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

Tentamen i

Kompilatorer och interpretatorer

för Dataingenjörsprogrammet m fl

måndag 3 november 2014

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

Hjälpmedel:	Inga hjälpmedel.
Poängkrav:	Maximal poäng är 42. 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 måndag 24 november 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?

Uppgift 2 (3 p)

När vi "bygger" (dvs kompilerar och länkar) och till sist (efter att ha rättat kompileringsfelen) provkör följande C-program, får vi de angivna fel- och varningsmeddelandena. I vilken av kompilatorns faser (eller på annan plats) upptäcks vart och ett av dessa fel och varningar?

Program	Fel och varningar

`#include <math.h>`
 	
`int main(void) {`
`  int ena = 1, andra = 2 * ena, tredje = 3andra;`
`  double x = sqrt(ena)  sqrt(andra);`
`  double y, z;`
 	
`  x = x * y;`
`  printf("Hej, världen! x = %f\n", x);`
 	
`  return "Ok!";`
`}`
	`invalid suffix "andra" on integer constant`
	`expected ',' or ';' before 'sqrt'`
	`unused variable 'z'`
	
	`'y' is used uninitialized in this function`
	`missing declaration of function 'printf'`
	
	`return makes integer from pointer without a cast`

Uppgift 3 (7 p)

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

```
if (a < b)
    c = a;
else
    c = b;
while (a < b) {
    a = a * 2 + 3 * d;
    b = b - d - 1;
}
```

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 (5 p)

Här är ett C-program. Ett programs adressrymd kan delas upp i fyra delar: programkod och konstanter, statiska data, heap och stack. Rita upp hur stacken (med aktiveringsposter), heapen och statiska data ser ut när programkörningen kommer till utskriften av "Här!".

```
#include <stdlib.h>
#include <stdio.h>

int a = 1, b = 2;

int f(int a) {
    int b = 3;
    printf("Här!\n");
    return a;
}

int g(int c) {
    int b = 4;
    int *d = malloc(sizeof(int));
    *d = c;
    if (c > 5)
        b = b * g(c - 1);
    else
        b = f(c);
    return b;
}

int main(void) {
    int b = 6;
    b = g(7);
    printf("b = %d\n", b);
    return 0;
}
```

Scenario till de övriga uppgifterna

En **struct** i C är ett dataobjekt som innehåller ett eller flera andra dataobjekt. Structar kallas också för poster. Vi ska skriva den del av en C-kompilator som läser (förenklade) struct-deklarationer, Här är ett exempel på en struct som innehåller tre dataobjekt, nämligen ett heltal och två flyttal:

```
struct Punkt {
    int nummer;
    float x, y;
};
```

I de förenklade struct-deklarationer som vårt program ska klara finns bara de två datatyperna **int** och **float**.

Här är ett annat exempel på en struct-deklaration, som visar att C-kod kan skrivas på fritt format, dvs att man kan stoppa in extra mellanslag och radbrytningstecken:

```
struct Banan { float pris;
      float             vikt;
int
banannummer;
    float pos1, pos2, pos3,     pos3b; };
```

Uppgift 5 (3 p)

a) Ange vilka terminaler, dvs typer av tokens, som behövs för att man ska kunna skriva en grammatik för struct-deklarationer.

b) Ange reguljära uttryck för de terminaler som inte har ett fast utseende.

Uppgift 6 (4 p)

Skriv en grammatik för struct-deklarationer. Startsymbolen ska vara **deklaration**, som representerar en enda struct-deklaration enligt scenariot ovan.

Uppgift 7 (3 p)

Rita upp parse-trädet (även kallat konkret syntaxträd) för nedanstående inmatning, enligt grammatiken i uppgiften ovan:

```
struct Vara {
    int nummer;
    float pris, vikt;
};
```

Uppgift 8 (7 p)

Skriv en prediktiv recursive-descent-parser för struct-deklarationer, 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.
