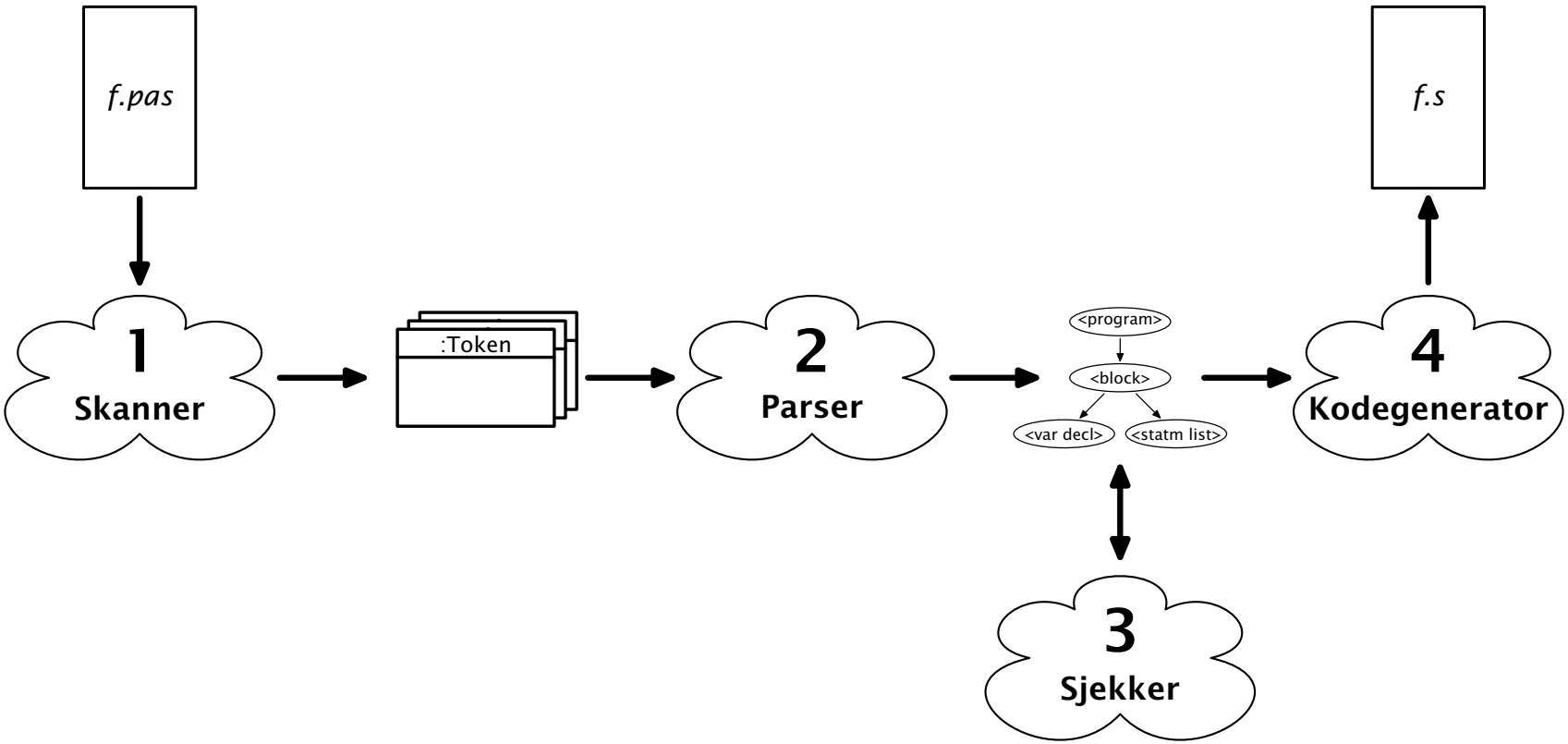
Dagens tema:

Maskinkode

- Program, prosedyrer og funksjoner
  - Blokker
- Tilordning
- Array-er (frivillig)
- Biblioteket
- Hovedprogrammet

Prosjektoversikt



Program, prosedyrer og funksjoner

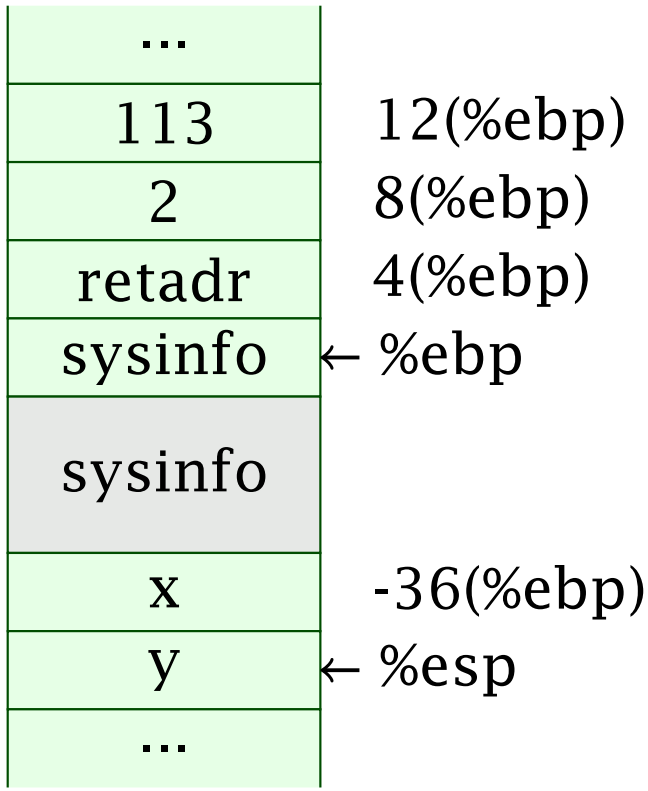
Disse kan behandles likt når det gjelder kodegenerering:

- Alle tre kalles (for program kalles av os-et)
- Alle kan ha parametre (unntatt program)
- Alle oppretter en **blokk**.
- Alle utfører noen setninger.
- Alle rydder opp etter seg og avslutter.

Alt ligger på stakken

Anta at vi utfører kallet
`f(2,'q')`
på funksjonen

```
procedure f(a:integer, b:char);
var x:integer;
    y: char;
begin
  :
end; {f}
```



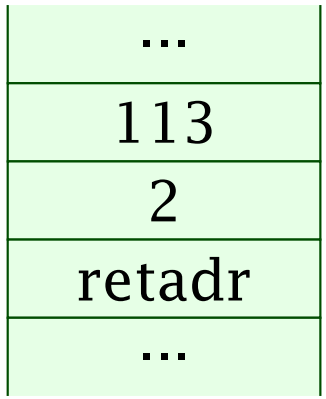
Prosedyre- og funksjonskall

Kallet oversettes slik:

1. Legg parametrene på stakken i *omvendt* rekkefølge.
2. Gjør kallet med **call**-instruksjonen.

Alt ligger på stakken

For kallet
`f(2,'q')`
ser det slik ut:



I programmet/prosedyren/funksjonen

Programmet/prosedyren/funksjonen må sette av plass til en blokk. Da må den vite to ting:

1. Hvor mange byte trengs til lokale variabler?
2. Hvilket **blokknivå** er jeg på?

Blokknivå

Blokkens nivå er rett og slett hvor dypt den er deklarert:

- Programmet er nivå 1.
- Prosedyrer deklarert i programmet er på nivå 2.
- Prosedyrer deklarert i en prosedyre på nivå n, er på nivå $n + 1$.

Alt ligger på stakken

```
program Blokker;
var Level1: Integer;  /* Level 1 */

procedure P2;
var Level2: Integer;  /* Level 2 */

procedure P3;
var Level3: Integer;  /* Level 3 */
    Info: Integer;
begin
end; { P3 }

begin
end; { P2 }

begin
end. { program }
```

Variabelplass

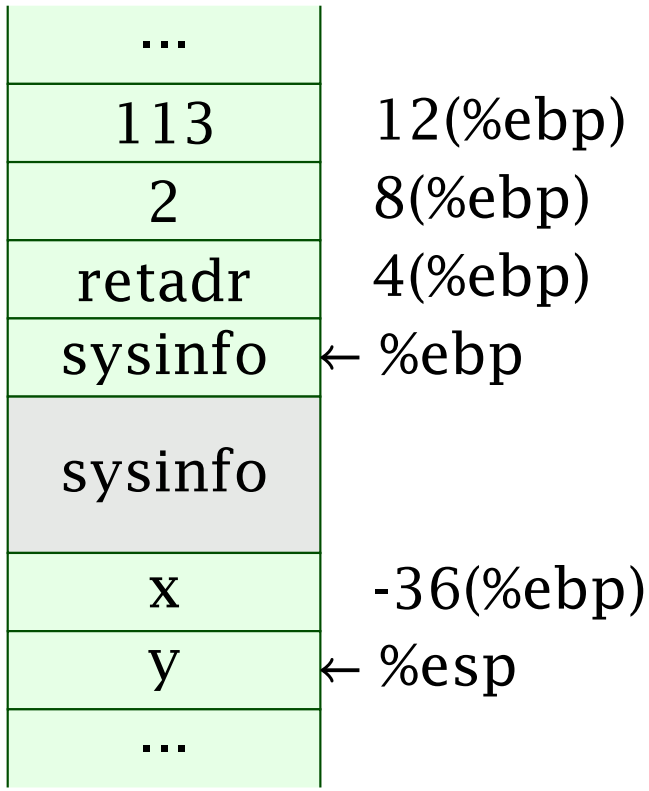
Hvor mye plass tar variabelene i en blokk? Det må vi telle opp selv: hver variabel trenger 4 byte. (Array-er tar vi senere.)

Alt ligger på stakken

Instruksjonen

```
enter $32+D, $L
```

der D er lokal variabelplass og L er blokknivået.



Avslutning av blokker

Blokker avsluttes med

```
leave
ret
```

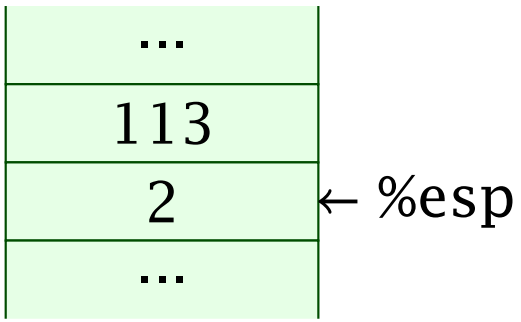
og da fjernes blokken fra stakken.

Avslutning av kalleren

Kalleren må fjerne parametrene fra stakken med

```
addl    $4N,%esp
```

der N er antall parametre.



Alt ligger på stakken

```
program Kall;

procedure f(a: integer;
            b: char);
var x: integer;
    y: char;
begin
end; { f }

begin
   f(2, 'q')
end.
```

```
# Code file created by Pascal2100 compiler 2015-11-11 01:30:20
        .extern write_char
        .extern write_int
        .extern write_string
        .globl  _main
        .globl  main
_main:
main:   call    prog$kall_1             # Start program
        movl    $0,%eax                 # Set status 0 and
        ret                             # terminate the program
proc$f_2:
        enter   $40,$2                  # Start of f
        leave                           # End of f
        ret
prog$kall_1:
        enter   $32,$1                  # Start of kall
        movl    $113,%eax               #   char 113
        pushl   %eax                    # Push param #2.
        movl    $2,%eax                 #   2
        pushl   %eax                    # Push param #1.
        call    proc$f_2
        addl    $8,%esp                 # Pop parameters.
        leave                           # End of kall
        ret
```

Variabler

Variabler ligger i en eller annen blokk på stakken, nøyaktig hvor angis med et **offset** i forhold til %EBP. Dette tillegget må vi finne selv:

- Første parameter har tillegg 8, neste 12, så 16 osv.
- Første variabel har tillegg -36, neste -40, så -44 osv.

(Det blir litt mer avansert når vi skal håndtere array-er.)

Hvor ligger variablene?

Når vi skal lese en variabel i et uttrykk, trenger vi den *blokknivå* og `offset`:

```
movl    -4L(%ebp),%edx
movl    T(%edx),%eax
```

der `L` er blokknivå og `T` er offset.

Det samme gjelder for tilordning.

Hvor ligger variablene?

```
program Assign;
var v : Integer;
   w  : Integer;
begin
   w := v
end.
```

```
# Code file created by Pascal2100 compiler 2015-11-11 01:46:51
        .extern write_char
        .extern write_int
        .extern write_string
        .globl  _main
        .globl  main
_main:
main:   call    prog$assign_1           # Start program
        movl    $0,%eax                 # Set status 0 and
        ret                             # terminate the program
prog$assign_1:
        enter   $40,$1                  # Start of assign
        movl    -4(%ebp),%edx
        movl    -36(%edx),%eax          #   v
        movl    -4(%ebp),%edx
        movl    %eax,-40(%edx)          # w :=
        leave                           # End of assign
        ret
```

Returverdien

I Pascal angir man returverdien ved å tilordne til funksjonen som om den var en variabel:

```
function Min(a: Integer; b: Integer): Integer;
begin
   if a < b then Min := a
   else Min := b
end; { Min }
```

Hvor skal denne returverdien lagres?

Hvor ligger variablene?

Løsningen er å legge den blant systeminformasjonen i blokken, i `-32(%ebp)`.

Det eneste da er å huske å legge den i %EAX ved retur.

Hvor ligger variablene?

```
func$min_2:
        enter   $32,$2                  # Start of min
                                        # Start if-statement
        movl    -8(%ebp),%edx
        movl    8(%edx),%eax            #   a
        pushl   %eax
        movl    -8(%ebp),%edx
        movl    12(%edx),%eax           #   b
        popl    %ecx
        cmpl    %eax,%ecx
        movl    $0,%eax
        setl    %al                     # Test <
        cmpl    $0,%eax
        je      .L0003
        movl    -8(%ebp),%edx
        movl    8(%edx),%eax            #   a
        movl    %eax,-32(%ebp)          # min :=
        jmp     .L0004
.L0003:
        movl    -8(%ebp),%edx
        movl    12(%edx),%eax           #   b
        movl    %eax,-32(%ebp)          # min :=
.L0004:                                 # End if-statement
        movl    -32(%ebp),%eax          # Fetch return value
        leave                           # End of min
        ret
```

Array-er (frivillig)

En array er en samling variabler av samme type. Siden grensene er konstanter, vet vi alltid hvor stor array-en er.

Til forskjell fra C og Java behøver ikke nedre grense være 0. Dette gjør oppslaget litt mer komplisert; se kodeskjemaet.

En array er bare en samling variabler

Hva blir anderledes når vi skal håndtere array-er?

- Beregne variablenes plass og offset
- Hente verdien i et array-element i et uttrykk
- Tilordne til et array-element

Biblioteket

Det meste i biblioteket benyttes bare under navnebindingen og sjekkingen og kan ignoreres under kodegenereringen. Det eneste unntaket er `write`:

- `write` må spesialbehandles under sjekkingen: kall på den kan ha vilkårlig mange parametre, og de kan være av enhver type.
- `write` må også spesialbehandles under kodegenereringen siden antallet parametre varierer.

Løsning: Betrakt

```
write(1, 'c', 'Abc')
```

som

```
write(1); write('c'); write('Abc')
```

Hvert kall på `write` genereres da som et kall på

`write_int` om parameteren er en Integer eller et delintervall av en Integer

`write_char` om det er en Char eller et delintervall av Char

`write_string` om det er et tekst av annen lengde enn 1[1]

[1] Koden for tekster er sær; se kodetabellene.

Hva er det med 'write'?

For virkelig å få skrevet ut, må vi kalle operativsystemfunksjoner. Vi gjør dette gjennom noen enkle C-funksjoner:

`libpas2100.c`

```
#include <stdio.h>

void write_char (int c)
{
  printf("%c", c);
}

void write_int (int n)
{
  printf("%d", n);
}

void write_string (char *s)
{
  printf("%s", s);
}
```

Disse kompileres til en biblioteksfil `libpas2100.a`[2] som kobles til det kjørbare programmet av `gcc`.

[2] Les «man ar».

Hovedprogrammet

Hvordan starter vi det hele? Vi vet at i Linux starter alltid programmet med funksjonen `main`.

1. Vi lager en falsk funksjon `main`.
2. Det eneste den gjør er å kalle hovedprogrammet. (Om du husker: Pascal2100-programmet ble kompilert akkurat som om det var en prosedyre.)

Et eksempel helt på tampen

```
program Demo;
type Int = Integer;
var a: array [1..2] of Int;
    i: integer;

function Min(a:Int; b:Int):Int;
begin
   if a < b then Min := a
   else Min := b
end; { Min }

begin
   a[1] := 5;  a[2] := -2;
   write('Min er ',
         Min(a[1],a[2]), eol)
end.
```

```
# Code file created by Pascal2100 compiler 2015-11-11 02:50:17
        .extern write_char
        .extern write_int
        .extern write_string
        .globl  _main
        .globl  main
_main:
main:   call    prog$demo_1             # Start program
        movl    $0,%eax                 # Set status 0 and
        ret                             # terminate the program
func$min_2:
        enter   $32,$2                  # Start of min
                                        # Start if-statement
        movl    -8(%ebp),%edx
        movl    8(%edx),%eax            #   a
        pushl   %eax
        movl    -8(%ebp),%edx
        movl    12(%edx),%eax           #   b
        popl    %ecx
        cmpl    %eax,%ecx
        movl    $0,%eax
        setl    %al                     # Test <
        cmpl    $0,%eax
        je      .L0003
```

Et eksempel helt på tampen

```
program Demo;
type Int = Integer;
var a: array [1..2] of Int;
    i: integer;

function Min(a:Int; b:Int):Int;
begin
   if a < b then Min := a
   else Min := b
end; { Min }

begin
   a[1] := 5;  a[2] := -2;
   write('Min er ',
         Min(a[1],a[2]), eol)
end.
```

```
        movl    -8(%ebp),%edx
        movl    8(%edx),%eax            #   a
        movl    %eax,-32(%ebp)          # min :=
        jmp     .L0004
.L0003:
        movl    -8(%ebp),%edx
        movl    12(%edx),%eax           #   b
        movl    %eax,-32(%ebp)          # min :=
.L0004:                                 # End if-statement
        movl    -32(%ebp),%eax          # Fetch return value
        leave                           # End of min
        ret
prog$demo_1:
        enter   $44,$1                  # Start of demo
        movl    $5,%eax                 #   5
        pushl   %eax
        movl    $1,%eax                 #   1
        subl    $1,%eax
        movl    -4(%ebp),%edx
        leal    -40(%edx),%edx
        popl    %ecx
        movl    %ecx,0(%edx,%eax,4)     # a[x] :=
```

Et eksempel helt på tampen

```
program Demo;
type Int = Integer;
var a: array [1..2] of Int;
    i: integer;

function Min(a:Int; b:Int):Int;
begin
   if a < b then Min := a
   else Min := b
end; { Min }

begin
   a[1] := 5;  a[2] := -2;
   write('Min er ',
         Min(a[1],a[2]), eol)
end.
```

```
        movl    $2,%eax                 #   2
        negl    %eax                    #   - (prefix)
        pushl   %eax
        movl    $2,%eax                 #   2
        subl    $1,%eax
        movl    -4(%ebp),%edx
        leal    -40(%edx),%edx
        popl    %ecx
        movl    %ecx,0(%edx,%eax,4)     # a[x] :=
        .data
.L0005: .asciz   "Min er "
        .align  2
        .text
        leal    .L0005,%eax             # Addr("Min er ")
        pushl   %eax                    # Push param #1.
        call    write_string
        addl    $4,%esp                 # Pop parameter.
        movl    $2,%eax                 #   2
        subl    $1,%eax
        movl    -4(%ebp),%edx
        leal    -40(%edx),%edx
        movl    0(%edx,%eax,4),%eax     #   a[...]
        pushl   %eax                    # Push param #2
```

Et eksempel helt på tampen

```
program Demo;
type Int = Integer;
var a: array [1..2] of Int;
    i: integer;

function Min(a:Int; b:Int):Int;
begin
   if a < b then Min := a
   else Min := b
end; { Min }

begin
   a[1] := 5;  a[2] := -2;
   write('Min er ',
         Min(a[1],a[2]), eol)
end.
```

```
        movl    $1,%eax                 #   1
        subl    $1,%eax
        movl    -4(%ebp),%edx
        leal    -40(%edx),%edx
        movl    0(%edx,%eax,4),%eax     #   a[...]
        pushl   %eax                    # Push param #1
        call    func$min_2
        addl    $8,%esp                 # Pop parameters
        pushl   %eax                    # Push param #2.
        call    write_int
        addl    $4,%esp                 # Pop parameter.
        movl    $10,%eax                #   char 10
        pushl   %eax                    # Push param #3.
        call    write_char
        addl    $4,%esp                 # Pop parameter.
        leave                           # End of demo
        ret
```