UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i :	INF5110 - Kompilatorteknikk
Eksamensdag :	Onsdag 2. juni 2010
Tid for eksamen :	14.30 - 17.30
Oppgavesettet er på :	5 sider (pluss vedlegg)
Vedlegg :	1 side (side 6 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. Deler av oppgavene 2 og 3 besvares ved bruk av vedlegg.

Oppgave 1 (25%)

Vi skal se på grammatikker for uttrykk som kan inneholde såkalte ”unære minuser”, altså at man kan skrive uttrykk som ”$– x * y + z$”. Man kan imidlertid tolke dette uttrykket på forskjellige måte, f.eks. på en av de to følgende:

(a) $(– (x * y)) + z$
(b) $((– x) * y) + z$

1a

Vi ser på følgende grammatikk G1:

E → U | U + V
U → T | -T
V → V + T | T
T → T * F | F
F → (E) | **name**

Her er E, U, V, T, og F ikke-terminaler, E er startsymbol, og resten er terminal-symboler.

Avgjør hvilken av de to tolkningen (a) eller (b) denne grammatikken vil gi av ”$– x * y + z$”, og tegn et parserings-tre (altså *ikke* bare et *abstrakt* syntaks-tre) som viser at grammatikken kan produsere denne setningen med den tolkningen du har valgt.

1b

Legg på produksjonen ”E’→ E”. Tegn så starttilstanden av LR(0)-DFA’en til G1, samt de tilstandene du får om du går ett hakk ut fra denne.

1c
Grammatikken G1 er opplagt ikke LL(1). Din oppgave her er å skrive om grammatikken slik at det ikke lenger finnes opplagte tegn på at grammatikken ikke er LL(1). Du behøver ikke sjekke om den resulterende grammatikken faktisk er LL(1).

1d
Finn en grammatikk G2 som gir den motsatte tolkningen ((a) eller (b)) av den du anga i oppgave **1a**. Merk at G2 skal tolke alle tradisjonelle uttrykk som *ikke* har noen unær minus (men har + og *) på vanlig måte. Forsøk å lage G2 slik at den ikke tillater flere unære minuser etter hverandre, altså slik at "- - *x*" er ulovlig, mens " - *(- x)*" er lovlig.

Oppgave 2 (20%)

I denne oppgaven har vi et helt vanlig objekt-orientert språk, hvor en virtuell metode i en klasse kan redefineres i en subklasse av denne klassen. En virtuell metode deklareres med en `virtual` modifier, mens en redefinisjon deklareres med modifieren `redef`. Når vi kaller en virtuell metode for et gitt objekt vil klassen til objektet bestemme på vanlig måte hvilken metode som kalles. I motsetning til i Java finnes det altså metoder som ikke er virtuelle.

Det følgende er klasser definert i dette språk:

```
class A {
  virtual void m(){...}
  void p(){...}
  virtual void q(){...}

}
class B extends A{
  redef void q(){...}
  void r(){...}
}
class C extends B{
  redef void m(){...}
}
class D extends C{
  redef void m(){...}
  redef void q(){...}
}
```

2a
Lag virtuell-tabellene for klassene `A`, `B`, `C` og `D`. For hvert element i tabellene skal du bruke notasjonen `A::m` for å angi hvilken metode som gjelder. Indeksen i disse tabeller starter på 1.

2b
I denne del av oppgaven skal vi se på hvordan man kan implementere en alternativ redefinisjons-regel:

> For et objekt av en klasse `K`, som har en virtuell metode `vm`, vil et kall på denne metoden virke som følger:
>
> - hvis `K` ikke har noen superklasse, så skal det bli et kall på `K::vm`
> - hvis `K` har en eller en sekvens av superklasser, så skal vi få en *sekvens* av kall bestående av: først et kall på `vm` i den superklassklassen der den introduseres som virtuell, så kall på `vm` i de superklasser hvor `vm` redefineres (i en sekvens som følger subklasse-hierarkiet), eventuelt avsluttet med av et kall på `vm` i `K` (hvis redefinert i `K`).

For eksemplet i **2a** vil et kall på `q()` for et objekt av klassen `D` føre til følgende kall-sekvens: `'A::q(); B::q(); D::q()'`, altså ingen `C::q()`, da `q` ikke er redefinert i klassen `C`.

Vi antar at virtuelle metoder kan ha hverken parametere eller returverdi.

Til å implementere dette innføres det for hver klasse en virtuell-tabell, hvor det for hver virtuell metode er en rad med lengde lik 1 + <antall superklasser inntil den klasse som introduserer den virtuelle> , og hvor et element i denne raden angir en eventuelt redefinert metode. Indeksen for hver rad starter på 0. Elementet med indeks 0 i hver rad angir den (eventuelle) redefinisjon for klassen selv, elementet med indeks 1 angir den (eventuelle) redefinisjon for superklassen, osv. Hver rad vil dermed angi sekvensen av de metoder som skal kalles.

Tegn de nye virtuell-tabeller for klassene `C` og `D`. Tabellene for `A` og `B` er gitt under. For redefinisjoner brukes samme notasjon som før, mens det for elementer som representerer ingen redefinisjon brukes en '-'.

A

	0
1	A::m
2	A::q

B

	0	1
1	-	A::m
2	B::q	A::q

2c

Vi skal her se på hvordan tabellene fra **2b** kan brukes. Når et kall til en virtuell metode blir gjort går kontrollen til en runtime-rutine som får en peker til riktig rad i riktig virtuell-tabell, og lengden av denne raden. Vi antar at denne rutinen også har tilgang til størrelsen på aktiveringsblokkene for metodene i raden. Denne rutinen skal så legge aktiveringsblokker på stakken slik at den til slutt kan gi kontrollen til metoden på toppen av stakken (fp – frame pointer), og at utførelsen, inklusive termineringen, skal gå videre av seg selv som ved vanlig eksekvering.

Vi skal se på hvordan rutinen da må sette opp stakken, og spesielt hvordan retur-adressene og dynamisk linkene (control link) samt fp skal settes. Du skal vise dette for det tilfellet at vi kaller metoden `m` for et objekt av klassen `D` (anta at kallet på `m` foregår i en `main` metode). Angi for hver aktiveringsblokk (activation record) hvilken metode den tilsvarer (på formen klasse::metodenavn) og hvor dynamisk link skal peke (ved en pil). Retur-adressen (return address) kan du angi ved en kort forklarende tekst. Statisk link (access link) trenger du ikke angi.

Hver aktiveringsblokk skal ha form som angitt under, og stakken skal vokse nedover:

Oppgave 3 (30%)

3a
Det følgende er en del av grammatikken for et språk med funksjoner.

```
func → type func id signature stmt-list
type → int
type → bool
stmt-list → stmt-list stmt
stmt-list → stmt
stmt → assign-stmt
stmt → if-stmt
stmt → return-stmt
return-stmt → return exp
exp → id
exp → id + id
exp → true
exp → false
```

Ord i *kursiv* er ikke-terminaler, ord og tegn i **fet** skrift er terminal-symboler. *id* representerer et navn.
En funksjon har en type som enten er Integer eller Boolean. Reglen i dette språket er (ikke overraskende) at return-setningen (**return** exp) skal returnere en verdi med samme type som typen til funksjonen.

Fyll ut de tomme felter (markert med *) i attributtgrammatikken for de relevante deler av grammatikken på side 7 slik at attributtet *ok* for *return-stmt* er `true` hvis typen til retur-uttryket *exp* er det samme som typen til funksjonen, ellers `false`.

Du kan anta at `lookup-kind(`*id.name*`)` leverer den *type*, som er lagt inn i symboltabellen ved deklarasjonen av variablen med navnet *id.name*.

(slutt oppgave 3a)

I de to neste delspørsmål ser vi på parameteroverføring i dette språk. Funksjoner kan ha én parameter, og den er enten ’by value’ eller ’by-value-result’ (keyword **result**). Dette er reflektert i følgende definisjon av *signature*:

```
signature → type id | result type id
```

Følgende program deklarerer en integer array `a` og en funksjon `inc`. Programmet initialiserer `i` og `a`, kaller funksjonen `inc(a[i])` og skriver ut verdien av `a[0], a[1]`. Språket følger vanlige statiske skopregler.

```
{
  int i;
  int[2] a;
  int func inc(result int j) {
    j = j + i;
    i = i + 1;
    return i
  };
  a[0]=1; a[1]=2
  i=0;
  inc(a[i]);
  write(a[0], a[1])
}
```

3b
Anta at semantikken for 'by-value-result' er slik at adressen (location) til den aktuelle parameter beregnes ved funksjonskallet.
Hva skrives ut?

3c
Anta at semantikken for 'by-value-result' er slik at adressen (location) til den aktuelle beregnes ved *avslutning* av funksjonskallet.
Hva skrives ut?

Oppgave 4 (25%)

4a
Kari har en BNF-grammatikk hun mener ikke er LR(1), men hun mener likevel at grammatikken beskriver et regulært språk. Anne mener det ikke går an. Hvem har rett? Forklar.

4b
Per har en BNF-grammatikk han mener er flertydig, men at den likevel er LL(1). Ole mener det ikke går an. Hvem har rett. Forklar.

4c
Nora påstår at når en Java-klasse oversettes til byte-kode (i form av en class-fil) så må kompilatoren ha tak i class-fila til en eventuell superklasse for der står det hvor mye plass (i byte) som trengs til variablene i denne superklassen (og dens superklasser igjen). Frida mener hun må ha misforstått. Hvem mener du har rett? Forklar.

4d
Arne har sett på kodegenerings-algoritemen på slutten av det utdelte heftet (fra kap. 9 i ASU). Han mener da at for de to treadresse-instruksjonene: "t1 = a – b; t2 = b – c;" så vil algoritmen produsere instruksjonene under. Han har antatt at det er to registre, og at begge er tomme ved starten.

```
MOV a, R0
MOV b, R1
SUB  R1, R0
SUB  c, R1
```

Ellen er uenig. Hvem har rett? Forklar.

4e
Anta at vi ser på et språk der kompilatoren har frihet til å oversette uttrykk slik at de forskjellige delene av uttrykket blir beregnet i den rekkefølgen kompilatoren finner best. Videre er det i den maskinen vi oversetter til viktig å holde så mange mellom-verdier som mulig i registre under beregningen av uttrykk. Petter sier da at om man har et stort uttrykk som (f.eks.) er summen av to sub-uttrykk så lønner det seg oftest å generere kode slik at det *største* sub-uttrykket blir beregnet først, mens Unni mener at det er motsatt. Hvem har rett og hvordan bør man definere "størrelsen" av et sub-uttrykk i dette tilfellet?

Lykke til!

Stein Krogdahl og *Birger Møller-Pedersen*

Vedlegg til besvarelse av Oppgave 3

Kandidat nr: .....

Dato: .....

3a

Grammar Rule	Semantic Rule	
func → *type* **func** *id signature* *stmt-list*		*
type → **int**	*type*.type = Integer	
type → **bool**	*type*.type = Boolean	
$stmt\text{-}list_1$ → $stmt\text{-}list_2$ *stmt*		*
stmt-list → *stmt*		*
stmt → *return-stmt*		*
return-stmt → **return** *exp*	*return-stmt*.ok =	
exp → *id*	*exp*.type = lookup(*id*.*name*)	
exp → id_1 **+** id_2		*
exp → **true**	*exp*.type = Boolean	
exp → **false**	*exp*.type = Boolean	