Örebro universitet
Institutionen för naturvetenskap och teknik
Thomas Padron-McCarthy (thomas.padron-mccarthy@oru.se)

Tentamen i

Kompilatorer och interpretatorer

för Dataingenjörsprogrammet m fl

lördag 6 december 2014

Gäller som tentamen för:
DT3030 Datateknik C, Kompilatorer och interpretatorer, provkod 0100

Hjälpmedel:	Inga hjälpmedel.
Poängkrav:	Maximal poäng är 43. För godkänt betyg krävs totalt minst 24 poäng, varav minst 8 poäng på uppgift 1.
Resultat:	Meddelas på kursens hemsida eller via e-post senast lördag 27 december 2014.
Återlämning av tentor:	Efter att resultatet meddelats kan tentorna hämtas på universitetets centrala tentamensutlämning.
Examinator och jourhavande:	Thomas Padron-McCarthy, telefon 070 - 73 47 013.

- Skriv tydligt och klart. Lösningar som inte går att läsa kan naturligtvis inte ge några poäng. Oklara och tvetydiga formuleringar kommer att misstolkas.
- Skriv den personliga tentamenskoden på varje inlämnat blad. Skriv *inte* namn eller personnummer på bladen.
- Skriv bara på en sida av papperet. Använd inte röd skrift.
- Antaganden utöver de som står i uppgifterna måste anges.
- Skriv gärna förklaringar om hur du tänkt. Även ett svar som är fel kan ge poäng, om det finns med en förklaring som visar att huvudtankarna var rätt.

LYCKA TILL!

Formelsamling

1. Eliminering av vänsterrekursion

En vänsterrekursiv grammatik kan skrivas om så att den inte är vänsterrekursiv. Antag att en regel (eller, korrektare uttryckt, två produktioner), i grammatiken ser ut så här:

```
A -> A x | y
```

A är en icke-terminal, men **x** och **y** står för godtyckliga konstruktioner som består av terminaler och icke-terminaler.

Regeln ersätts av följande två regler (eller, korrektare uttryckt, tre produktioner), som beskriver samma språk men som inte är vänsterrekursiva:

```
A -> y R
R -> x R | empty
```

2. Vänsterfaktorisering

Antag att grammatiken innehåller denna regel (två produktioner):

```
A -> x y | x z
```

A är en icke-terminal, men **x**, **y** och **z** står för godtyckliga konstruktioner som består av terminaler och icke-terminaler.

Skriv om till dessa tre produktioner:

```
A -> x R
R -> y | z
```

Uppgift 1 (10 p)

En kompilators arbete brukar delas in i ett antal faser. Ange vilka det är, och förklara kort vad varje fas gör. Vad är in- och utdata från respektive fas? (Man kan eventuellt få vissa ledtrådar till svaret genom att läsa fråga 2 nedan.)

Uppgift 2 (6 p)

Vi antar att vi programmerar i C i en utvecklingsmiljö med en kompilator som innehåller de traditionella faserna från uppgift 1. Dessutom finns en preprocessor och en länkare.

Visa tydliga exempel, med programkod och förklaringar, på fel som kommer att upptäckas:

a) av preprocessorn
b) av scannern
c) av parsern
d) av den semantiska analysen
e) av länkaren
f) vid körningen av programmet

Uppgift 3 (7 p)

Det här är ett programavsnitt i ett C-liknande språk:

```
x = 7;
y = 6;
while (x < y) {
    z = x + y;
    while (x < z) {
        x = x + 5;
        t = t - x * 4 - 3;
    }
}
```

Översätt ovanstående programavsnitt till var och en av följande tre typer av mellankod.

a) ett syntaxträd, även kallat abstrakt syntaxträd (genom att rita upp trädet!)

b) postfixkod för en stackmaskin

c) treadresskod

Uppgift 4 (2 p)

I uppgift 3 ovan kommer den yttre while-loopen aldrig att köras, eftersom loopvillkoret är falskt redan från början. Vilken eller vilka av kompilatorns faser har detta stor betydelse för, och vad kommer att hända?

Scenario till de övriga uppgifterna

Databashanterare brukar ha särskilda mekanismer för att hantera att flera användare samtidigt arbetar med samma data, för att bland annat hindra att flera användare ändrar i data samtidigt och skriver över varandras ändringar så resultatet blir ett enda sammelsurium. Vanligast är att man använder olika typer av lås.

En användare som ska arbeta med ett dataobjekt i databasen låser det objektet, och om någon annan användare vill arbeta med samma dataobjekt, måste den andra användaren vänta tills de första användaren låser upp objektet.

Vi ska skriva programmet **Locksim** som simulerar låsning. Dataobjekten i databasen är variabler och har namn som **X** och **kalle22**. Man kan *läslåsa* ett dataobjekt med kommandot **ReadLock**, till exempel **ReadLock(X)**, och man kan *skrivlåsa* ett dataobjekt med kommandot **WriteLock**, till exempel **WriteLock(kalle22)**, Dessutom kan man *låsa upp* ett dataobjekt med kommandot **Unlock**.

Det finns redan en Bison-grammatik för inmatningen till programmet:

```
%token TRANSACTION READ WRITE ID LOCK READLOCK WRITELOCK UNLOCK

%%

scenario : transactions ;

transactions : transaction transactions
             | /* empty */
             ;

transaction : TRANSACTION ID '{' commands '}' ;

commands : command commands
         | /* empty */
         ;

command : READ '(' ID ')' ';'
        | WRITE '(' ID ')' ';'
        | LOCK '(' ID ')' ';'
        | READLOCK '(' ID ')' ';'
        | WRITELOCK '(' ID ')' ';'
        | UNLOCK '(' ID ')' ';'
        ;


%%
```

Uppgift 5 (3 p)

Ge tre exempel på Locksim-inmatningar för att visa hur inmatningsspråket ser ut. (Vi kan anta att tokentyperna i grammatiken har vettigt valda namn.)

Uppgift 6 (3 p)

Välj en av dina Locksim-inmatningar från uppgiften ovan och rita upp parse-trädet (även kallat konkret syntaxträd) för den.

Uppgift 7 (5 p)

Programmet Locksim ska förstås inte bara läsa sin inmatning, utan det ska även köra simuleringen och hålla reda på vilka variabler som är låsta. Här är några tekniker från kompilatorteorin.

Förklara för var och en av dem kort (en eller två meningar) vad det är, och om vi kan ha någon nytta av den när vi skriver programmet Locksim. Motivera svaren, dvs förklara antingen hur vi kan ha nytta av tekniken, eller varför vi inte kan det.

a) stackmaskin
b) copy propagation
c) eliminering av svansrekursion
d) Flex
e) mark-sweep

Uppgift 8 (7 p)

Skriv en prediktiv recursive-descent-parser för Locksim-inmatningar, i ett språk som åtminstone liknar C, C++, C# eller Java. Du behöver inte skriva exakt korrekt programkod, men det ska framgå vilka procedurer som finns, hur de anropar varandra, och vilka jämförelser med tokentyper som görs. Du kan anta att det finns en funktion som heter **scan**, och att den returnerar typen på nästa token.
