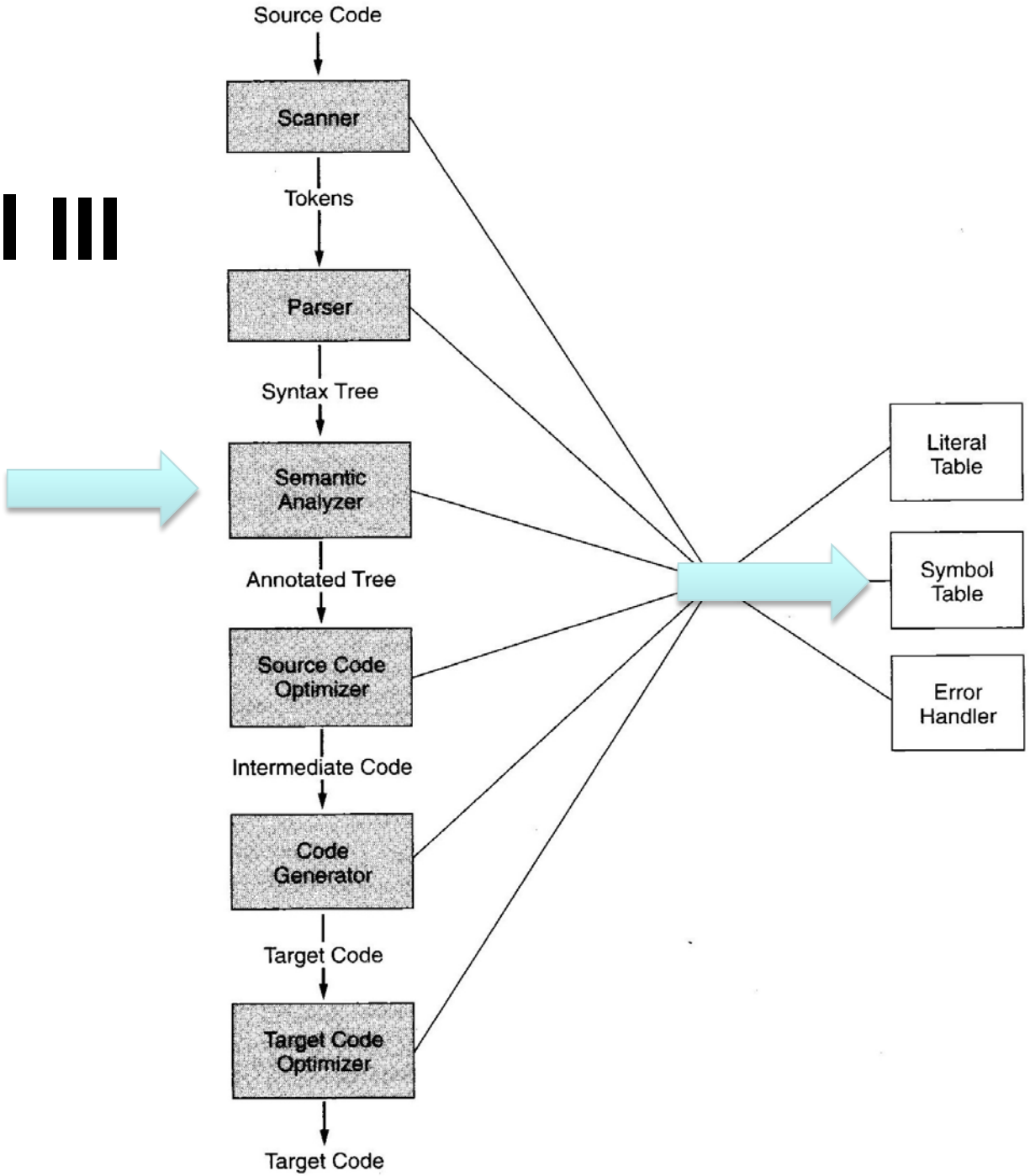
Semantisk Analyse del III

Typesjekking

Kapittel 6.4



INF 5110 - 2013

Datatyper og typesjekking

- Om typer generelt
  - Hva er typer?
  - Statisk og dynamisk typing
  - Hvordan beskrive typer syntaktisk?
  - Hvordan lagre dem i kompilatoren?
- Gjennomgang av noen typer
  - Grunntyper
  - Type-konstruktører
  - Verdisett og operasjoner
  - Lagring under utførelse (mer i kap 7)
  - Run-time tester og spesielle problemer: array/record/union/peker/...

- Hva vil det si at to variableuttrykk har samme type?
- Hvordan utføre selve type-sjekkingen?

Generelt om typer

En type er en mengde verdier, sammen med et sett av assosierte operasjoner

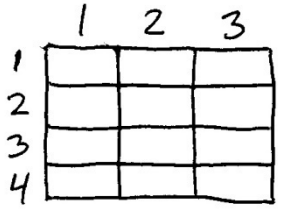
```
integer                         +, -, *, /, ...

real                            +, -, *, /, ...

array[0..9] of real             a[i+2]

record
  integer i;                    r.x
  real x
end
```

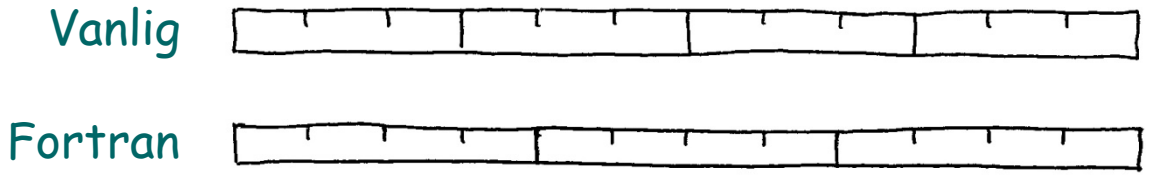
Array

array [<indekstype>] of <komponent-type>

- Verdier: Mengden av funksjoner fra indeks-typen til komponent-typen
- Indekstypen:
  - Bare heltall, fra ... til?
  - Andre typer: oppramstyper, character
- Operasjoner: indeksering `A[i+2]`
- Ting å tenke på
  - Indeksering uten for grensene
  - Er grensene kompilator-kjente?
- Implementasjon
  - En-dimensjonale
  - To og flere dimensjoner

```
array [1..4] of array [1..3] of real
array [1..4, 1..3] of real
```

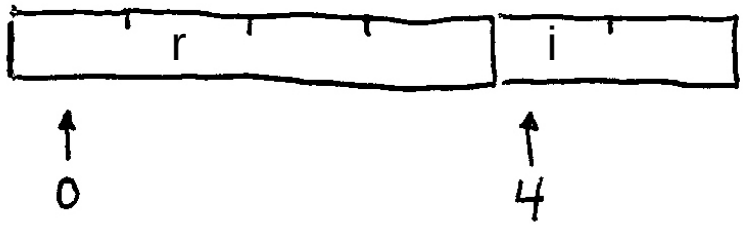




Record/Struct

```
struct {
    real r;
    int i
}
```

- Verdier: Alle verdier av det tilsvarende kartesiske produkt (real * int)
- Operasjoner: attributt-aksess (dot-notasjon) `x.i`
- Ting å tenke på: Lite
- Implementasjon
  - Layout der attributtene ligger etter hverandre

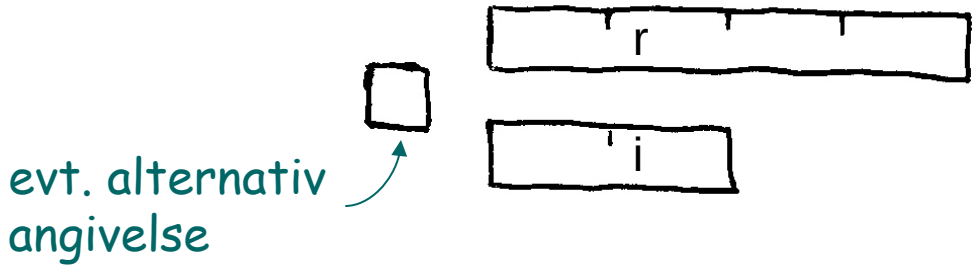


  - Attributtene får faste relativadresse, som gjør attributt-aksess enkelt

Union-typer

```
union {
  real r;
  int i
}
```

- Verdier: Unionen av alle par (variant, verdi av tilhørende type)
- Operasjon: aksess (dot-notasjon) `u.i`
- Ting å tenke på
  - Hvordan sikre at verdier settes inn og leses av som samme alternativ?
    - Mange språk overlater det til brukeren
    - Pascal har variant-record med angivelse av alternativ
    - Hva med klasser/subklasser?
- Implementasjon



  - Alle alternativer får samme relativ-adresse, gjerne 0

Peker-typer

```
^ integer          Pascal
integer*           C
```

- Verdier: Adresser til objekter/verdier av den aktuelle typen
- Operasjoner: dereferencing, dvs objektet/verdien som det pekes til

```
var a: ^integer;
var b: integer
...
a:= &i   eller a:= new integer;
b:= a^ + b;
```

Avh. av språket blir dereferencing ofte forenklet og slått sammen med dot-notasjon

- Ting å tenke på
  - Må teste på none (nil, null, ...)
  - Fri peking til variable kan gi problemer i blokk-strukturerte språk

Peker-typer, eksempel

```
{
  ^int a;
  int b;
  void p(){
    int c;
    ...
    a = &c;
    ...
  }
  ...
  p();
  ...
  b = a^ + 1
}
```

Flere kall som bruker samme areal som P()

- Løsning: gjør forskjell på
  - Vanlige deklarerte variable (på stakken)
  - Variable generert ved 'new' (på heapen)
  - Og, det er bare den siste typen man kan ha pekere til

Funksjon/prosedyre/metode/subrutine

- Generelt kan man lage en type av heading på en funksjon

```
var f: procedure (integer): integer;          Modula 2

int (*f)(int)                                 C
```

- Verdier
  - Alle funksjons-deklarasjoner (med setninger) som stemmer med headingen.
- Ting å tenke på
  - Vanlige prosedyre/funksjons-deklarasjoner er da å betrakte som konstanter av denne typen
  - Problemer med blokk-struktur og helt frie prosedyre-variable

Funksjon/..., eksempel

```
program
   var pv: procedure (integer);

   procedure Q();
      var a: integer;
      procedure p(integer i)
         a := a + i;
      end;
      ...
      pv := p
      ...
   end;
   ...
   Q();
   pv(3);
end
```

'a' finnes ikke

Men:
Prosedyrer som parametre til prosedyrer gir ingen slike problemer

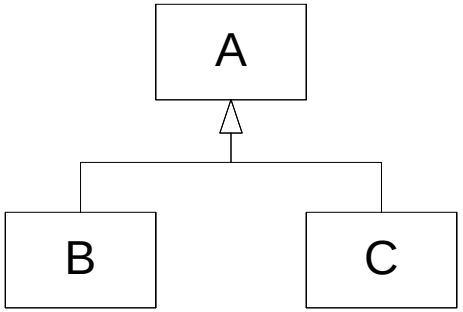
Klasser og subklasser

- Som *type* brukes klasser mest til å type pekere

```
class A {
  int i;
  void f()
{...}
}
```

```
class B
extends A {
  int i;
  void f()
{...}
}
```

```
class C
extends A {
  int i;
  void f()
{...}
}
```




```
A rA
```

- Klasser ligner på records, med følgende tillegg
  - Kan ha lokale metoder/prosedyrer/funksjoner
  - Kan ha subklasser
  - Objekter kan ofte bare dannes dynamisk, og ingen peking inn i stakken
  - Polymorfi: A-pekere kan også peke til B-objekter og til C-objekter
  - To aksess-måter ved ’rA.id’
    - Vanlig (ikke-virtuell): rA.i og rA.f() gir begge i og f i A, uavh. av hva rA peker på
    - Virtuell aksess: rA.f() gir f i klassen for det aktuelle objekt, som rA peker på
  - Spesielle problemer: merkelig få

Rekursive datatyper

```
datatype intBST = Nil | Node of int*intBST*intBST
```

ML

C

```
struct intBST
{ int isNull;
  int val;
  struct intBST left,right;
};
```

```
struct intBST
{ int val;
  struct intBST *left,*right;
};
typedef struct intBST * intBST;
```

```
typedef struct intBST * intBST;
struct intBST
{ int val;
  intBST left,right;
};
```

Diverse

- Overloading
  - Vanlig for standard-operasjonene
  - Ved egen def. av funskjoner
  - Implementasjon: to greie muligheter
    - Legge parametertypene inni navnet
    - La lookup levere en mengde med alternativer
- Type-konvertering
  - Kan gi problemer sammen med overloading
- Polymorfisme

```
procedure max (x,y: integer): integer;
procedure max (x,y: real): real;
```

```
void f(int i, double d)
void f(double d, int i)
```

```
f(i1,i2)
```

```
procedure swap (var x,y:anytype);
```

```
var x,y: integer;
    a,b: char;
. . .
swap(x,y);
swap(a,b);
swap(a,x);
```

Neste forelesning: Tirsdag 12. mars (seminarrom Sed)

TYPE-LIKHET OG TYPE-SJEKKING OPPGAVE 6.20 OG 6.21