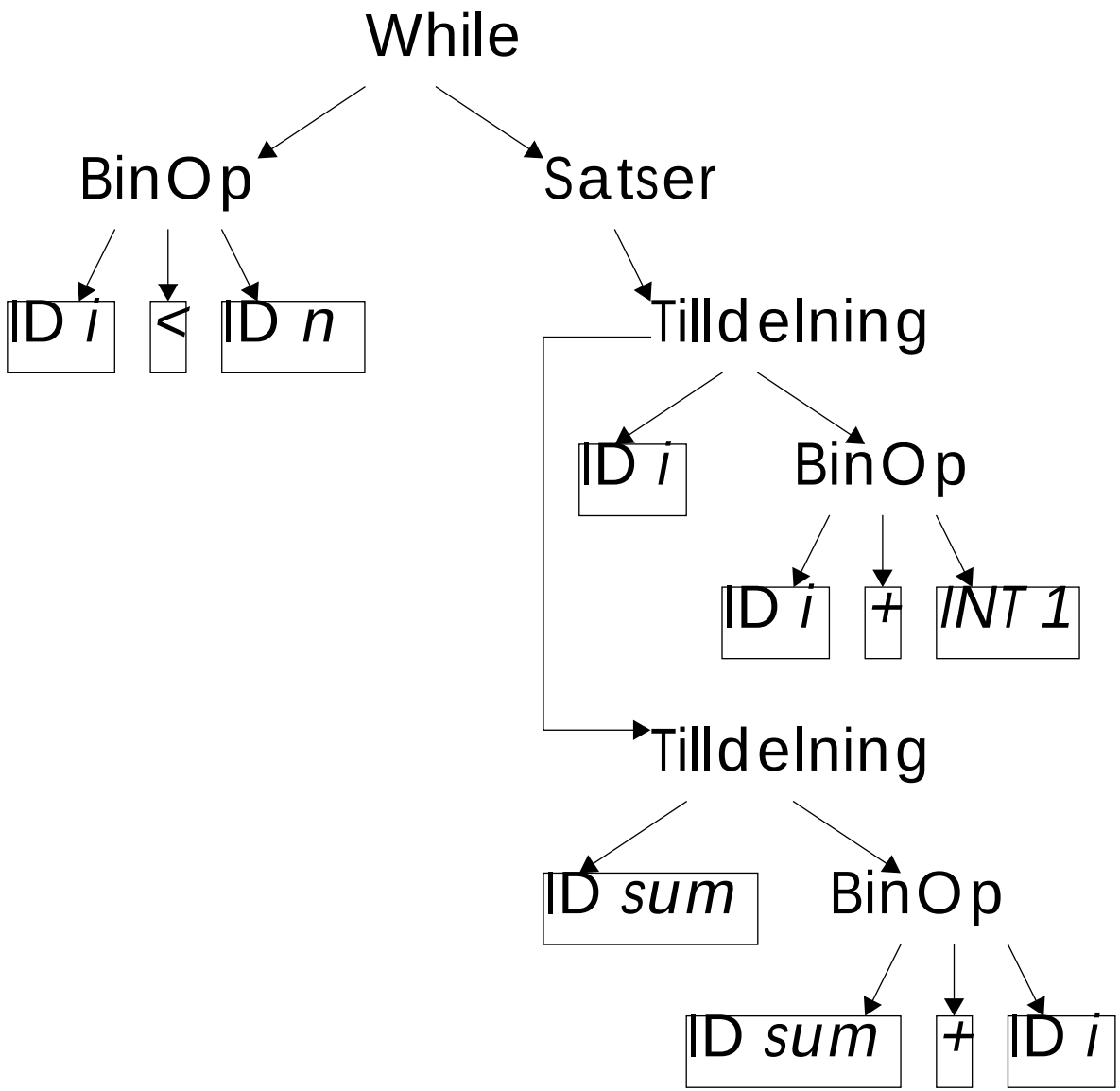
Parsning

Förutom att kontrollera syntaxen vill man oftast ändra representationen av programmet från en sekvens av tokens till ett **abstrakt syntaxträd**.

Sats = While Uttryck Sats
| Tilldelning Variable Uttryck
| Return Uttyck
| Satser [Sats]
...

Uttryck = BinOp Uttryck Op Uttryck
...

Parsning


While
BinOp
Satser
ID *i*
<
ID *n*
Tilldelning
ID *i*
BinOp
ID *i*
+
INT 1
Tilldelning
ID *sum*
BinOp
ID *sum*
+
ID *i*

Parsning, del 2

Parsen använder grammatiken för språket när den bygger syntaxträdet.

Syntaxkontrollen blir delvis automatisk då syntaxfel i många fall leder till att det inte går att bygga ett abstrakt syntaxträd.

Språkets grammatik beskrivs ofta på Backus-Naur Form (BNF). Det ger en kontext-fri grammatik, något som inte räcker för vanliga programspråk. Resultatet blir att en del felaktiga program slinker igenom parsern:-(

Statisk Semantik

Ett vanligt fel som inte kan upptäckas med en kontext-fri grammatik är att använda odefinierade variabler eller variabler av fel typ.

Båda dessa fel är brott mot den “statiska semantiken” för programspråket.

Statisk semantik beskrivs mha:

- Naturligt språk
- kraftfullare grammatiker
- formella regler

$$\frac{e1{:}int \quad e2{:}int}{e1+e2{:}int} \qquad \frac{e1{:}int \quad e2{:}int}{e1<e2{:}boolean}$$

Typkontroll

Ett sätt att upptäcka brott mot den statiska semantiken är att typkontrollera programmet. En metod att göra det är att bygga en symboltabell som givet namnet på en identifierare returnerar dess typ.

För vårt exempel får vi:

sumto :: int -> int
n :: int
i :: int
sum :: int

Även symbolerna i programmet har typer (de kan betraktas som funtioner):

< :: (int,int) -> boolean
+:: (int,int) -> int

Typkontroll, del 2

Om vi sätter ut typer i while-satsen får vi

```
while(((i::int)<(n::int))::boolean) {
        i::int =
                (i::int +1::int)::int;
        sum::int =
                (sum::int + i::int)::int;
        }
```

Vissa språk tillåter att två funktioner kan ha samma namn, om deras typer inte är identiska.

```
<:: (int,int) -> boolean
<:: (double,double) -> boolean
```

Kodgenerering

Efter att kompilatorn övertygat sig själv om att programmet är korrekt går det att generera kod.
Vad koden skall göra bestäms av den **Dynamiska Semantiken**, som ges av:

- Naturligt språk

- Översättningssemantik
  *e1 && e2 => **if** e1 **then** e2 **else** false*

- Denotationssemantik
  Matematiska funktioner

- Operationelsemantik

$$\frac{\langle b,w\rangle \Rightarrow true \qquad \langle cs,w\rangle \Rightarrow w'}{\langle \text{if } b \text{ then } cs,w\rangle \Rightarrow w'}$$

Kodgenerering

Det gäller att generera kod:

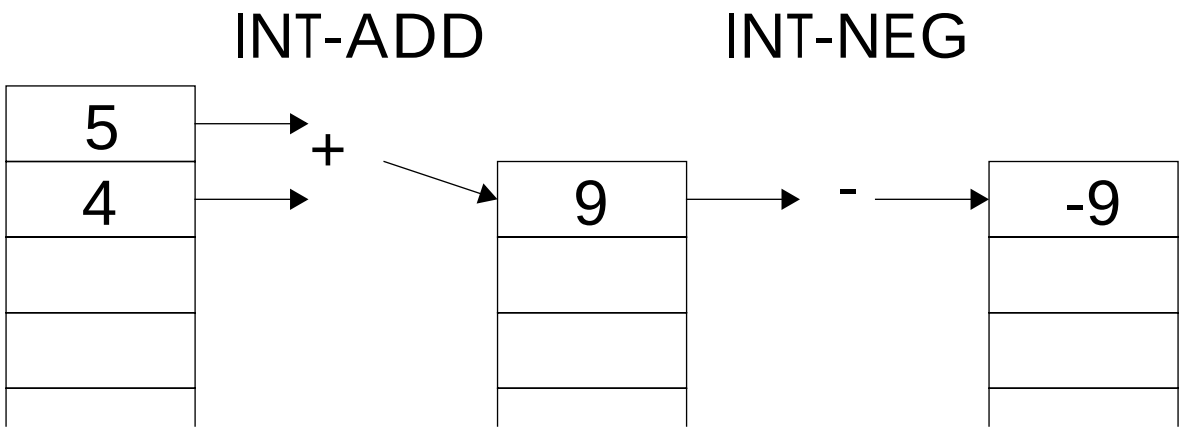
- som är korrekt
- effektiv

Kodens utseende beror på vilken sorts instruktioner målprocessor har, två vanliga sorter är stack- och 3-operands instruktioner.

Det är enklast med stackinstruktioner så låt oss börja med dem.

Stackmaskin

Alla operander ligger på stacken, alla resultat likaså.

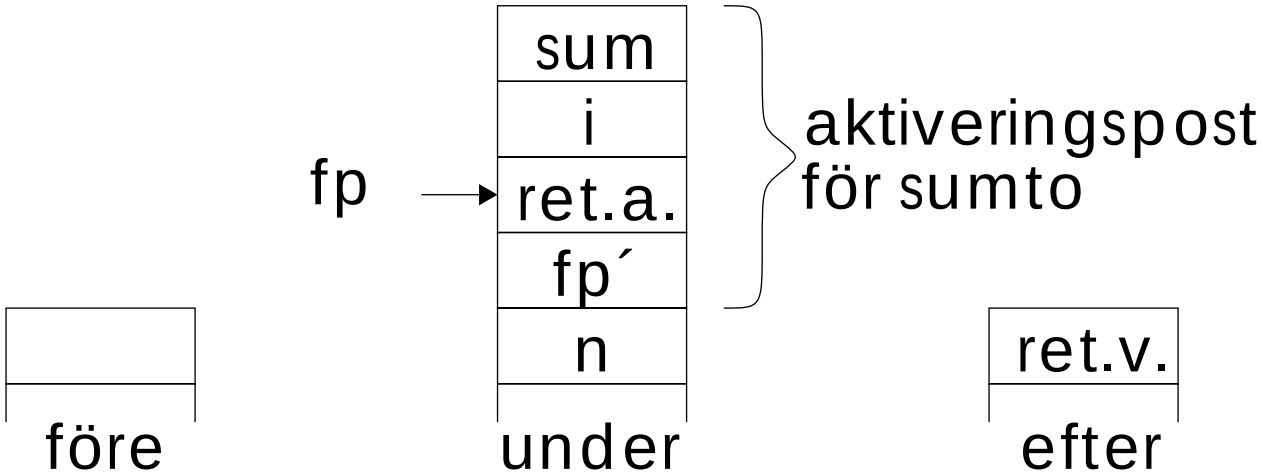


+ Enkelt att generera kod från syntaxträd.

- Svårt att optimera koden.
- “Riktiga” processorer är inte stackmaskiner

Stackmaskin, funktionsanrop

Även lokala variabler och argument ligger på stacken.



anropet till sumto.

De två “extra” orden på stacken är retur-adressen, samt det gamla värdet på “framepointern”.

Efter anropet ligger värdet av uttrycket i return-satsen kvar.

3-operandsmaskin

Operander och resultat är i register.
Processorns register motsvarar, på ett ungefär, variabler i programkoden.

INT_ ADD	r3,r1,r2	r3 = r1 + r2
INT_NEG	r4,r3	r4 = - r3

+ Lätt att dela på uttryck.
+ Enkelt att ändra ordning på instruktioner.
+ “Riktiga” processorer är 3-operandsmaskiner (eller åtminstone registerbaserade)

- Registren räcker inte till.
- Register fungerar inte bra ihop med rekursion.

3-operandsmaskin, del 2

För att lösa problemet med att registerna inte räcker till för alla variabler lägger man några variabler i minnet.

```
INT_ LOAD    r1,a
INT_ LOAD    r2,b
INT_ ADD     r3,r1,r2        r3 = r1 + r2
INT_NEG      r4,r3           r4 = - r3
INT_ STORE   c,r4
```

Tricket är att hålla de värden man räknar på i register så mycket som möjligt.

3-operandsmaskin, funktionsanrop

Argument som skickas till en funktion skall vanligtvis användas snart, likaså returvärdet. **Anropskonventionen** för registerbaserade processorer använder därför register för att skicka argument och returvärden.

r0 - r3 innehåller de fyra första argumenten.
r0 innehåller returvärdet.

Vid färre än fyra argument är innehållet i de oanvända registerna odefinierat.
Vid fler än fyra argument läggs de överskjutande argumenten på stacken.

Det blir krångligare på processorer som har flera sorts register, t ex för flyttal och heltal, något nästan alla har.

“Sumto” för Sparc

Sparc skickar argument i register %o0 till %o7 och resultat i %o0 (, gäller endast i lövrutiner). Instruktionen efter ett hopp utförs alltid.

```
_ sumto:                        ;Register %o0 är n.
        mov     %o0,%g3         ;Spara n i %g3.
        mov     0,%o0           ;Register %o0 är nu sum.
        cmp     %o0,%g3         ;Om 0>n ...
        bg      L3              ; ... hoppa till L3
        mov     0,%g2           ;,men först i=0.
        add     %o0,%g2,%o0;sum = sum + i.
L6:     add     %g2,1,%g2 ;i = i + 1.
        cmp     %g2,%g3         ;Om i<=n ...
        ble,a   L6              ;... hoppa till L6
        add     %o0,%g2,%o0
                                ; ,men först sum = sum + i.
L3:     retl                    ;Klart ...
        nop                     ;,men först gör ingenting!
```