Forelesning 1

Introduksjon

Joakim Bjørk

Universitetet i Sørøst-Norge

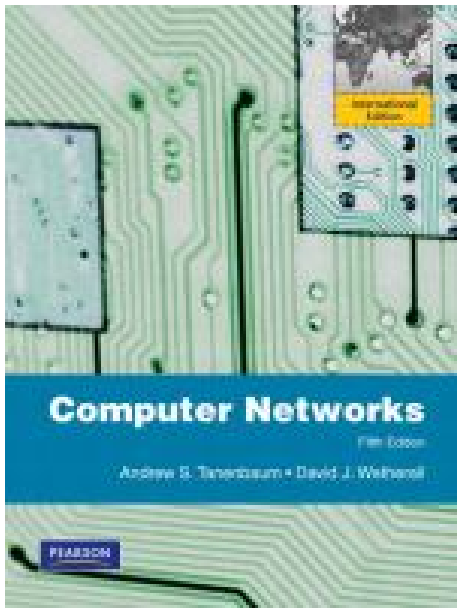
15. august 2018

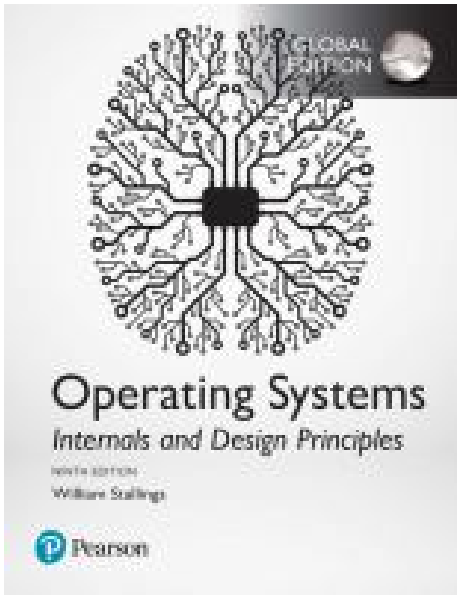
Hva skal vi lære

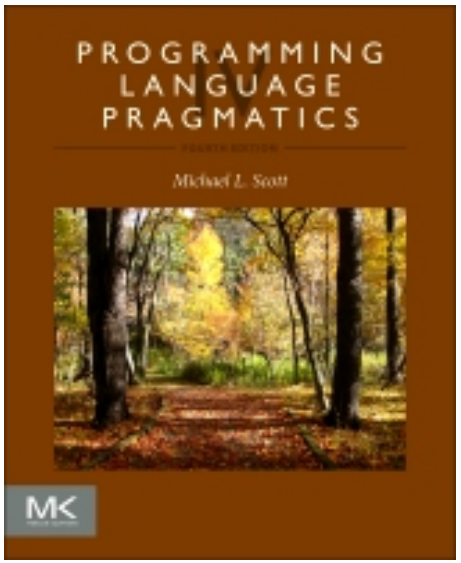
- I dette faget skal vi lære mer om hvordan:
  - opreativsystemer virker
  - maskiner kan kommunisere i nettverk
  - hvordan et programmeringsspåk fungerer

Bøkene

Vi skal bruke disse tre bøkene, men alt er ikke pensum.







Hvorfor har vi programmeringsspråk

- En datamaskin er stokk tett så vi må kunne fortelle den hva den skal gjøre
- Den har et sett med instruksjoner den forstår.
  - Fetch
  - Add
  - Cmp
  - +++++
- Disse er representert ved en hexadesimal kode og vi kan bruke det for å fortelle maskinen hva den skal gjøre

GCD maskinkode

Om vi ønsker å lage et lite program som regner ut minste felles multiplum for to heltall med instruksjonene for en x86 maskin vil de kunne se slik ut

GCD i maskinkode

```
55 89 e5 55   83 ec 04 83   e4 f0 e8 31   00 00 00 89   c3 e8 2a 00
00 00 39 c3   74 10 8d b6   00 00 00 00   39 c3 7e 13   29 c3 39 c3
75 f6 89 1c   24 e8 6e 00   00 00 8b 5d   fc c9 c3 23   d8 eb eb 90
```

Når man begynte å skrive større programmer ble det uholdbart å skulle skrive maskinkode

Assembler

- Assembler språk ble oppfunnet for å få mer meningsfulle kodeord til instruksjonene

GCD i asemblerkode

```
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $4, %esp
    andl    $-16, %esp
    call    getint
    movl    %eax, %ebx
    call    getint
    cmpl    %eax, %ebx
    je      C
A:  cmpl    %eax, %ebx
    jle     D
    subl    %eax, %ebx
B:  cmpl    %eax, %ebx
    jne     A
C:  movl    %ebx, (%esp)
    call    putint
    movl    -4(%ebp), %ebx
    leave
    ret
D:  subl    %ebx, %eax
    jmp     B
```

Assebler

- Til å begynne med var det en en til en relasjon mellom kodeord og instruksjoner
- Det å oversette fra assemblerkode til maskinkode var jobben til et lite program som kalles assembler
- Ulike maskiner med ulikt instruksjonssett hadde egne assemblerspråk
- Man begynte å ønske seg språk som var:
  - maskinuavhengige
  - mer likt matematikk
- Midt på 50-tallet kom Fortran som et svar på dette
- Lisp og Algol kom ikke lenge etter

Hvorfor så mange språk

I dag er det tusenvis av høynivåspråk. Hvorfor er det så mange?

- Evolusjon - Dette er et nytt fagfelt og man finner stadig bedre måter å gjøre ting på
- Spesialiseringer - Noen språk er laget for spesielle oppgaver
- Personlige preferanser - Vi er forskjellige og liker ulike måter å gjøre ting på

Ulike typer programmeringsspråk

I boka er høynivåspråk delt inn i to hovedgrupper med tre undergrupper hver.

- Deklarative - Fokus på hva maskinen skal gjøre
  - funksjonelle - Basert på rekursive funksjonsdefinisjoner inspirert av λ-calculus
  - dataflow - Baserer seg på tokens som flyttes mellom noder
  - logiske - Basert på predikatlogikk.
- Imerative - fokus på hvordan maskinen skal utføre en oppgave.
  - von Neumann - språk hvor man gir verdier til variable (vanlige programmeringsspråk)
  - objektorienterte - Utvidelse av von Neuman. Mer struktur.
  - skripting - Ment som lim mellom andre programmer og for rask prototyping

Ulike typer programmeringsspråk

declarative	
functional	Lisp/Scheme, ML, Haskell
dataflow	Id, Val
logic, constraint-based	Prolog, spreadsheets, SQL
imperative	
von Neumann	C, Ada, Fortran, ...
object-oriented	Smalltalk, Eiffel, Java, ...
scripting	Perl, Python, PHP, ...

gcd eksempelet i C - et von Neumann språk

Hvordan tenker en C-programmerer om GCD

For å regne ut gcd av a og b så sjekker man om de er like.
Om de er like så skriver man ut en av dem og avslutter.
Er de ikke like så erstatter man den største med differansen mellom a og b og gjentar prosessen

```
int gcd(int a, int b){
  while(a != b){
    if(a > b) a = a - b;
    else b = b - a;
  }
  return a;
}
```

gcd eksempelet i OCaml - et funksjonelt språk

Hvordan tenker en COCaml-programmerer om GCD

gcd av a og b er definert som:
1) a når a og b er like
2) gcd av b og a - b når a $>$ b
3) gcd av a og b - a når a $<$ b

```
let rec gcd a b =
  if a = b then a
  else if a > b then gcd b (a-b)
    else gcd a (b-a)
```

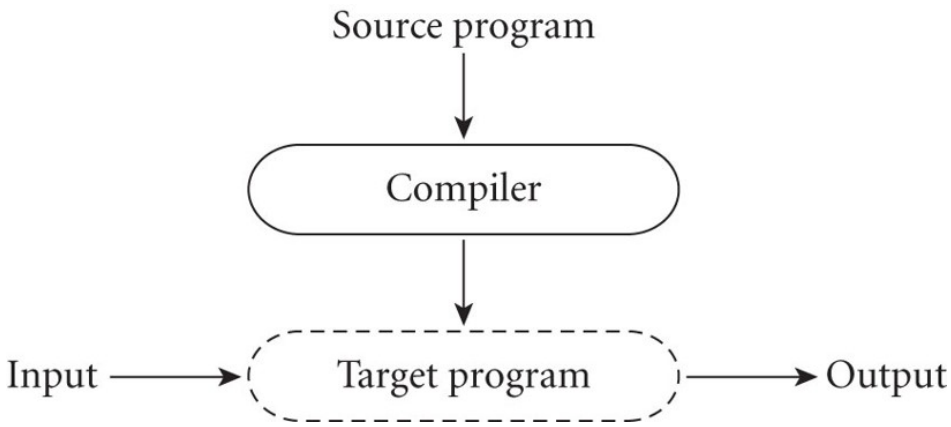
gcd eksempelet i Prolog - et logisk språk

Hvordan tenker en Prolog-programmerer om GCD

utsagnet gcd(a,b,c) er sant hvis:
1) a, b og g er like
2) a er større enn b og det eksisterer et tall c slik at c er a - b og gcd(c,b,g) er sant, eller
3) a er mindre enn b og det eksisterer et tall c slik at c er b - a og gcd(a,c,g) er sant

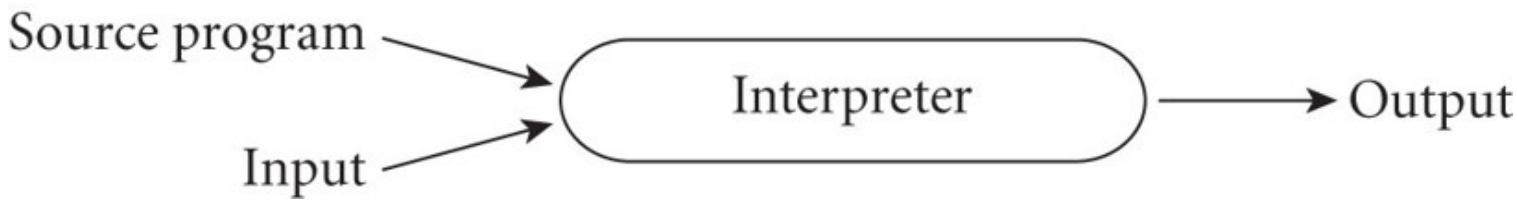
```
gcd(A,B,G) :- A = B, A = G.
gcd(A,B,G) :- A > B, C is A-B, gcd(C,B,G).
gcd(A,B,G) :- A < B, C is B-A, gcd(A,C,G).
```

Pure compilation



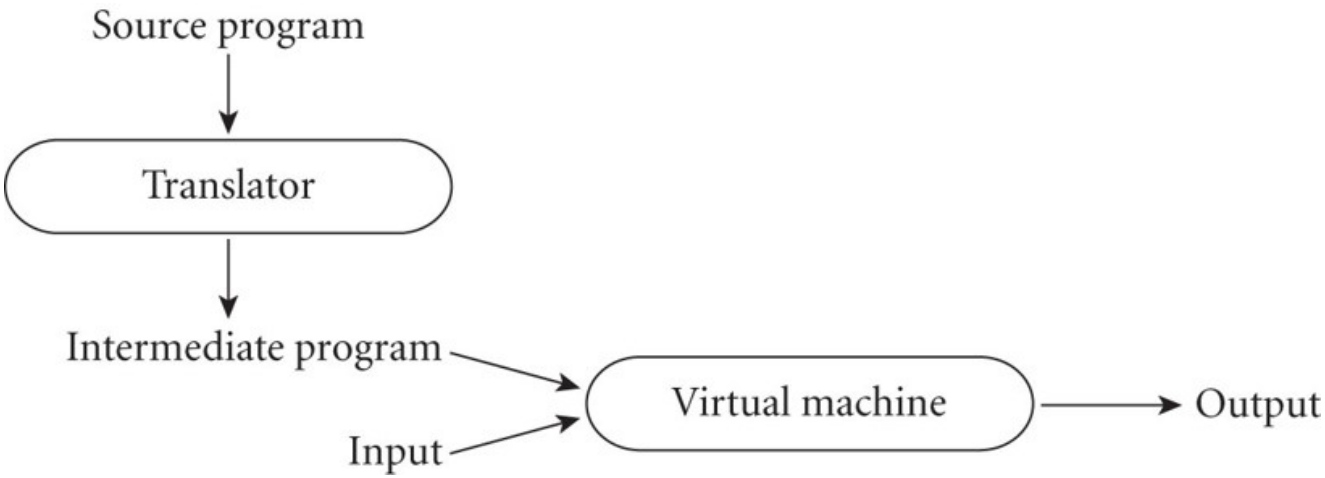
- Kompilatoren oversetter et program skrevet i et programmeringsspråk til et program i maskinkode
- Brukeren kan siden be operativsystemet om å kjøre programmet
- Kompilatoren er et eget program i maskinkode

Pure interpretation



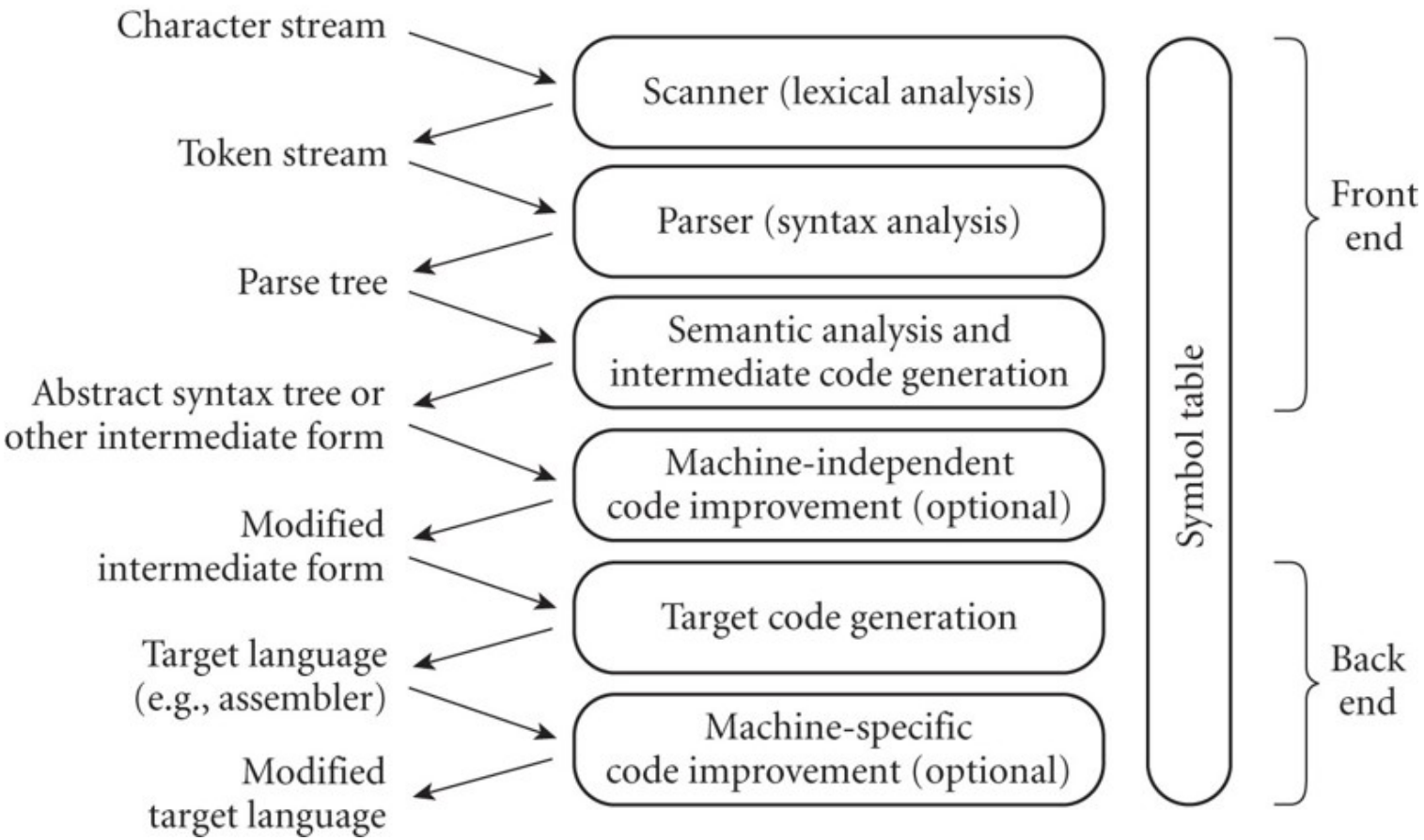
- Til forskjell fra kompilatoren vil en interpreter være aktiv gjennom hele kjøringen av programmet
- interpreteren vil være som en virtuel maskin som har høynivåspråket som maskinspråk
- iterpreteren vil lese et og et statement og eksekvere dem
- en interpreter vil kunne være bedre til debugging enn en kompilator, men vil ikke kunne gi samme hastighet

Hybridløsning

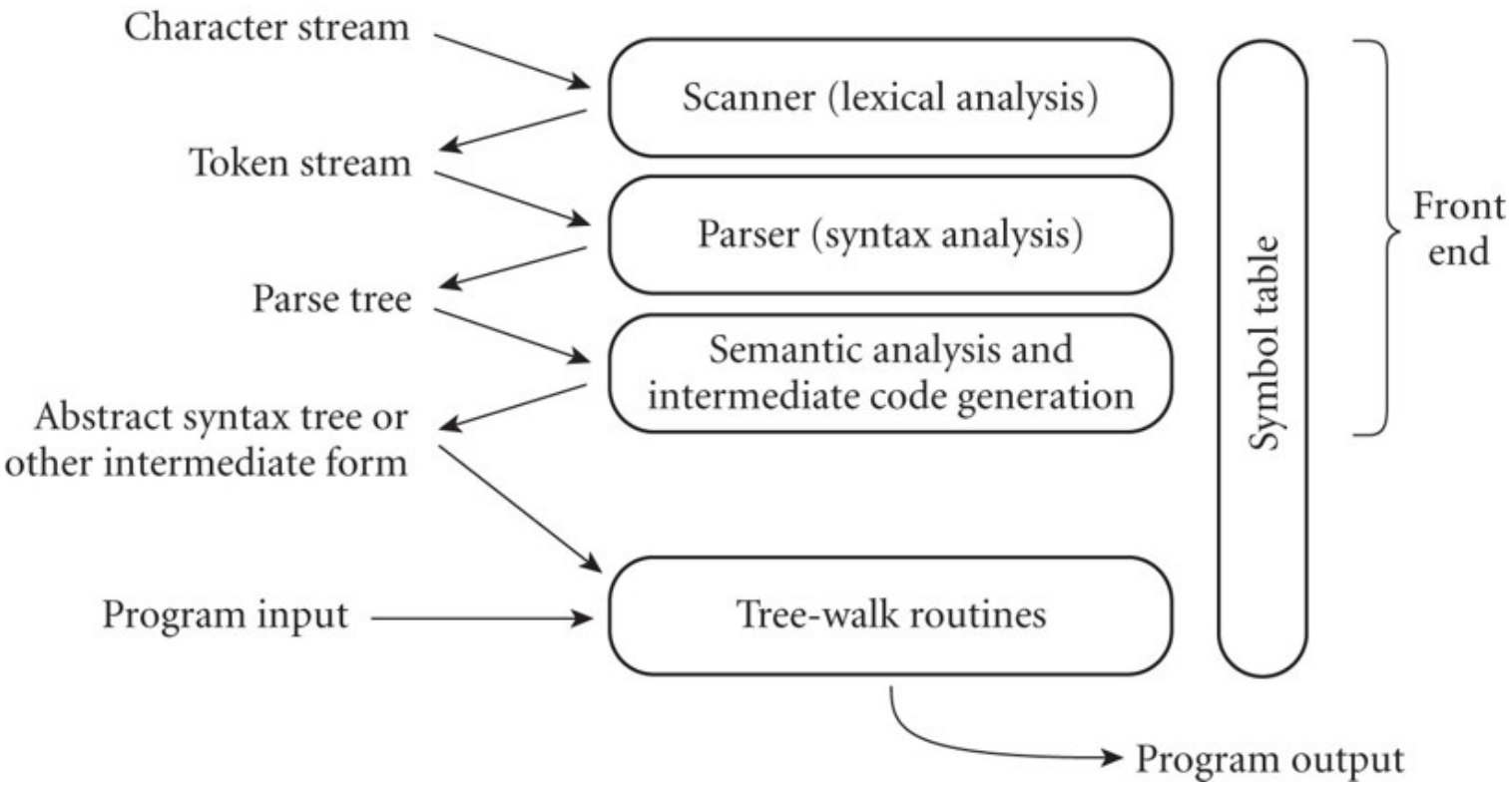


- De fleste språk bruker en eller annen hybridløsning
- Les kapittel 1.4

Faser i kompilering



Faser i intrepretering



Scanner

- En skanner deler programmet opp i tokens (som i kalulatoren vi lagde før sommeren)
- Token er de minste delene av et program som gir mening
- Skanneren leser tegn som:
  'i' 'n' 't' '␣' 'm' 'a' 'i' 'n' '(' ')'
- og grupperer dem til tokens som
  int main ( )
- Skanneren vil også ofte fjerne kommentarer og merke tokens med linjenummer for at kompilatoren skal kunne gi bedre feilmeldinger

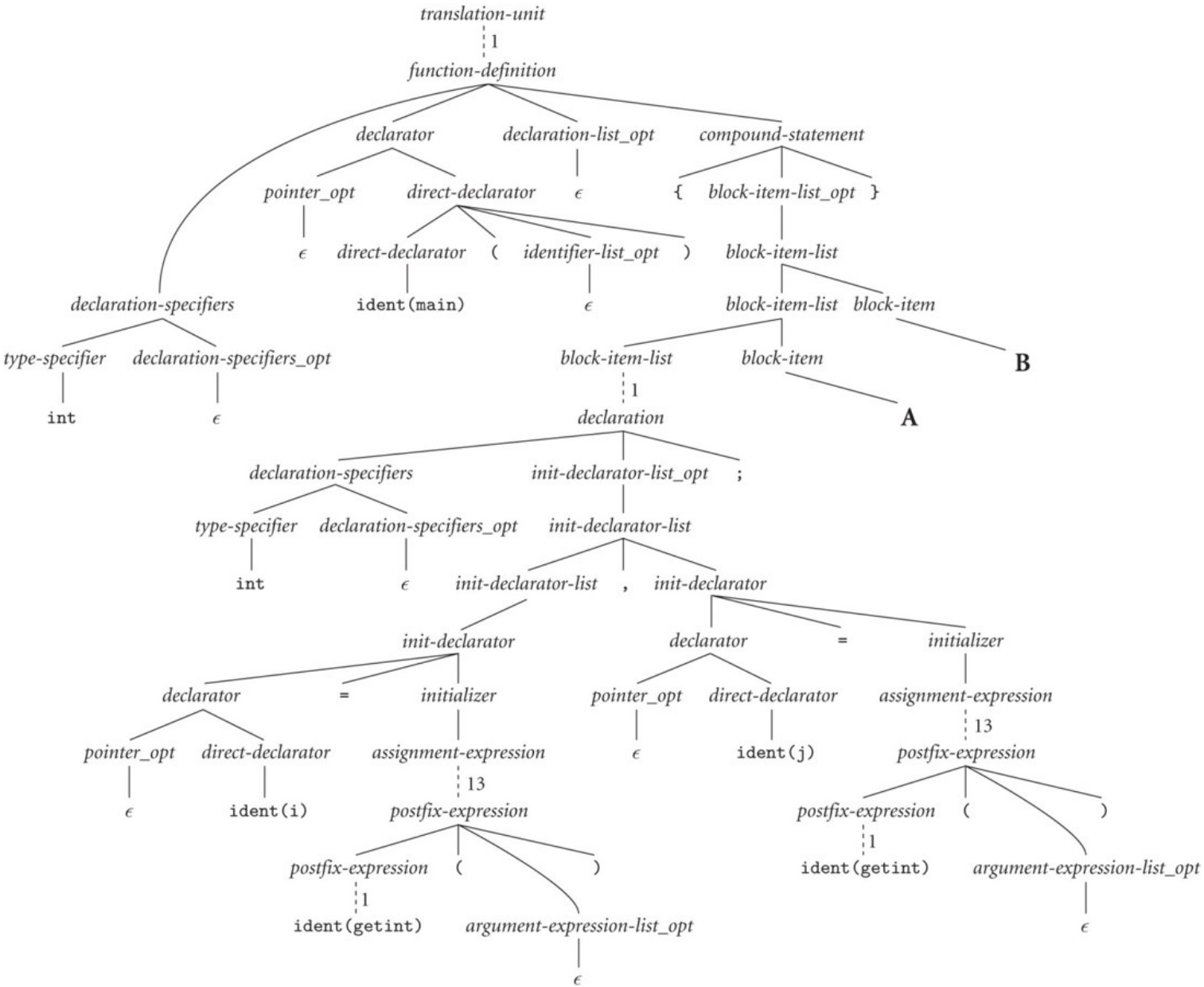
Parser

- En pasrser bruker en strøm av tokens for å lage et parse tree
- for å gjøre det trenger den en grammatikk.
- Om vi har C-programmet

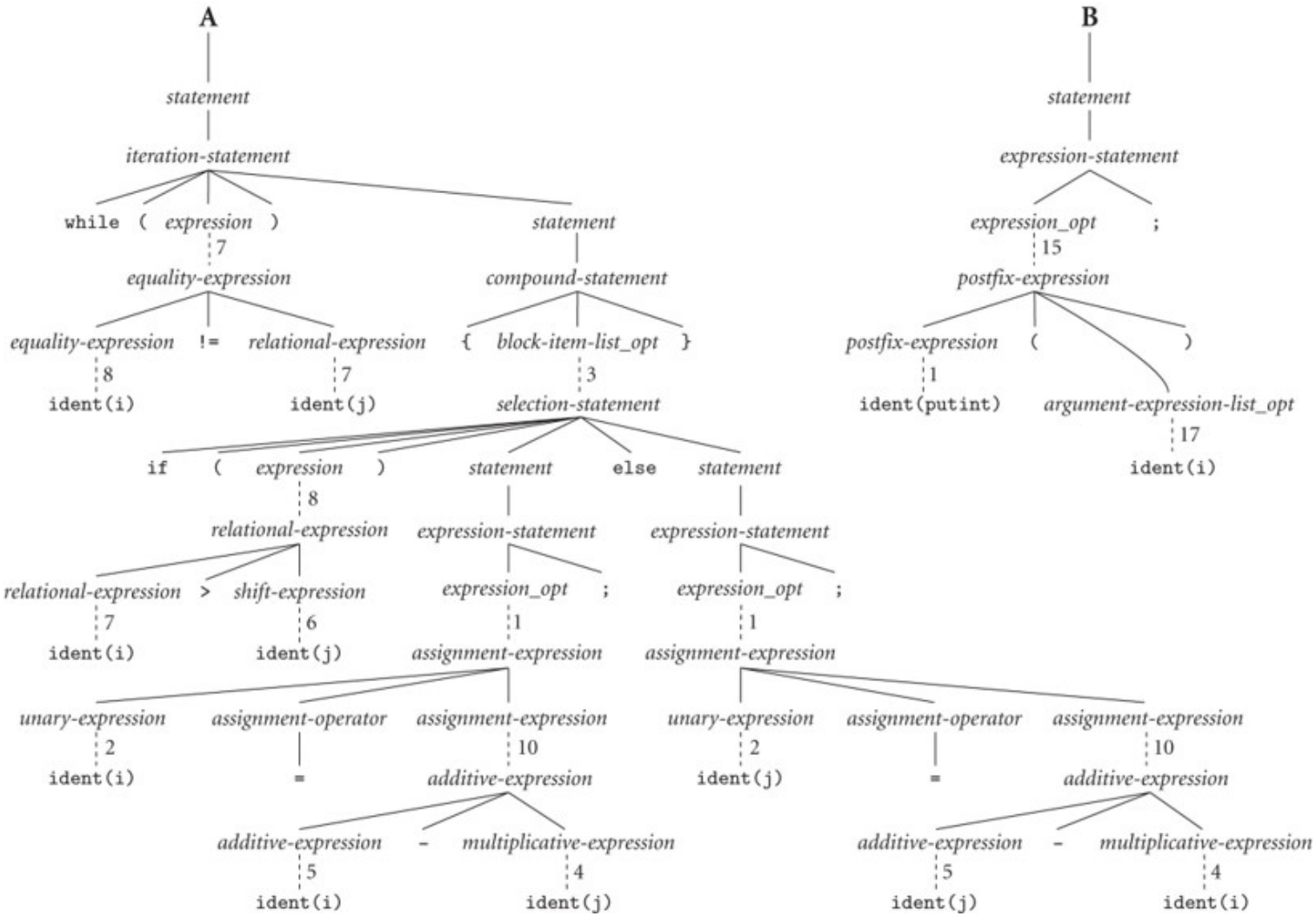
```
int main(){
   int i = getint(), int j = getint();
   while (i != j){
      if (i>j) i 0 i - j;
      else j = j - i;
   }
   putint(i);
}
```

- Får vi et parse tree som ser slik ut....

Parse tree del 1



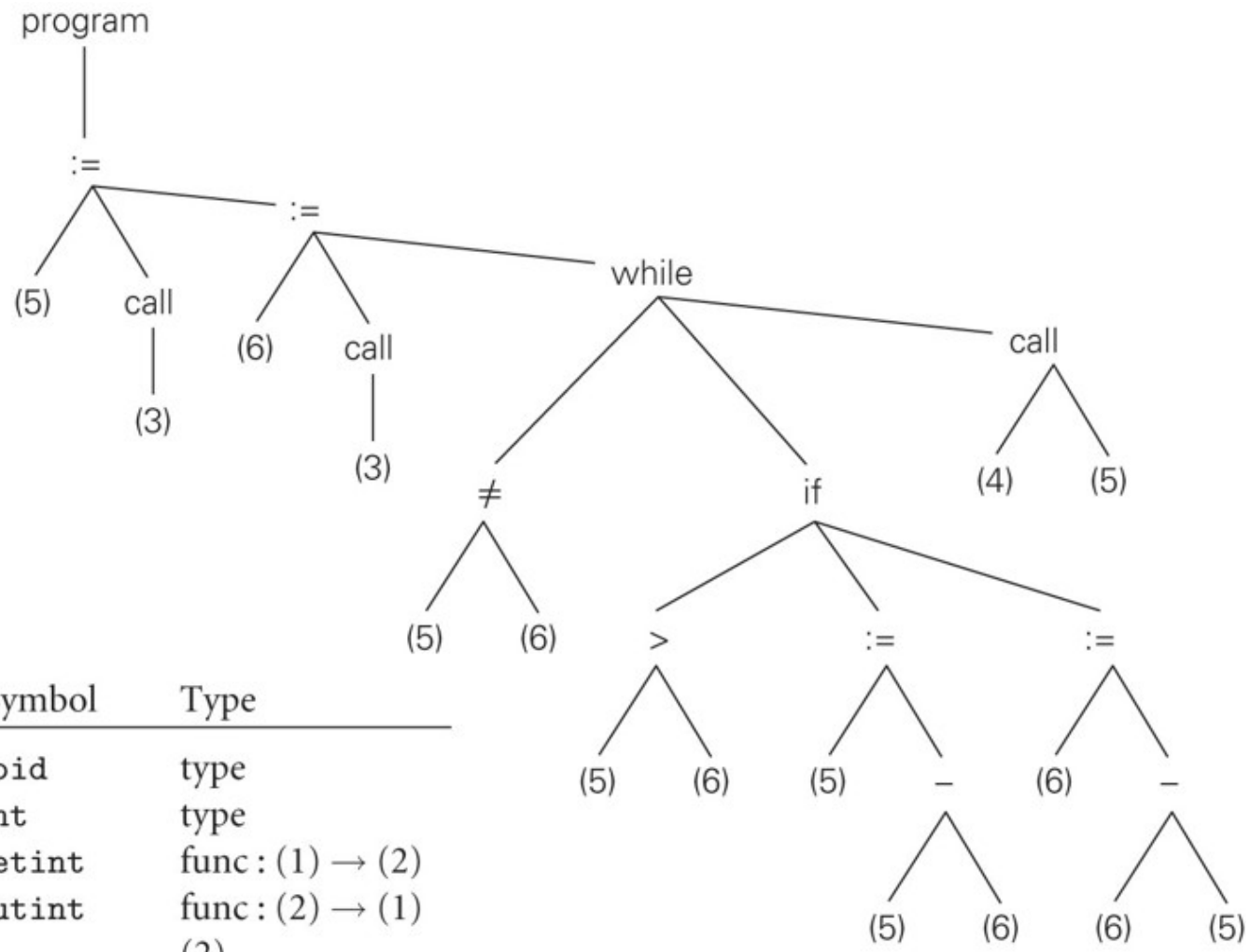
Parse tree del 2



Semantisk analyse

- Semantisk analyse er å finne meningen i er program
- I et C-program vil den sjekke ting som at:
  - Alle variabler og funksjoner er deklarert før de blir brukt
  - Kall på funksjoner har riktig antall og typer på parametere
  - Alle fuksjoner uten void som returtyper returnerer noe
- Dette er ting grammatikken ikke sier noe om
- Den bygger gjerne en symboltabell for å holde styr på navn og typer
- Den lager også et abstrakt syntakstre
- for vårt lille gcd program vil det kunne se slik ut.....

abstrakt syntakstre og symboltabell



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)