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

- Hva er en type
  - En mengde verdier
  - Assosierte operasjoner

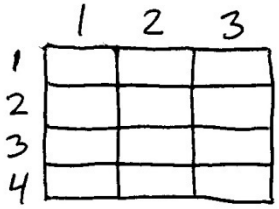
```
integer                  +, -, *, /, ...

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

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

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

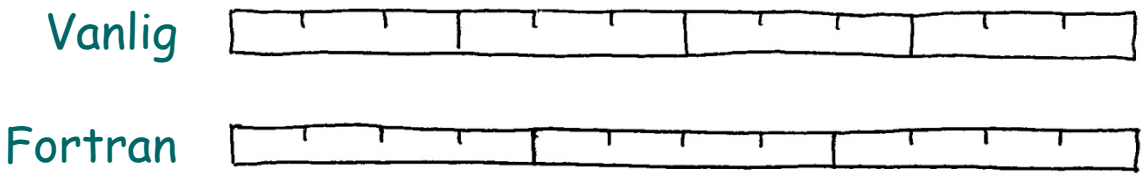
Array

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

- Verdier: Mengden av funksjoner fra indeks-typen til komponent-typen
- Indekstypen:
  - Bare heltal, 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
```

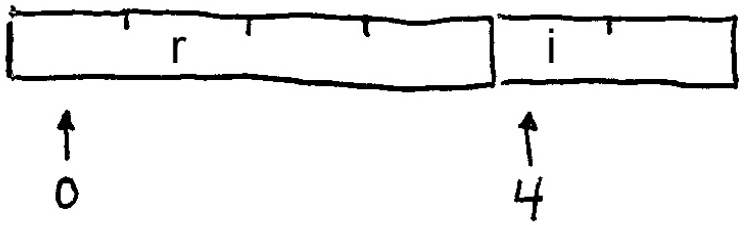




Record/Struct

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

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

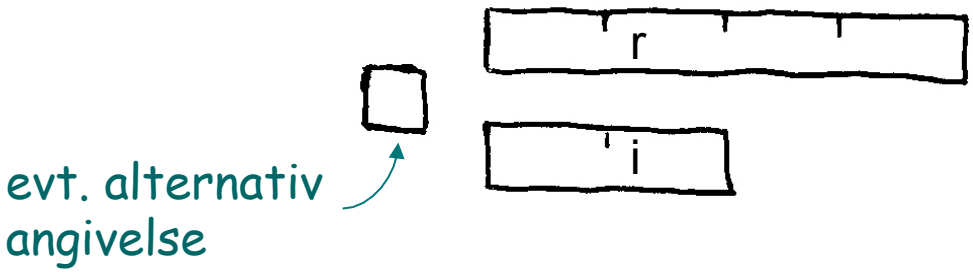


  - Attributene får faste relativadresse, som gjør attributt-aksess enkelt

Union-typer

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

- Verdier: Unionen av alle par (variant, verdi af 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
    - Sammenlignet med klasser/subklasser?
- Implementasjon



  - Alle alternativer får samme relativ-adresse, gjerne 0

Variant record

```
record case isReal: boolean of
   true:(r: real);
   false:(i: integer);
end;
```

```
record case boolean of
   true:(r: real);
   false:(i: integer);
end;
```

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 test 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 headig 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

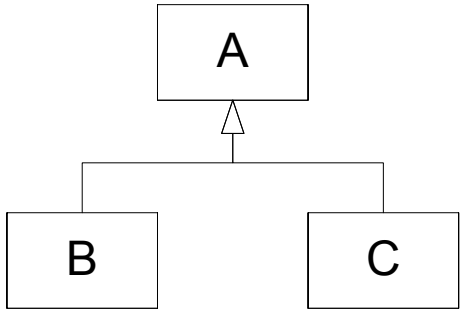
Klasser og subklasser

- 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
  - Kan ha subklasser
  - Objekter kan ofte bare skapes 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: **`rA.i`** og **`rA.f()`** gir **`i`** og **`f`** i A, uavh. av hva **`rA`** peker på
    - Virtuell aksess: **`rA.f()`** gir **`f`** i klassen for det objekt, som **`rA`** peker på
  - Spesielle problemer: merkelig få

Rekursive datatyper

ML

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

C

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

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

Når er to typer like?

```
var a,b:
       record
         int i;
         double d
       end
```

```
var c: record
         int i;
         double d
       end
```

```
typedef idRecord:record
                   int i;
                   double d
                 end
```

```
var d: idRecord;
var e: idRecord;
```

```
a:= c;        a:= b
a:= d;        d:= e
```

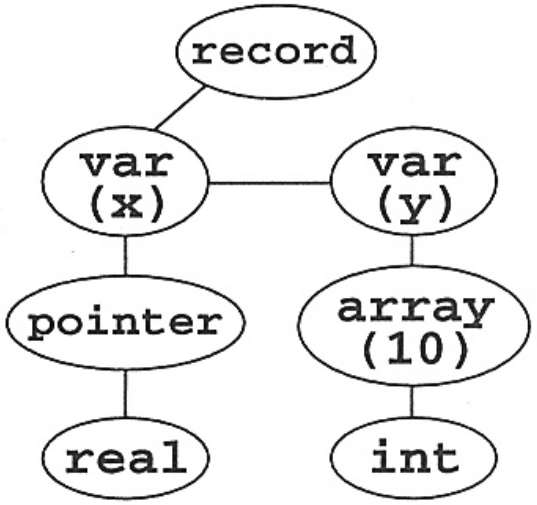
Strukturlikhet eller navnelikhet?

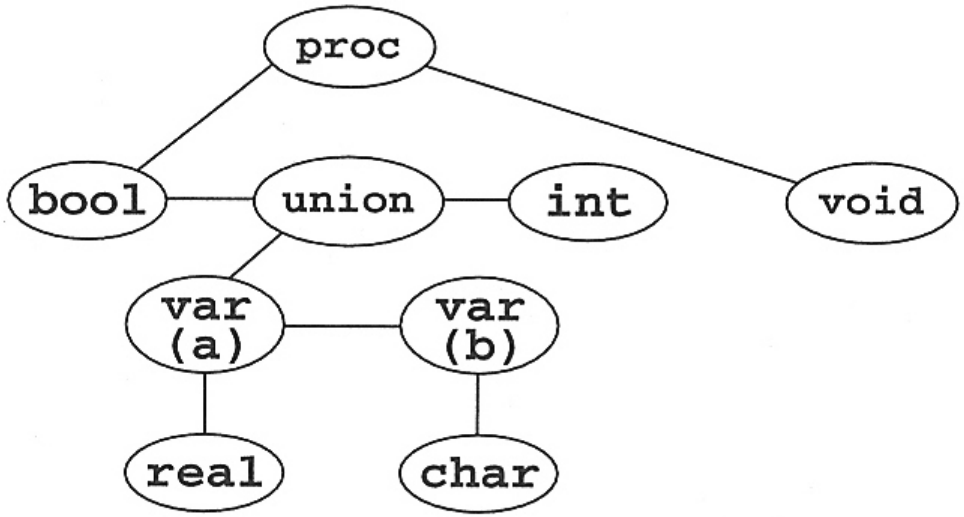
- Finnes det en type-definisjon?
  - Om ikke: Må bruke struktur-likhet
  - Om de finnes: Kan kreve full navnelikhet, eller en blanding

Type-uttrykk representert som (abstrakt) syntakstre

```
proc(bool,union a:real; b:char end,int):void
```

```
record
   x: pointer to real;
   y: array [10] of int
end
```



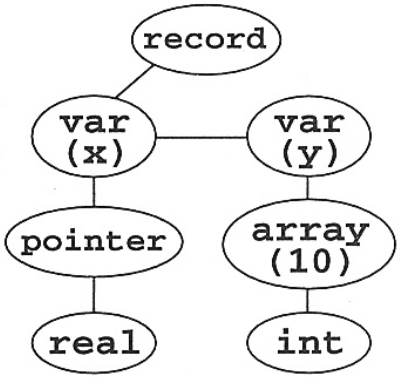


Eksempel I

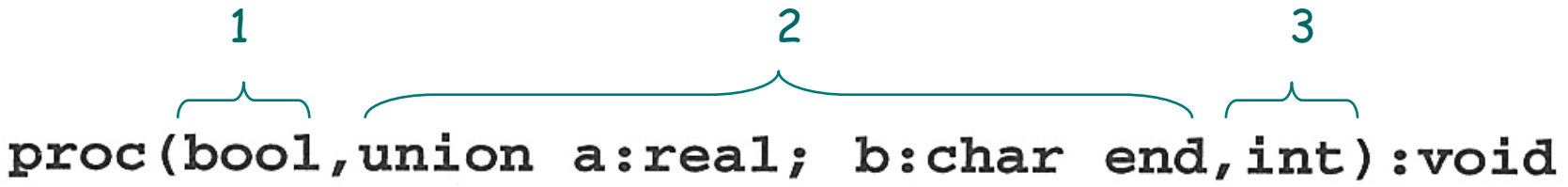
- Grammatikk uten typedef, med strukturlikhet av typer

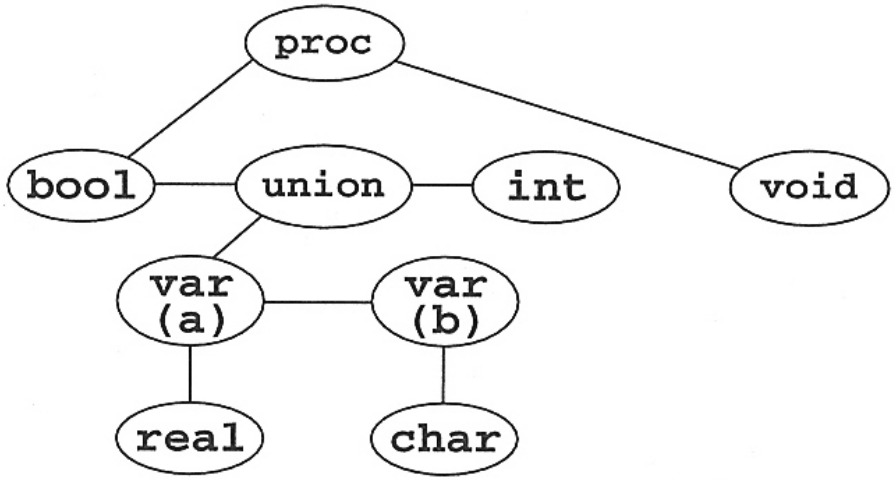
var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id :** *type-exp*
type-exp → *simple-type* | *structured-type*
simple-type → **int** | **bool** | **real** | **char** | **void**
structured-type → **array [num] of** *type-exp* |
record *var-decls* **end** |
union *var-decls* **end** |
pointer to *type-exp* |
proc (*type-exps* **)** *type-exp*
type-exps → *type-exps* **,** *type-exp* | *type-exp*

```
record
  x: pointer to real;
  y: array [10] of int
end
```



Eksempel II


1
2
3
proc(bool,union a:real; b:char end,int):void


proc
bool
union
int
void
var (a)
var (b)
real
char

Test av om to typer er like (struktur-likhet)

```
function typeEqual ( t1, t2 : TypeExp ) : Boolean;
var temp : Boolean ;
    p1, p2 : TypeExp ;
begin
  if t1 and t2 are of simple type then return t1 = t2
  else if t1.kind = array and t2.kind = array then
     return t1.size = t2.size and typeEqual ( t1.child1, t2.child1)
  else if t1.kind = record and t2.kind = record
       or t1.kind = union and t2.kind = union then
  begin
    p1 := t1.child1 ;
    p2 := t2.child1 ;
    temp := true ;
    while temp and p1 ≠ nil and p2 ≠ nil do
        if p1.name ≠ p2.name then
          temp := false
        else if not typeEqual ( p1.child1 , p2.child1 )
        then temp := false
        else begin
          p1 := p1.sibling ;
          p2 := p2.sibling ;
        end;
    return temp and p1 = nil and p2 = nil ;
  end
  else if t1.kind = pointer and t2.kind = pointer then
     return typeEqual ( t1.child1 , t2.child1 )
  else if t1.kind = proc and t2.kind = proc then
  begin
    p1 := t1.child1 ;
    p2 := t2.child1 ;
    temp := true ;
    while temp and p1 ≠ nil and p2 ≠ nil do
       if not typeEqual ( p1.child1 , p2.child1 )
       then temp := false
       else begin
         p1 := p1.sibling ;
         p2 := p2.sibling ;
       end;
    return temp and p1 = nil and p2 = nil
           and typeEqual ( t1.child2 , t2.child2 )
  end
  else return false ;
end ; (* typeEqual *)
```

```
else if t1 and t2 are type names then
   return typeEqual(getTypeExp(t1), getTypeExp(t2))
```

Navne-likhet

- Alle typer må gis navn, og type-likhet krever samme navn

```
record
   x: pointer to real;
   y: array [10] of int
end
```

```
t1 = pointer to real;
t2 = array [10] of int;
t3 = record
        x: t1;
        y: t2
end
```

var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id** **:** *simple-type-exp*
type-decls → *type-decls* **;** *type-decl* | *type-decl*
type-decl → **id** **=** *type-exp*
type-exp → *simple-type-exp* | *structured-type*
simple-type-exp → *simple-type* | **id**
simple-type → **int** | **bool** | **real** | **char** | **void**
structured-type → **array** **[num]** **of** *simple-type-exp* |
record *var-decls* **end** |
union *var-decls* **end** |
pointer to *simple-type-exp* |
proc(*type-exps* **)** *simple-type-exp*
type-exps → *type-exps* **,** *simple-type-exp* | *simple-type-exp*

} type-dekl

```
function typeEqual ( t1, t2 : TypeExp ) : Boolean;
var temp : Boolean ;
    p1, p2 : TypeExp ;
begin
   if t1 and t2 are of simple type then
      return t1 = t2
   else if t1 and t2 are type names then
      return t1 = t2
   else return false ;
end;
```

Bare navnelikhet (eller samme basaltype) gir likhet

Struktur-likhet, hvor man også tillater navn

else if *t1 and t2 are type names* **then**
  return *typeEqual*(*getTypeExp*(*t1*), *getTypeExp*(*t2*))

tilføyes som siste alternativ
i struktur-likhets-testen

Da skal her `t1` og `t2` være samme typen:

```
t1 = record
       x: int;
       t: pointer to t2;
     end;

t2 = record
       x: int;
       t: pointer to t1;
     end;
```

Type-deklarasjon ekvivalens

- type alias (Pascal, C)

```
t2 = t1;
```

```
t1 = int;
t2 = int
```

```
t1 = array [10] of int;
t2 = array [10] of int;
t3 = t1;
```

Sjekking av type-riktighet for uttrykk, program, etc

program → *var-decls* **;** *stmts*
var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id** **:** *type-exp*
type-exp → **int** | **bool** | **array** **[num]** **of** *type-exp*
stmts → *stmts* **;** *stmt* | *stmt*
stmt → **if** *exp* **then** *stmt* | **id** **:=** *exp*

exp → *exp* + *exp* | *exp* **or** *exp* | *exp* **[** *exp* **]**

Sjekking av type

Grammar Rule	Semantic Rules
var-decl → **id** : *type-exp*	*insert*(**id** .*name*, *type-exp.type*)
type-exp → **int**	*type-exp.type* := *integer*
type-exp → **bool**	*type-exp.type* := *boolean*
$type\text{-}exp_1$ → **array** [**num**] **of** $type\text{-}exp_2$	$type\text{-}exp_1$.*type* := *makeTypeNode*(*array*, **num** .*size*, $type\text{-}exp_2$.*type*)
stmt → **if** *exp* **then** *stmt*	**if not** *typeEqual*(*exp.type*, *boolean*) **then** *type-error*(*stmt*)
stmt → **id** := *exp*	**if not** *typeEqual*(*lookup*(**id** .*name*), *exp.type*) **then** *type-error*(*stmt*)
exp_1 → exp_2 + exp_3	**if not** (*typeEqual*(exp_2 .*type*, *integer*) **and** *typeEqual*(exp_3 .*type*, *integer*)) **then** *type-error*(exp_1) ; exp_1 .*type* := *integer*
exp_1 → exp_2 **or** exp_3	**if not** (*typeEqual*(exp_2 .*type*, *boolean*) **and** *typeEqual*(exp_3 .*type*, *boolean*)) **then** *type-error*(exp_1) ; exp_1 .*type* := *boolean*
exp_1 → exp_2 [exp_3]	**if** *isArrayType*(exp_2 .*type*) **and** *typeEqual*(exp_3 .*type*, *integer*) **then** exp_1 .*type* := exp_2 .*type.child1* **else** *type-error*(exp_1)
exp → **num**	*exp.type* := *integer*
exp → **true**	*exp.type* := *boolean*
exp → **false**	*exp.type* := *boolean*
exp → **id**	*exp.type* := *lookup*(**id** .*name*)

Her er rekkefølgen ventre -> høyre viktig

Kunne også ha
- record
- pointer
- ...

Gir nå typen på gjeldende dekl av dette navnet

Kommentarer til tabell 6.10

- Bruker synboltabell, som man setter inn i (men forblir samme tabell)
- Mer realistisk enn i tabell 6.9. Der ga en innsetting en helt ny tabell
- Til gjengjeld blir rekkefølge viktig: Den følger ikke av noen avhengighetsgraf f.eks. for insert(...)
- Rekkefølge: dybde-først, venstre -> høyre
- Attributter
  - exp: type
  - type-exp: type
- Funksjoner
  - Insert(navn,type) ---
  - typeEqual(type1, type2) boolean
  - Lookup(navn) type
  - Type-error(tre-node) ---
    - Skriver ut feilmelding, med mindre et subtre har type=error

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);
```

procedure(var anytype, var anytype): void

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