UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i :	INF5110 - Kompilatorteknikk
Eksamensdag :	Onsdag 1. juni 2011
Tid for eksamen :	14.30 - 18.30
Oppgavesettet er på :	7 sider (pluss vedlegg)
Vedlegg :	1 side (side 8 rives ut, fylles ut og leveres i ”hvit” besvarelse)
Tillatte hjelpemidler :	Alle trykte og skrevne

Les gjennom ***hele*** oppgavesettet før du begynner å løse oppgavene. Dersom du savner opplysninger i oppgavene, 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 (25%)

Vi skal se på et antall grammatikker, nemlig følgende:

i. $A \rightarrow b\ A\ c \mid \varepsilon$
ii. $A \rightarrow b\ A\ b \mid b$
iii. $A \rightarrow b\ A\ b \mid c$

1a
Her er A startsymbol og eneste ikke-terminal, mens b og c er terminaler. For hver av de tre grammatikkene: Beregn First og Follow til A, tegn LR(0)-DFA’en (etter utvidelse med A’ → A), og angi om den er en SLR-grammatikk eller ikke. Angi også hvilke av grammatikkene, om noen, som er LR(0)-grammatikker.

1b
For hver av de tre grammatikkene, avgjør om grammatikken er LR(1). Det er her mulig å avgjøre dette og forklare det uten å tegne opp LR(1)-DFA’en, men det er også en OK besvarelse om du tegner opp denne og avgjør det ut fra den.

1c
Angi for hver av grammatikkene over om det språket de genererer er regulært, og for de som er det skal du angi et regulært uttrykk for språket. For de grammatikker du mener *ikke* genererer et regulært språk, forklar hvorfor.

1d
Tegn opp en parseringstabell for grammatikk i, og sørg for at den blir uten konflikter. Angi så en steg-for-steg LR-analyse av setningen ”bbcc”, på samme måte som øverst på side 213 i boka (tabell 5.8).

Oppgave 2 (20%)

Anta at vi har et objekt-orientert språk, hvor en virtuell metode i en klasse kan redefineres (”overriding”) i en subklasse av denne klassen. En virtuell metode deklareres med en `virtual` modifier, mens en redefinisjon deklareres med modifieren `redef`. Metoder uten `virtual` er vanlige metoder og kan altså ikke redefineres. Merk at dette ikke er helt som i Java. I Java er alle metoder virtuelle, mens her gjelder det bare de som har modifieren `virtual`.

Det følgende er klasser definert i dette språket:

```
class A {
  virtual void m(int x,y){...}
  void p(){...}
  virtual void q(){...}
}
class B extends A{
  redef void m(int x,y){...}
  void r(){...}
}
class C extends A{
  redef void q(){...}
}
class D extends B{
  redef void m(int x,y){...}
}
class E extends B{
  redef void q(){...}

class F extends C{
  redef void m(int x,y){...}
```

2a
Vi antar nå først at klassen for et gitt objekt bestemmer, på vanlig måte, hvilken versjon av en virtuell metode som kalles.

Lag virtuell-tabellene for klassene `A`, `B`, `C`, `D,E` og `F`. 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 resten av oppgaven skal vi for virtuelle metoder ha den semantikk at en redefinert virtuell metode, for eksempel `m`, skal, som det første den gjør, kalle den tilsvarende virtuelle eller redefinerte metode (dvs `m`) i den nærmeste superklasse som har en slik, før den utfører sin egen body. Dette vil i sin tur føre til at redefinerte eller virtuelle metoder `m` i videre superklasser utføres.

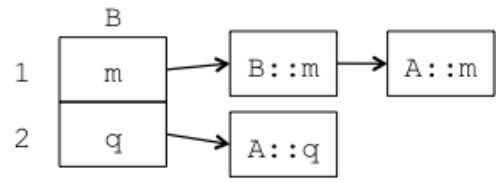
Dette kunne man implementere ved å sette inn det riktige kall som første statement i bodies på redefinerte metoder. Men semantikken rundt parameteroverføring skal her være litt spesiell slik at denne enkle måten ikke vil fungere. Parametrene som gis med i det opprinnelige kallet skal nemlig gå direkte som parametre til den metoden som utføres først, altså den som er merket `virtual` i programmet. Når denne er ferdig utført skal de verdiene som da står i dens parametervariable overføres som aktuelle parametre til den neste dypere redefinerte metode, osv. Dette gjør at stakken av kall må settes opp først, og de aktuelle parametre må gis til den første virtuelle metode som skal

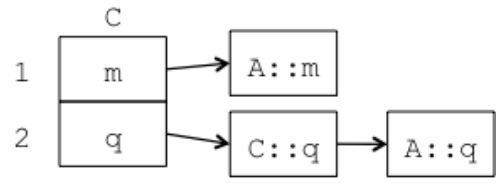
utføres.

Hvis for eksempel `m` kalles med `m(1,2)` på et `D`-objekt, så skal stakken settes opp og de aktuelle parametre gis til aktiveringsblokken tilsvarende `A::m`, og utførelsen skal starte med utførelsen av `A::m`. Ved exit av `A::m` skal verdiene av parametrene `x` og `y` gis som aktuelle til den versjon av `m` som da skal utføres.

For å implementere denne nye semantikk utvides virtuell-tabellen, slik at det for hvert indeks blir en liste av metode-angivelser. Denne listen vil dermed angi sekvensen av de metoder som skal kalles.

Tegn de nye virtuell-tabeller for klassene `D` og `F`. Tabellene for `B` og `C` er gitt under. For metode-angivelser brukes samme notasjon som før.





2c

I denne del av oppgaven skal du skissere hvordan stakken i **2b** kan lages ved hjelp av de nye virtuell-tabeller. Du kan anta at du har en run-time rutine `makeActivationRecord(metode)`. Denne tar som parameter en metode-angivelse (for eksempel `A::m`) med nok informasjon til å lage en aktiveringsblokk med riktig størrelse, men du skal skissere hvordan control-link og retur-adresse settes i aktiveringsblokken.
Anta at den aktuelle virtuell-tabell holdes i en variabel med navn `vt`, den aktuelle metoden holdes i en variabel `method`, og funksjonen `index(method)` gir deg index i `vt`. Anta videre at inngangen i tabellen gir en peker til første metode-angivelse, og at hver av disse har en `next` peker. For metode-angivelsen som svarer til den virtuelle metode hvor den defineres for første gang (i eksemplet her `m` i `A`) er denne peker `none`.

Du kan gjerne illustrere resultatet for et kall `m(1,2)` på et `D`-objekt, men om resten er riktig er det OK uten.

2d

Overføring av parametre mellom de enkelte utførelser av en virtuell metode kan åpenbart ikke gjøres som en del av det å sette opp stakken i **2c**, men må gjøres ved bl.a. å sette inn ekstra kode i metoder merket `virtual` eller `redef`. Du må også innføre en ekstra variabel i de aktuelle aktiveringsblokkene. Hvilken kode skal settes inn og hvor? Koden kan gjøre bruk av alle deler av den aktuelle aktiveringsblokk.

Oppgave 3 (30%)

Det følgende er et fragment av en grammatikk for et språk med klasser. En klasse kan *ikke* ha noen superklasse, men den må implementere en eller flere interfacer:

```
class → class name implements interfaces { decls }
decls → decls ; decl | decl
decl → variable-decl | method-decl
method-decl → type name ( params ) body
type → int | bool | void
interfaces → interfaces, interface | interface
interface → name
```

Ord i kursiv er ikke-terminaler, ord og tegn i **fet skrift** er terminal-symboler, mens ***name*** representerer et navn som scanneren leverer. Det kan antas at ***name*** har attributtet 'name'.

Det som er spesielt med dette språk er at de av en klasses metoder som har samme navn som en av interfacene klassen implementerer er 'konstruktører' for klassen. Det kan i klassen gjerne være flere metoder, som har samme navn som et interface, men da med forskjellige parametre; dette er imidlertid ikke temaet i denne oppgave. Generering av objekter har formen "new <class-name>.<interface-name>(<actual parameters>)", da forskjellige klasser kan implementere samme interfacen.

En krav i dette språket er at konstruktører må være spesifisert med typen void, og det er dette kravet som skal sjekkes med de semantiske regler du skal sette opp.

Lag semantiske regler for dette kravet i følgende fragment av en attributtgrammatikk.
For å lage reglene kan du bruke de funksjoner og de mengder du trenger, bare du definerer dem.

Besvar dette spørsmålet ved å bruke vedlegget side 8.

Grammar Rule	Semantic Rule
class → **class** ***name*** ***implements*** *interfaces* **{** *decls* **}**	
$decls_1$ → $decls_2$ **;** *decl*	
decls → *decl*	
decl → *method-decl*	
method-decl → *type* ***name*** **(** *params* **)** *body*	
type → **int**	*type*.type = int
type → **bool**	*type*.type = bool
type → **void**	*type*.type = void
$interfaces_1$ → $interfaces_2$**,** *interface*	
interfaces → *interface*	
interface → ***name***	*interface*.interfaceName = ***name***

Oppgave 4 (25%)

Vi skal her se på verifikasjon (omtrent som i en Java/JVM-loader) av en enkel type P-kode. Den har få instruksjoner, og alle verdier er heltall. Vår P-kode utføres på vanlig måte, med en stakk med verdier under utførelsen. Under er v en programvariabel, og L er adressen til et sted i programmet. Vår spesielle P-kode har følgende instruksjoner:

lda v Henter adressen til variablen v opp på toppen av stakken. En adresse er også et heltall.
ldv v Henter verdien av variablen v opp på toppen av stakken
ldc k Henter konstanten k opp på stakken
add Legger sammen de to øverste verdier på stakken, fjerner (popper) dem fra stakken og legger svaret på toppen av stakken.
sto Her tolkes det som ligger på toppen av stakken som en verdi, og det nest øverst som en adresse. Instruksjonen kopierer verdien inn til den angitte adressen i lageret, og popper både verdien og adressen.
jmp L Hopp til program-adressen L
jge L (og likeledes: jgt L, jle L, jlt L, jeq L, jne L) Denne instruksjonen er litt enklere enn vanlig, nemlig slik: Om *verdien på toppen* av stakken er større eller lik 0 så hoppes det (og tilsvarende for de andre fem). Verdien på toppen av stakken poppes uansett om det hoppes eller ikke.
lab L Angir at program-adressen L er på dette stedet i programmet.

4a

Vi tenker oss at vi skal lage en verifikator for programmer i vår P-kode (altså for sekvenser av P-instruksjoner). Angi flest mulig ting som denne verifikatoren bør/kan teste angående et gitt slikt program, og skisser hvilke datastrukturer m.m. du vil bruke for å utføre testen. Beskriv tingene direkte i forhold til vår spesielle P-kode. Et program skal både starte og avslutte med tom stakk. Du kan anta at programmets hoppinstruksjoner faktisk går til en instruksjon i programmet, så dette behøver du ikke teste.

Forklar også i hvilken forstand et P-kode-program er ”riktig” om det passerer testen din.

4b

Under står tre programmer i vår P-kode. Sjekk for hver av dem om de passerer testen din, og angi hva som eventuelt går galt om de ikke gjør det.

Program 1:

```
lda x
ldv y
ldv z
jge L1
add
add
ldc 5
lab L1
ldc 8
sto
```

Program 2:

```
lda x
ldv y
ldv z
jge L1
ldc  5
add
lab L1
sto
```

Program 3:

```
lda x
ldv y
ldv z
jge L1
ldc  5
add
ldv u
lab L1
sto
```

4c

Vi vil oversette vår P-kode til maskinkode for en maskin der alle operasjoner (inkl. sammenlikninger) må gjøres mellom verdier som ligger i registre, og der kopiering mellom lageret og registre bare kan gjøres med egne LOAD- og STORE-instruksjoner. Under oversettelsen har vi en stakk med diskriptorer.

Vi skal se på det å oversette P-instruksjonen ”ldv v”. Spørsmålet er om det da er fornuftigst å produsere en LOAD-instruksjon som henter verdien av variabelen ”v” opp i et register, eller om det er best bare å legge en diskriptor på stakken som sier at denne verdien ligger i variabelen ”v”. Drøft dette ut fra forskjellige forutsetninger, f.eks. ut fra hva språket som vi oversetter *fra* lover om rekkefølgen ved beregning av uttrykk (men også ut fra andre ting som du mener er aktuelle).

4d

Vi vil igjen oversette vår P-kode til maskinkode, slik som i oppgave **4c**, og vi skal anta at vi skal oversette én og én basal blokk, og at alle registre skal tømmes på kontrollert måte etter utførelsen av en basal blokk. Spørsmålet her er hvilke data diskriptorene på stakken skal inneholde, og hva du eventuelt trenger av andre typer diskriptorer. Vi antar at vi *kan* tillate oss å lete gjennom alle kompilator-stakkens diskriptorer hver gang vi lurer på hvor visse verdier er etc., slik at informasjon vi kan finne på denne måten ikke behøver å lagres i ekstra diskriptorer.

Forklar også kort hvordan du kan finne den informasjonen du trenger under kodegenereringen, og hvordan du eventuelt vil bruke de ekstra diskriptorene du vil ha.

Lykke til!

Stein Krogdahl og *Birger Møller-Pedersen*

Vedlegg til besvarelse av Oppgave 3

Kandidat nr: .....

Dato: .....

Grammar Rule	Semantic Rule
class → **class** ***name*** ***implements*** *interfaces* **{** *decls* **}**	
$decls_1$ → $decls_2$ **;** *decl*	
decls → *decl*	
decl → *method-decl*	
method-decl → *type* ***name*** **(** *params* **)** *body*	
type → **int**	*type*.type = int
type → **bool**	*type*.type = bool
type → **void**	*type*.type = void
$interfaces_1$ → $interfaces_2$**,** *interface*	
interfaces → *interface*	
interface → ***name***	*interface*.interfaceName = ***name***