UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i :	INF5110
Eksamensdag :	Tirsdag 5. juni 2007
Tid for eksamen :	14.30 - 17.30
Oppgavesettet er på :	6 sider (pluss vedlegg)
Vedlegg :	3 sider (side 7, 8 og 9). Disse rives ut, fylles ut og leveres
Tillatte hjelpemidler :	Alle trykte og skrevne

Les gjennom ***hele*** oppgavesettet før du begynner å løse den første oppgaven. Dersom du savner opplysninger i oppgaven, kan du selv legge dine egne forutsetninger til grunn og gjøre rimelige antagelser, så lenge de ikke bryter med oppgavens "ånd". Gjør i så tilfelle rede for disse forutsetningene og antagelsene.

Oppgave 1 (30%)

Gitt følgende grammatikk G1:

A → x | B C
B → x
C → B | y | C y

Her er A, B og C ikke-terminaler, A er startsymbol, og x og y er terminalsymboler.

1.a
Tegn LR(0)-DFA'en til G1 (etter å ha lagt på "A' → A" på vanlig måte)

1.b
Beregn First- og Follow-mengdene til A, B og C. Det holder å angi resultetet

1.c
Avgjør om grammatikken G1 er SLR(1). Forklar.

1.d
Vi forandrer litt på G1, og ser nå på følgende grammatikk G2:

A → x | B C
B → x
C → B | y | C x

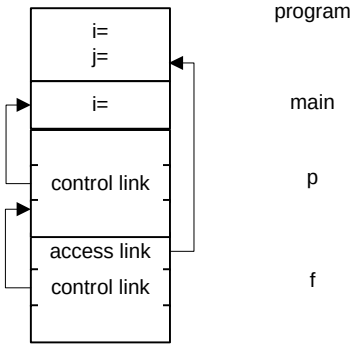
Avgjør om G2 er LALR(1). Forklar.

Oppgave 2 (26%)

2.a
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`. Oppgaven besvares ved å ta utgangspunkt i stakken under (stakken vokser nedover). Besvar oppgaven ved å bruke vedlegg side 7.



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.

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*. Besvar oppgaven ved å bruke vedlegg side 8.

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*	*function-decl.has_parameter = no*
function-decl → *type* ***id*** **(** *parameter* **)** *body*	*function-decl.has_parameter = yes* *function-decl.param-kind =* *parameter.kind* *function-decl.param-type =* *parameter.type*
parameter → type ***id***	
parameter → type **func** ***id***	
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)*
call → $\boldsymbol{id_1}$*(*$\boldsymbol{id_2}$*)*	*call.ok =*

Oppgave 3 (24%)

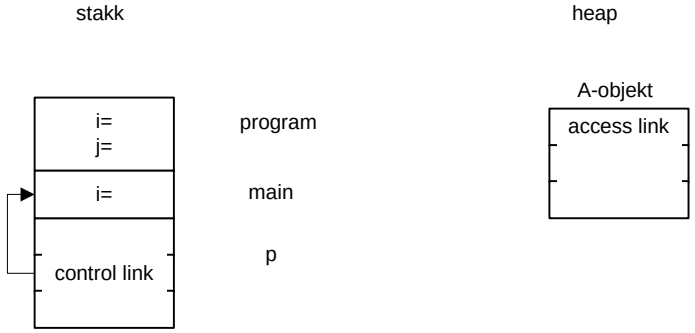
3a
Vi vil her utvide språket i Oppgave 2 med klasser, som også kan defineres inne i funksjoner. Disse klassene har ikke konstruktører, men har parametere som spesifiseres i klassens hode, på samme måte som parametre til funksjoner. Disse parameterene kan sees direkte innenfra klassen. Den aktuelle parameteren 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();}
}
```

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.



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:

```
{ 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() }
}
```

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.

Oppgave 4 (20 %)

Gitt følgende program, der alle setningene er tre-addresse-instruksjoner, bortsett fra at vi også tillater if- og while-setninger på vanlig måte. Instruksjonene ”x = input” og ”output x” regnes som vanlige tre-addresse-instruksjoner, med den opplagte betydning.

```
 1:    a = input
 2:    b = input
 3:    d = a + b
 4:    c = a * d
 5:    if ( b < 5 ) {
 6:        while ( b < 0 ) {
 7:            a = b + 2
 8:            b = b + 1
 9:        }
10:        d = 2 * b
11:    } else {
12:        d = b * 3
13:        a = d - b
14:    }
15:    output a
16:    output d
```

Vi skal se på hvilke variable som er ”i live” (”live”) et gitt sted i dette programmet, slik dette begrepet er definert (med rettinger) i det utdelte notat fra boka til Aho, Sethi og Ullmann (ASU-boka) og på lysarkene. Vi antar at ingen variable er i live ved slutten av programmet.

4a
Angi for hver av variablene *a*, *b*, *c* og *d* om de er i live eller ikke *umiddelbart etter* linje 4. Gi en kort forklaring for hver av variablene.

4b
Forklar hva man i kodegenererings-algoritmen i kap 9.6 (i den utdelte kopien fra ASU-boka) mener med at en variabel har en *neste bruk* (”next use”) og hvordan dette skiller seg fra å være *i live*. Forklar hvorfor denne algoritmen skiller mellom disse to begrepene.

Slutt på oppgavesettet

Lykke til!

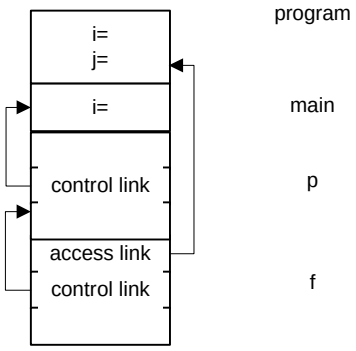
Stein Krogdahl og Birger Møller-Pedersen

Vedlegg til besvarelse av Oppgave 2

Kandidat nr: .....

Dato: .....

2a



Vedlegg til besvarelse av Oppgave 2 Kandidat nr: .....

Dato: .....

2b

Grammar Rule	Semantic Rule
function-decl → *type* ***id*** **(** **)** *body*	*function-decl.has_parameter = no*
function-decl → *type* ***id*** **(** *parameter* **)** *body*	*function-decl.has_parameter = yes* *function-decl.param-kind = parameter.kind* *function-decl.param-type = parameter.type*
parameter → *type* ***id***	
parameter → *type* **func** ***id***	
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)*
call → id_1*(*id_2*)*	*call.ok =*

Vedlegg til besvarelse av Oppgave 3

Kandidat nr: .....

Dato: .....

3a