Diverse eksamensgaver

Noen har fått den idé å lage et språk hvor klasser kan ha noe tilsvarende ’by-value-result’-parametere. Klasser har ingen konstruktører, og ’by-value-result’ parametere spesifiseres som en del av en klassedefinisjon ’(val-res int v)’.

Parametrene kan brukes innen klassen som om de var vanlige variable. De aktuelle parametrene må være variable av riktig type, og ved generering av et objekt av klassen overføres adressene til disse variablene. Til forskjell fra call-by-result for funksjoner blir verdiene ikke returnert når et objekt er generert. For å få verdiene returnert må man kalle en predefinert metode return, som alle klasser har og som man derfor ikke trenger å erklære. Man kan således returnere verdier flere ganger, ved gjentatte kall på return, men vi skal dog ikke bruke dette i det følgende.

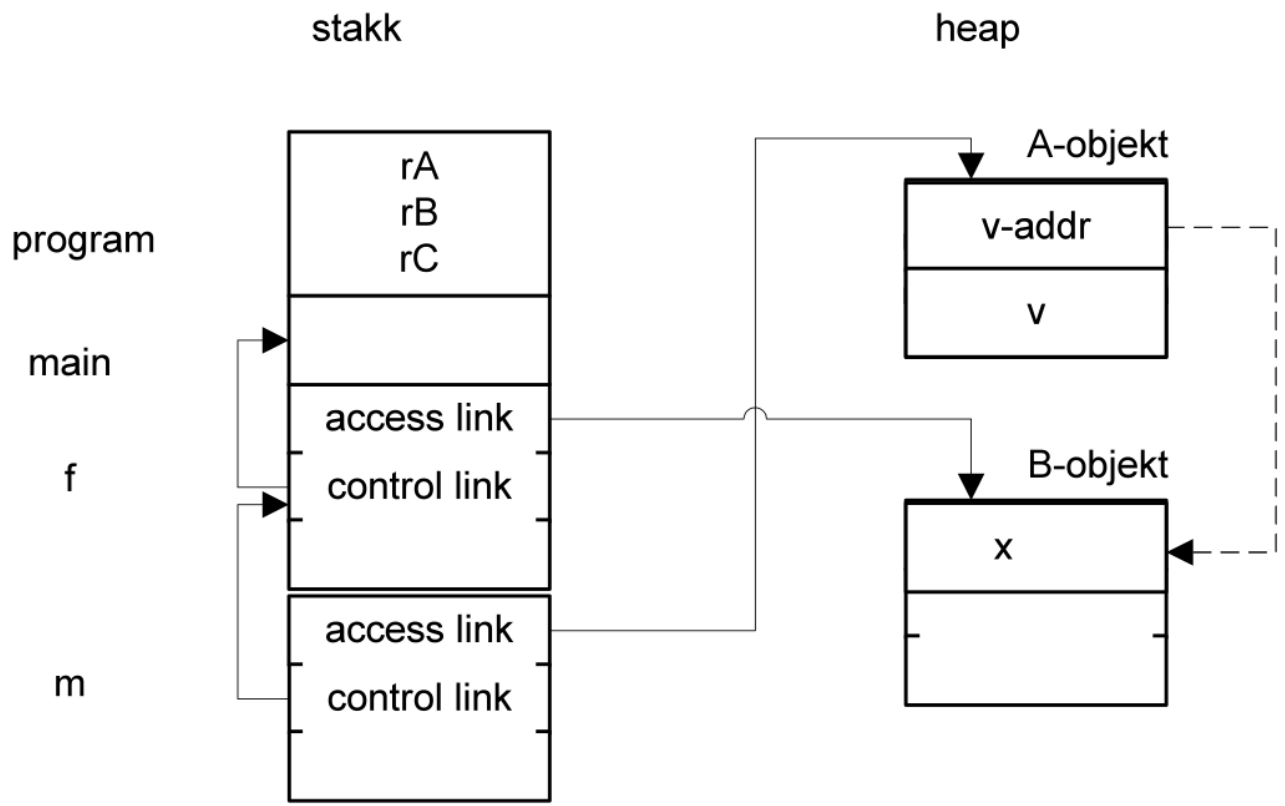
```
{
  class A (val-res int v) {
    void m(){v = v + 1;}
  };
  A rA;
  B rB;
  C rC;
  class B {
    int x = 1;
    void f() {
      rA = new A(x);
      rA.m();
    };
  }
```

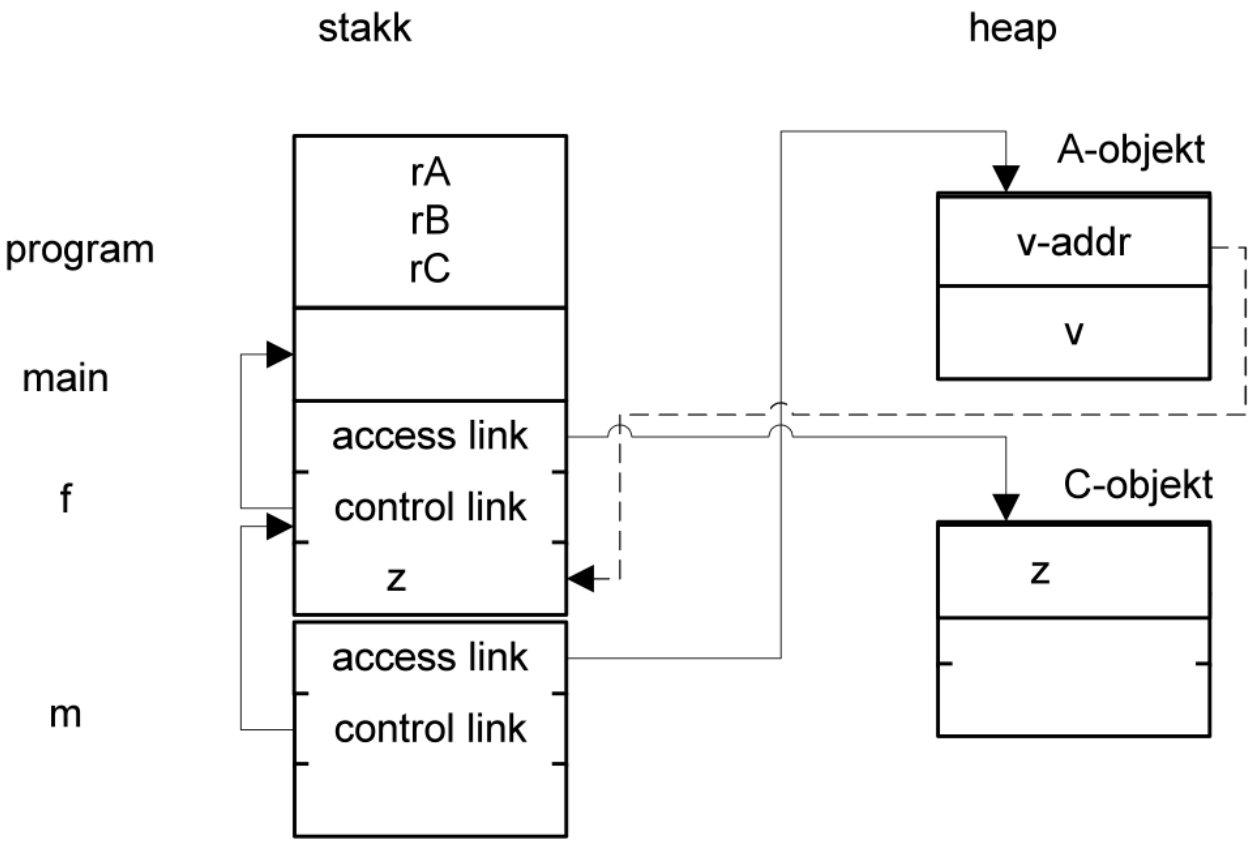
```
class C {
    int z = 1;
    void f() {
      int z = 3;
      rA = new A(z);
      rA.m();
    };
  }
  void main() {
    rB = new B(); rB.f();      // 1
    rC = new C(); rC.f();      // 2
  }
}
```

Tegn stakkene som de ser ut når aktiveringsblokken for m er på toppen av stakken for både //1 og //2, dvs på det tidspunkt hvor den ene setning i m er utført men før exit av m.

Antyd hvordan du ville implementere ’by-value-result’ parametre for klasser ved også å tegne inn objektene av klassene A, B og C. Språket er statisk skopet.

Besvar spørsmålet ved å gjøre figurene på side 6 ferdige, dvs skissér hvordan parameteren v representeres, og fyll inn manglende linker i figurene.


stakk
heap
program
main
f
m
rA
rB
rC
access link
control link
access link
control link
A-objekt
v-addr
v
B-objekt
x

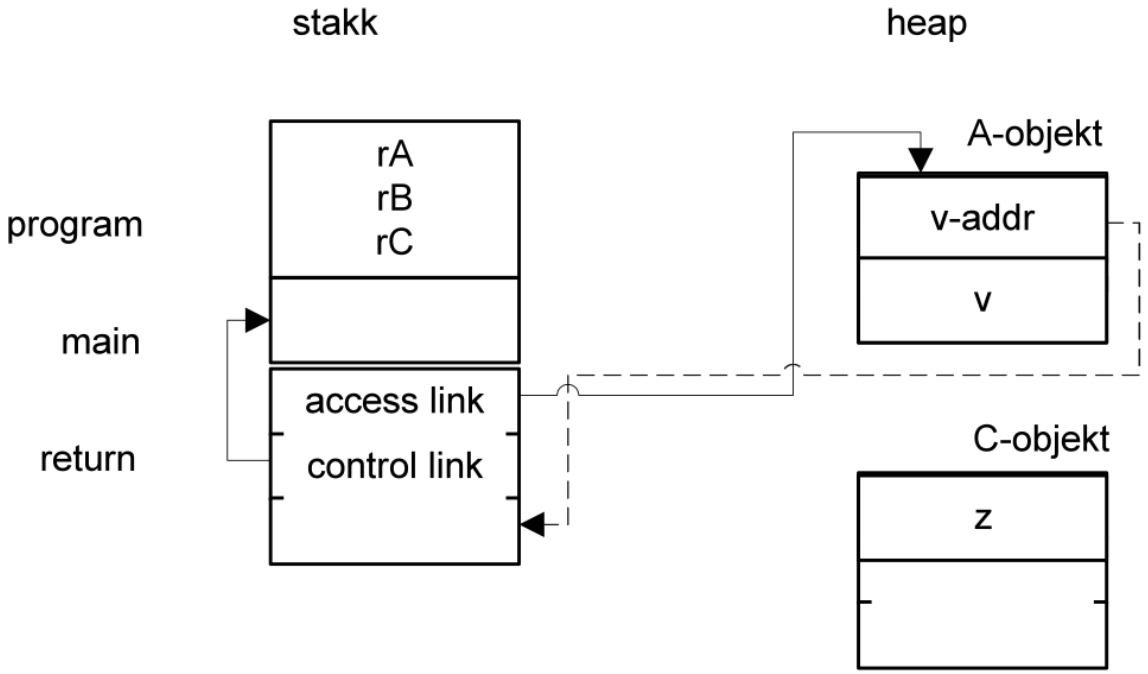

stakk
heap
program
main
f
m
rA
rB
rC
access link
control link
z
access link
control link
A-objekt
v-addr
v
C-objekt
z

2b

Designeren av språket i **2.a** vil gjerne bruke stakk-organisering av metodekall.
I hvilket av tilfellene //1 og //2 går dette galt, og hvorfor? Som en del av svaret kan du eventuelt tegne stakkene som de ser ut når return er på toppen av stakken.

Svar: I //2 overføres addressen til variablen z, som er en del av aktiveringsblokken for kallet på rC.f og som derfor ikke er der når v engang skal returneres.

//2



3a

class → **class** ***id*** *signatures*
class → **class** ***id*** *signatures bodies*
class → **class** ***id*** *class-heading signatures*
class → **class** ***id*** *class-heading signatures bodies*

class-heading → *extends* | *implements* | *extends implements* | *e*
extends → **extends** ***id***
implements → **implements** ***id***

Språket er spesielt på den måten at selv om det bare finnes klasser, så defineres interfacer som klasser, der det ikke er implementasjoner av noen av metodene. Av denne grunn er definisjonen av en metode delt opp i en signatur (navn, type, og parametere) og i en body. Tilsvarende er en klasse definert ved et sett med signaturer og eventuelt ved et sett av bodies. En klasse som bare inneholder et sett av signaturer er altså en interface. En klasse med bodies må ha implementasjoner av alle metoder.

I en klasse kan man i tillegg til metoder (ved signaturer eller ved signaturer samt bodies) også spesifisere (ved en class-heading) at klassen extender en annen klasse og/eller implementerer en annen klasse.

Det er to regler i dette språket:

1. I en klasse kan det etter **extends** bare følge et klasse-navn, og etter ***implements*** bare følge et interface-navn
2. I en interface kan det etter **extends** bare følge et interface-navn, og det skal ikke være noen **implemets**-del

Fyll ut de tomme felter (markert med *) i attributtgrammatikken på side 7 slik at attributtet *ok* for *class* er *true* hvis en klassedefinisjon er gjort ifølge disse regler, ellers *false*. Det skal ikke testes om en klasse faktisk har implementasjon av alle metoder i *signatures*.

Du kan anta at det finnes en insert (***id***, kind) som legger inn navnet ***id*** med egenskapen kind i symboltabellen. Du kan også anta at at lookup-kind(***id***.*name)* leverer den *kind*, som er lagt inn ved hjelp av insert.

Grammar Rule	Semantic Rule
class → ***class id*** *class-heading signatures*	*insert(**id**,* interface*)* *class-heading.kind=interface* *class.OK = class-heading.OK*
class → ***class id*** *class-heading signatures bodies*	*insert(**id**,* class*)* *class-heading.kind=class* *class.OK = class-heading.OK*
class → ***class id*** *signatures*	*insert(**id**,* interface*)* *class.OK = true*
class → ***class id*** *signatures bodies*	*insert(**id**,* class*)* *class.OK = true*
class-heading → *ε*	*class-heading.OK = true*
class-heading → *extends*	*class-heading.OK =* ***if*** *class-heading.kind = class* ***then*** *extends.kind =* class ***else if*** *class-heading.kind = interface* ***then*** *extends.kind =* interface

class-heading → *implements*	*class-heading.OK* =
	if *class-heading.kind* = *class* ***then***
	  implements.kind = interface
	else if
	class-heading.kind = *interface* ***then***
	  false

class-heading →
  extends implements
	class-heading.OK =
	if *class-heading.kind* = *class* ***then***
	  extends.kind = class
	  and
	  implements.kind = interface
	else if
	class-heading.kind = *interface* ***then***
	  false

extends → ***extends id***	*extends.kind* = lookup(***id.****name*)
implements → ***implements id***	*implements.kind* = lookup(***id.****name*)

Eksamen 2008 - 2a

Det følgende er et program i et språk med funksjoner. Språket er statisk skopet. I programmet under er det en funksjon f med to formelle parametre x,y. I main kalles f med a og a som aktuelle parametre.

```
{
  int i = 0;
  int a = 0;
  void f(int x, int y) {
    x = x + 1;
    i = i + 1;
    y = x + i;
  };
  void main(){ f(a,a)}
}
```

	a
By-reference	2
By-value/result	2

Programmet startes ved å kalle funksjonen main.

Anta vi kjører dette program med to forskellige former for parameteroverføring for x og y: 'by-reference' og 'by-value/result'. Hva er verdiene av a etter utførelsen av f i de to tilfellene? Rekkefølgen av parameteroverføring både ved kall og ved exit er fra venstre mot høyre, og ved 'by value/result' beregnes adressen til returverdien ved exit og ikke ved kallet.

Eksamen 2008 – 2b

Det følgende er den interessante del av grammatikken for et språk som ligner på språket over, men funksjoner kan ha maksimalt én parameter. Parameteroverføring er 'by-reference'. Språket har også klasser, men grammatikken for disse er ikke viktig her.

```
decls →           decls ; decl | decl
decl →            var-decl | function-decl
var-decl →        type id = expression
function-decl →   type id ( parameter ? ) body
type →            int | bool | void
parameter →       type id
call →            id ( expression ?)
expression →      id | reference-expr.id |
                  arith-expr | bool-expr
```

Eksamen 2008 – 2b

Den enkle regelen i dette språket er at da parameteroverføring er 'by-reference', så må en aktuell parameter være enten variabel (dvs en ***id***), eller et 'remote expression', som også viser til en variabel (men som en del av et objekt). Typen til den aktuelle parameteren skal være samme type som den formelle, men dette skal *ikke* uttrykkes i attributgrammatikken. En funksjon uten parameter må kalles uten aktuell parameter.

Fyll ut de tomme felter (markert med *) i følgende attributtgrammatikk slik at attributtet *ok* for *call* er *true* hvis kallet er gjort ifølge denne regelen (bortsett fra at typene stemmer overens), ellers *false*.

Du kan anta at det finnes semantiske regler som legger navn inn i symboltabellen. Du kan også anta at at lookup(***id***.*name)*.has_parameter gir verdien *true* eller *false* for en funksjon (med navnet ***id***.*name*) avhengig av om funksjonen har en parameter eller ikke. Det er ikke behov for å sjekke om funksjonsnavnet (***id***) i en *call*-setning faktisk er deklarert.

Grammar Rule	Semantic Rule
function-decl → *type* ***id*** **(** **)** *body*	*insert(**id**,has_parameter=false)*
function-decl → *type* ***id*** **(** *parameter* **)** *body*	*insert(**id**,has_parameter=true)*
call → ***id*** *()*	*call.ok =* *not* lookup *(**id**.name)*.has_parameter
call → ***id*** *(expression)*	*call.ok =* lookup(***id***.*name)*.has_parameter *and* *expression.kind = var*
expression → ***id***	*expression.kind = var*
expression → *reference-expression.**id***	*expression.kind = var*
expression → arith-expr	*expression.kind = expression*
expression → bool-exp	*expression.kind = expression*

Eksamen 2008 – 3

For spørsmålene **3.a** – **3.c** utvider vi programmet fra oppgave 2 med klassene C og C1. Main vil i de tre delspørsmålene være forskjellige. Vi antar at parameteroverføring for f er 'by-reference'.

```
{ int i = 0; int a = 0;
  class C {
    int j;
    void C(int p){j = p;};
  };
  class C1 extends C {int k;};
  C rC= new C(1);
  C anotherC = new C(2);
  C1 rC1 = new C1();
  void f(int x, int y) {
    x = x + 1;
    i = i + 1;
    y = x + i;
    rC= anotherC;
  };
  void main() { ...}
}
```

3a

I dette delspørsmålet ser main slik ut, dvs f kalles med rC.j som aktuell parameter for x og y:

```
void main() {
  f(rC.j,rC.j);
}
```

Hvordan vil du representere parametrene x og y gitt at objekter kan flyttes som en del av en garbage collector. Illustrér eventuelt med en figur som viser en aktiveringsblokk for f med x og y, og et objekt av klassen C.

Svar: Ved en peker + offset/relativ addr

3b

I dette delspørsmålet ser main slik ut:

```
void main() {
  rC= rC1;
  f(rC.j,rC.k);
}
```

Det er jo slik at et objekt av klassen C1 har variablen k, og det er også slik at referansen rC på et gitt tidspunkt kan referere til et C1 objekt, som f.eks. hvis main er som ovenfor.

Hva er det som gjør at den aktuelle parameteren rC.k ikke er OK? k er jo åpenbart en variabel, og på runtime vil rC faktisk peke på et C1 objekt, og det er jo ved runtime parameteroverføringen foregår. Hva er da grunnen til at dette ikke går??

Svar: Klassen C har ikke noen variabel k, så kompilatoren kan ikke oversette Rc.k til peker+offsett.

3d

I denne delen av oppgaven tenker vi oss at ikke bare funksjoner, men også klasser kan ha ’by-reference’ parametre. De spesifiseres som en del av klassedefinisjonen (’ref int cp)’, og det finnes ingen constructor for slike. Parametre kan brukes innen klassen som om de var vanlige variable. Ved generering av objekter overføres adressen til den aktuelle parameteren.

I følgende program finnes det tre forskjellige tilfeller av aktuell parameter til en klasse med en ’by-reference’ parameter:

1. den globale variabel i, som vil være der under hele eksekveringen;
2. den lokale variablen x, og
3. variablen j i et objekt av klassen A

```
{
  int i = 0;
  class A {
    int j;
  };
  class B (ref int cp) {
    void m(){cp = cp + 1;}
  };
  A rA;
  B rB;
```

```
void f() {
    int x;
    x = x + 1;
    rB = new B(x);
  };
  void g() {
    int y;
    y = y + 1;
    rA = new A();
    rB = new B(rA.j);
  };
  void main() {
    rB = new B(i); rB.m();
    f(); rB.m();
    g(); rB.m()
  }
}
```

Designeren av dette spesielle språket vil gjerne bruke stakk-organisering av funksjonskall. Hvilket tilfelle (eller tilfeller) av aktuelle parametre bør da ikke tillates, og hvorfor?

Svar: Tilfelle 2

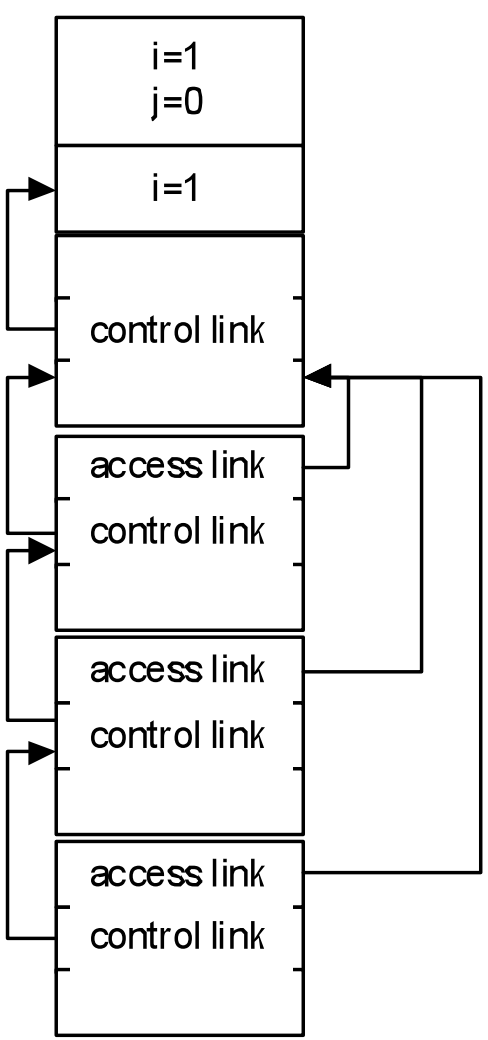
Eksamen 2007 - 2a

Det følgende er et program i et språk som tillater funksjoner inne i funksjoner. Språket er statisk skopet og tillater at en funksjon kan ha én funksjon som parameter. I programmet under har funksjonen f en formell parameter fp, og f kalles med funksjonen g1 som aktuell parameter.

```
{
  int i =0; int j = 0;
  void p() {
    void g1() {g2()};
    void g2() {i=i+1};
    void f(void func fp){ fp() };
    f(g1)
  }
  void main() { int i=1; p()}
}
```

Programmet startes ved å kalle funksjonen main. Vis stakken med statisk link (access link) og dynamisk link (control link) for alle aktiveringsblokker (unntatt for main) som de vil se ut rett etter at setningen i=i+1 er utført. main og p trenger ikke statisk link, da de er definert ytterst. Sett også inn verdier for i og j.

svar



program

main

p

f

g1

g2

2b

Det følgende er et fragment (dvs at ikke-interessante deler av grammatikken ikke er tatt med) av en grammatikk for dette språket.

```
decls →           decls ; decl | decl
decl →            var-decl | function-decl
var-decl →        type id = expression
function-decl →   type id ( parameter? ) body
type →            int | bool | void
parameter →       type id | type func id
call →            id ( id? )
```

Ord i *kursiv* er ikke-terminaler, ord og tegn i **fet** skrift er terminal-symboler. ***id*** representerer et navn.

En parameter er enten en verdi overført 'by value' eller en funksjon uten parameter. Den enkle reglen i dette språket er at en funksjon med en 'by value'-parameter bare kan kalles med en variabel som aktuell parameter (altså ikke med et generelt uttrykk), mens en funksjon med en funksjonsparameter bare kan kalles med en aktuell parameter som er en funksjon uten parametere. Typen til den aktuelle parameteren skal i begge tilfelle være samme type som den formelle. En funksjon uten parameter må kalles uten aktuell parameter.

2b

Fyll ut de tomme felter i følgende attributtgrammatikk slik at attributtet *ok* for *call* er *true* hvis kallet er gjort ifølge disse regler, ellers *false*.

Du kan anta at det finnes semantiske regler som legger navn inn i symboltabellen. Du kan også anta at *lookup(**id**.name).kind* gir verdien 'var' for en variabel og 'func' for en funksjon, *lookup(**id**.name).type* er typen til det som ***id**.name* er navnet på (funksjon, variabel eller parameter) og at lookup(***id**.name).*has_parameter gir verdien '*yes*' eller '*no*' for en funksjon (med navnet ***id**.name*) avhengig av om funksjonen har en parameter eller ikke.

Det er ikke behov for å sjekke om funksjonsnavnet (***id***) i en *call*-setning faktisk er deklarert (du kan altså anta at det allerede er gjort ved andre mekanismer).

Grammar Rule	Semantic Rule
function-decl → *type* **id** **(** **)** *body*	*insert(***id***,has_parameter=no)*
function-decl → *type* **id** **(** *parameter* **)** *body*	*insert(***id***,has_parameter=yes)*
parameter → *type* **id**	*insert(***id***, param_kind = var, param_type = type.*type *)*
parameter → *type* **func** **id**	*insert(***id***, param_kind = func, param_type = type.*type*)*
type → **int**	*type.*type = integer
type → **bool**	*type.*type = boolean
type → **void**	*type.*type = void
call → **id** *()*	*call.*ok = *(lookup(***id***.name).has_parameter=no)*

Grammar Rule	Semantic Rule
call → ***id*$_1$**(***id*$_2$**)	*call.ok =* *(lookup(**id**$_1$.name).has_parameter=yes)* *and* *(lookup(**id**$_2$.name).kind=* *(lookup(**id**$_1$.name).param_kind)* *and* *(lookup(**id**$_2$.name).param_kind=func and* *(lookup(**id**$_2$.name).has_parameter=no)* *and* *(lookup(**id**$_2$.name).type=* *(lookup(**id**$_1$.name).param_type)*

3a

Vi vil her utvide språket i Oppgave 2 med klasser, som også kan defineres inne i funksjoner. Disse klassene har ikke konstruktører, men har parametere som spesifiseres i klassens hode, på samme måte som parametre til funksjoner. Disse parameterene kan sees direkte innenfra klassen. Den aktuelle parameteren gies på vanlig måte når man genererer et objekt (altså, ved `new C (...)`).

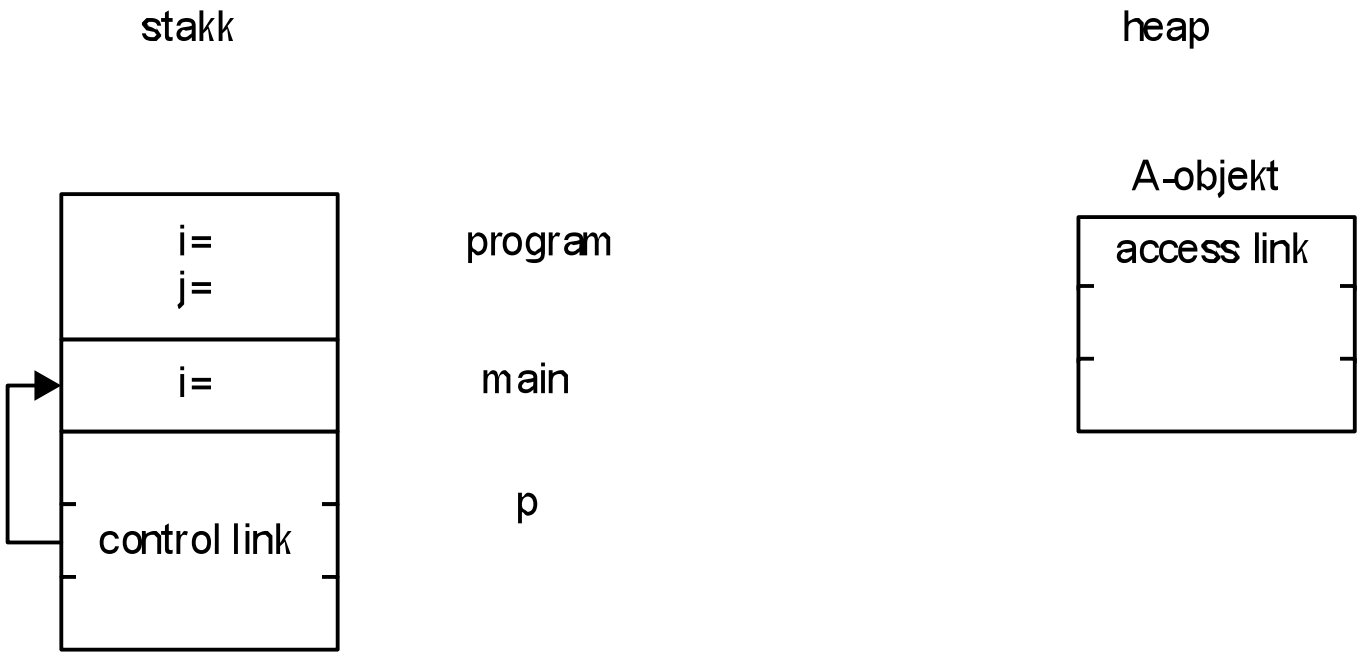
Som for funksjoner tillater vi også at parameteren kan være en funksjon, og den aktuelle funksjons-parameteren til et objekt kan altså kalles fra en metode i objektet på vanlig måte (ved hjelp av det formelle parameternavnet).

I denne del av oppgave 3 er reglen at en aktuell funksjons-parameter (f.eks. `aktfunk` i "`new C (aktfunk)`") må være definert slik at den er direkte synlig fra klassen `C`. Følgende program er laget i henhold til denne reglen:

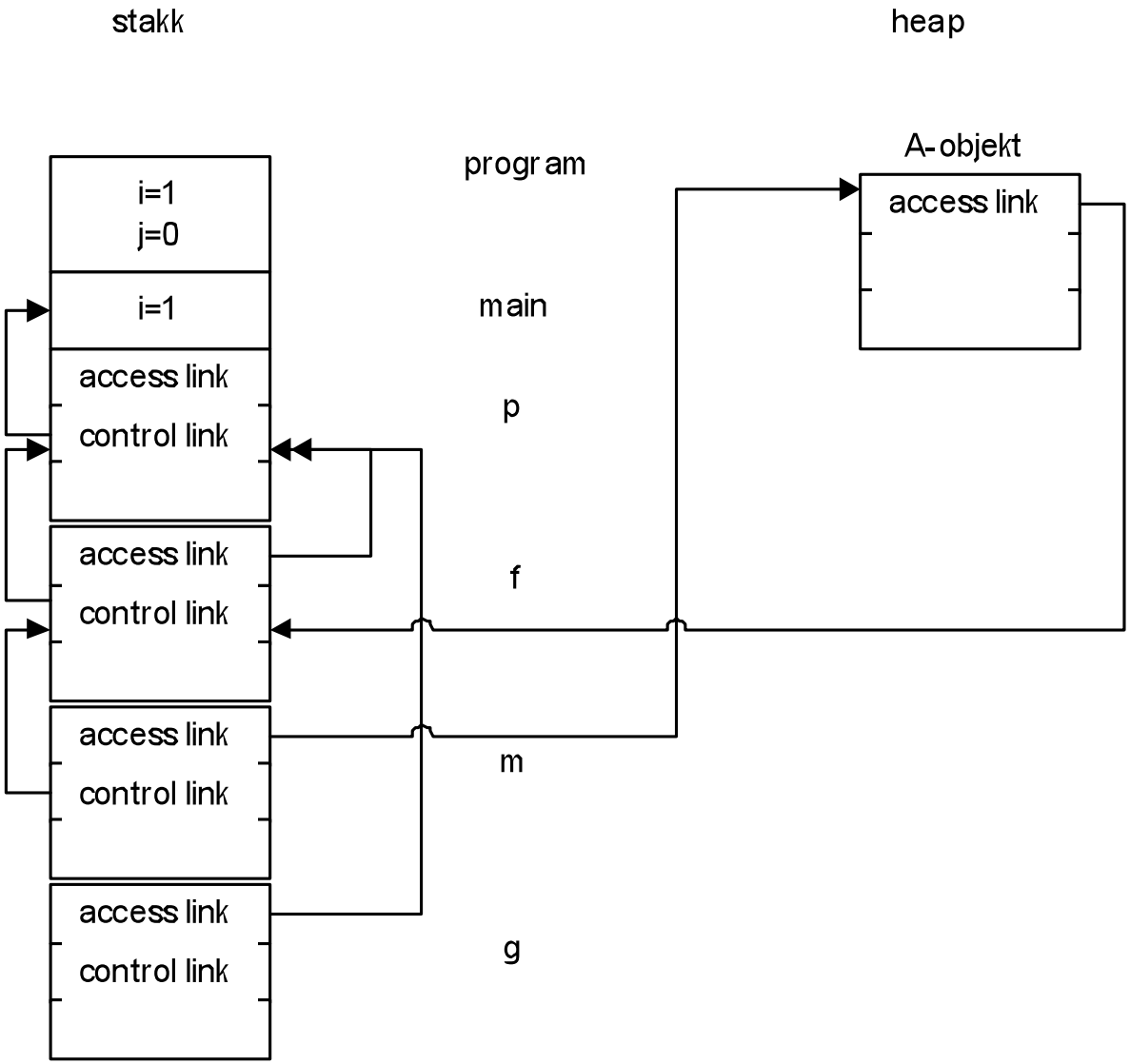
```
{ int i =0; int j = 0;
  void p() {
    void g() {};
    void f(){
      class A(void func fp){
        void m() {fp()};
      };
      A refA = new A(g);
      refA.m();
    };
    f();   // body of p
  };
  void main() { int i=1; p();}
}
```

3a

Vi vil implementere dette slik at også klasse-objekter har en statisk link (access link), og at den peker til aktiveringsblokken for den programblokk, hvor klassen er definert. Videre vil vi at aktiveringsblokker for lokale metoder i en klasse skal ha sine statiske link til det objektet som har metoden. Tegn stakken og det involverte objektet som de er mens m utfører fp. Vi tenker oss som vanlig at aktiveringsblokker for funksjoner legges på stakken, og at objekter legges på heapen. Tegn inn statiske link (access links) og dynamiske link (control links). Oppgaven besvares ved å fylle ut vedlegg side 9.



3a svar


stakk
heap
program
A-objekt
access link
i=1
j=0
i=1
main
access link
control link
p
access link
control link
f
access link
control link
m
access link
control link
g

3b

I denne del av oppgaven forsøker vi å tillate at aktuelle parametre til klasser kan være funksjoner som er synlige i det skop (og omsluttende) hvor new utføres. Følgende program gjør dette:

Forklar hvorfor dette programmet vil gå galt med vanlig stakkorganisering av aktiveringsblokkene for funksjoner. Skisser en simulering av programmet, og angi når det går galt.

```
{ int i =0; int j = 0;
  void p() {
    void g() {};
    void f() {
      class A(void func fp) {
        void m() { fp() };
      };
      A refA;
      h() {
        int i;
        void ff() {i=5};
        refA = new A(ff);
                // ff er ikke direkte synlig
                // fra klassen A
      };
      h();
      refA.m();
   };
   f();  // body of p
  };     // end of p
   void main() { int i=1; p() }
}
```

Svar: Aktiveringsblokken for kallet på h() er ikke på stakken når m forsøker å kalle ff (via kallet på fp)

Eksamen 2006 - I

```
procedure → proc id (param) stmt
param → type id | ref type id | result type id
call → id(exp)
exp → id
exp → id[exp]
exp → exp aritop exp
```

`1c`

Den enkle reglen i dette språket er at prosedyrer med en parameter 'by reference' eller 'by-value-result' bare kan kalles med et uttrykk som enten er en enkel variabel (*id*) eller en indisert variabel (*id*[*exp*]).

Fyll ut de tomme felter i følgende attributtgrammatikk slik at attributtet *ok* for *call* er *true* hvis kallet er gjort ifølge denne regel, ellers *false*.

Symboltabellen er innrettet på akkurat denne reglen, slik at navn på prosedyrer er assosiert med en verdi som sier om denne prosedyre har en parameter 'by reference' (verdien *ref*), 'by value-result' (verdien *result*) eller 'by value' (*value*). *lookupkind*(*id.name*) gir verdien som svarer til hvordan prosedyren med navn *id.name* er definert. Det er ikke behov for å sjekke om prosedyrenavnet (*id*) i en *call*-setning faktisk er deklarert.

Grammar Rule	Semantic Rule
procedure → **proc** ***id*** (*param*) *stmt*	*insert*(***id***.*name*, *param.kind*)
param → *type* ***id***	
param → **ref** *type* ***id***	
param → **result** *type* ***id***	
call → ***id*** (*exp*)	*call.ok* =
exp_1 → ***id*** [exp_2]	
exp → ***id***	
exp_1 → exp_2 *aritop* exp_3	

Grammar Rule	Semantic Rule
procedure → **proc** ***id*** **(***param***)** *stmt*	*insert(***id***.name, param.kind)*
param → *type* ***id***	*param.kind = value*
param → **ref** *type* ***id***	*param.kind = ref*
param → **result** *type* ***id***	*param.kind = result*
call → ***id*** **(***exp***)**	*call.ok =* ***if*** *(lookupkind(***id***.name)= ref* ***or*** *lookupkind(***id***.name)= result)* ***then*** *(exp.kind = var)* ***else*** *true*
exp_1 → ***id*** **[**exp_2**]**	exp_1*.kind = var*
exp → ***id***	*exp.kind = var*
exp_1 → exp_2 *aritop* exp_3	exp_1*.kind = value*