UNIVERSITETET I OSLO

**Det matematisk-naturvitenskapelige fakultet**

Eksamen i : INF5110 - Kompilatorteknikk

Eksamensdag : Onsdag 6. juni 2012

Tid for eksamen : 14.30 - 18.30

Oppgavesettet er på : 6 sider (pluss vedlegg)

Vedlegg : 1 side (side 7 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. Deler av oppgave 3 besvares ved bruk av vedlegg.

Oppgave 1 (25%)

Vi skal se på følgende grammatikk G1:

S → a | S # S | S @ S

Her er S startsymbol og eneste ikke-terminal, mens a, # og @ (samt avslutnings-symbolet $) er teminalsymboler.

**1.a**

Gi en konkret begrunnelse for at G1 er flertydig.

**1.b**

Anta at:

* Operasjonen # har lav presedens, og er høyreassosiativ
* Operasjonen @ har høy presedens, og er venstreassosiativ

Angi en ny grammatikk G2 som er entydig, som beskriver samme språket som G1 og som gir et syntaks-tre som følger de to reglene over. Du kan innføre nye ikke-terminaler, og du behøver ikke argumentere for at G2 er entydig ut over å vise til at den likner tilsvarende entydige grammatikker i pensum.

**1.c**

Vi ser på grammatikkene G1, G2, samt følgende grammatikk G3 (der + er et nytt terminal-symbol):

S → a | S # S | S @ S | + S +

Angi for hver av språkene L(G1), L(G2) og L(G3) om de *er* eller *ikke* er regulære. Forklar, og angi et regulært uttrykk for de som eventuelt er regulære.

**1.d**

Tegn opp LR(0)-DFA’en til den flertydige grammatikken G1 (med vanlig bruk av S’).

**1.e**

Beregn First og Follow til S i G1 (med vanlig bruk av $). Angi så hvilke tilstander i DFA’en fra **1.d** som har:

1. konflikter som *ikke* kan løses med LR(0)-betrakninger, men som *kan* løses med SLR(1)-betrakninger. Forklar.
2. konflikter som ikke kan løses med SLR(1)-betrakninger. Forklar.

**1.f**

For tilstandene under punkt B i spørsmål **1.e**, angi hvordan du ville løse konfliktene i disse «for hånd», om du skal få den presedensen og assosiativiteten som er angitt i **1.b**?

**1.g**

Sett opp en SLR(1)-parseringstabell for L(G1) ut fra svaret på spørsmålene **1.d** og **1.f**. Tabellen skal altså ha maks én aksjon i hver rute, og den resulterende syntaksanalysen skal altså følge reglene fra **1.b**.

Oppgave 2 (25%)

**2.a**

Vi tenker oss i denne oppgaven at vi har et Java-lignende språk hvor metoder kan ha lokalt definerte metoder. Dessuten kan man deklarere variable og metoder også på ytterste programnivå. Dette skal fungere som vanlig i språk som er statisk skopet.

Det følgende er et program i dette språket. Oppstart av programmet skjer ved å kalle main-metoden.

{

 class C {

 void m1(){

 void f() {};

 f();

 }

 void m2() {

 int i;

 void g() {

 int j;

 j=i;

 };

 i=1;

 rC.m1();

 };

 };

 C rC;

 void main() {

 rC = new C (); rC.m2();

 }

}

Tegn kall-stakken som den ser ut når aktiveringsblokken (’activation record’) for f er på toppen av stakken for første gang, inklusive variable, access-linker og control-linker, bortsett fra access-linker for metoder som er direkte deklarert i en klasse (kan da anta at access-linken peker til C-objektet, men dette er ikke viktig for oppgaven).

**2.b**

I resten av oppgave **2** innfører vi metoder som parametere. Det er en regel at aktuelle parametere til slike må være metoder som er direkte synlige fra kallstedet. Eksemplet over blir så endret slik at f blir en metodeparameter, og kallet på f i m1 blir da et kall på en metodeparameter:

{

 class C {

 void m1(void f()){

 f();

 }

 void m2() {

 int i;

 void g() {

 int j;

 j=i;

 };

 i=1;

 rC.m1(g);

 };

 };

 C rC;

 void main() {

 rC = new C (); rC.m2();

 }

}

Tegn kall-stakken som den ser ut når aktiveringsblokken for kallet rC.m1(g)er på toppen av stakken. Hvordan representeres metoden g i denne aktiveringsblokken slik at kallet på parameteren f (dvs kallet f() i m1) kan utføres?

**2.c**

Tegn kall-stakken som den ser ut når aktiveringsblokken for kallet av f i m1 er på toppen av stakken.

Forklar hvordan access-link for aktiveringsblokken på toppen av stakken settes i dette spesielle tilfellet.

Oppgave 3 (25%)

Det følgende er en del av en grammatikk for et språk med klasser. Det er bare tatt med de produksjoner som har betydning. Klasser har for eksempel også variable, men de er ikke viktige her.

*class →* **class *name {*** *methodDecls* ***}***

*methodDecls → methodDecls ; methodDecl*

*methodDecls → methodDecl*

*methodDecl →*

 *type* ***name******(*** *parameters* ***)*** *body*

*parameters → parameters , parameter*

*parameters → parameter*

*parameter → type* ***name***

*type →* **int**

*type →* **bool**

Den regel som skal spesifiseres ved hjelp av en attributtgrammatikk er at en klasse kan ha flere enn én constructor, men de må ha forskjellige signaturer i form av antall og/eller typer av parametere.

Lag attributtgrammatikken basert på ideen om at methodDecl har et attributt constructorName som er satt sammen av navnet på metoden og strenger som tilsvarer typene på parameterne. En constructor C med parametertypene (int, int) vi således få constructorName ’C\_i\_i’, mens en constructor C med parametertyper (int, bool) vil få constructorName ’C\_i\_b’. Testen vil derfor være, som antydet i første rekke i tabellen under, at alle navne i mengden av constructornavne er forskjellige. Vi innrømmer at dette ikke er det mest optimale, men det er ikke poenget her.

Definer de regler som gjør denne testen mulig. Anta at du har funksjoner og operatorer for å innsette navne i en menge og for å konkatenere strenge og tegn.

|  |  |
| --- | --- |
| **Grammar Rule** | **Semantic Rule** |
| *class →* **class *name***  ***{*** *methodDecls* ***}*** | *class.*OK = all names in *methodDecls.*SetOfConstructornames are different |
| *methodDecls1 →*  *methodDecls2 ; methodDecl*  |  |
| *methodDecls → methodDecl* |  |
| *methodDecl →*  *type* ***name******(*** *parameters* ***)*** *body* |  |
| *parameters1 →*  *parameters2 , parameter* |  |
| *parameters → parameter*  |  |
| *parameter → type* ***name*** |  |
| *type →* **int** | *type.*typeString = ‘i’ |
| *type →* **bool** | *type.*typeString = ‘b’ |

Oppgave 4 (25%)

En metode med to value-parametere er oversatt til følgende sekvens av TA-instruksjoner. Den eneste typen i språket er heltall.

x = <verdien av første aktuelle parameter> (*Du* kan skrive dette slik: «x = par1»)

y = <verdien av andre aktuelle parameter> (Tilsvarende)

z = x + y

label L1

u = x + 1

x = u + y

if (y < x) goto L4 // Vi antar at det finnes en TA-instruksjon av denne formen

y = u + 1

u = y + x

label L2

z = 5

if (x < u) goto L1

x = u + y

v = x + y

u = v + 1

goto L5

label L4

v = z + 3

x = y + u

label L5

z = v + 4

z = x + z

return z

**4.a**

Del programmet opp i basale blokker, og tegn opp flyt-grafen for programmet. Sett navnene B0, B1, osv. på blokkene.

**4.b**

For hver basal blokk, finn hvilke variable som faktisk er i live både foran og etter blokka (altså *inLive* og *outLive* for blokka). Du kan bruke metoden fra pensum til å finne svaret, eller gjøre egne betraktninger. Det er greit å enten gi svaret direkte på flytgrafen fra **4.a**, eller du kan tegne opp flyt-grafen en gang til (gjerne uten kode i nodene) med alle mengdene inLive og outLive satt på der de hører hjemme.

**4.c**

Ut fra informasjonen fra **4.b** og detaljene i TA-instruksjonene er det mulig å se

1. om noen av TA-instruksjonene i programmet kan fjernes uten at det forandrer sluttresultatet når programmet utføres. Angi i så fall disse
2. om det er variable som, i en eller annen eksekvering, kan bli brukt før de har fått verdi. Angi i så fall disse.

Om du ikke har fått til **4.b** kan du likevel forsøke å svare på denne oppgaven enten direkte fra programmet, eller fra flyt-grafen.

**4.d (Er uavhengig av det over)**

I pensum er det diskutert hvordan man kan lage TA-kode for kortsluttede boolske uttrykk som står som betingelser i if- eller while-setninger. Dette gjøres rekursivt, og den rekursive kodegenererings-metoden har to label-parametere som koden skal hoppe til når man vet at det lokale uttrykket er h.h.v. **true** eller **false**. Programmet for dette er gjengitt under.

Vi oversatte i pensum til TA-kode og ikke til P-kode bl.a. for å slippe å tenke på at det under beregning av uttykket kan være noe på stakken ved hopp, som kanskje ikke stemmer med det stedet det hoppes til. Vi skal her se nærmere på hvor stort dette problemet blir, og hvordan vi eventuelt kan korrigere for det.

Som svar forklar først i detalj hvordan ting vil forholde seg med stakkdybder under uttrykksberegningen, og skissér så hvordan dette kan håndteres under kodegenereringen, gjerne ved å henvise til koden under. **Merk:** Vi antar at P-koden skal lages slik at stakken under kjøring er tom mellom setninger, og at den derved er tom når beregningen av det boolske uttrykket starter.

**Hint:** Det er sikkert lurt å se på hvordan P-koden blir for noen konkrete boolske uttrykk.

**Program fra pensum.** Det genererer TA-kode for logiske uttrykk i if- og while-setninger:

void genBoolCode(String labT, labF) {

 …

 case ”||”: {

 String labx = genLabel();

 left.genBoolCode(labT, labx);

 emit2(”label”, labx);

 right.genBoolCode(labT, labF);

 }

 case ”&&”: {

 String labx = genLabel();

 left.genBoolCode(labx, labF);

 emit2(”label”, labx);

 right.genBoolCode(labT, labF);

 }

 case ”not”: { // Har bare ”left”-subtre

 left.genBoolCode(labF, labT);

 }

 case ”<”: {

 String temp1, temp2, temp3;

 temp1 = left.genIntCode(); temp2 = right.genIntCode();

 temp3 = genLabel();

 emit4(temp3, temp1, «lt», temp2); // Lager instruksjonen: ”temp3 = temp1 < temp2”

 emit3(«jmp-false», temp3, labF);

 emit2(«ujp», labT);

 }

 }

Lykke til!

Stein Krogdahl og Birger Møller-Pedersen

**Vedlegg til besvarelse av Oppgave 3** Kandidat nr: ...................

 Dato: ..............................

|  |  |
| --- | --- |
| **Grammar Rule** | **Semantic Rule** |
| *class →* **class *name***  ***{*** *methodDecls* ***}*** | *class.*OK = all names in *methodDecls.*SetOfConstructornames are different |
| *methodDecls1 →*  *methodDecls2 ; methodDecl*  |  |
| *methodDecls → methodDecl* |  |
| *methodDecl →*  *type* ***name******(*** *parameters* ***)*** *body* |  |
| *parameters1 →*  *parameters2 , parameter*  |  |
| *parameters → parameter*  |  |
| *parameter → type* ***name*** |  |
| *type →* **int** | *type.*typeString = ‘i’ |
| *type →* **bool** | *type.*typeString = ‘b’ |