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

Tentamen i

Kompilatorer och interpretatorer

för Dataingenjörsprogrammet m fl

lördag 17 december 2011

Gäller som tentamen för:
DT3004 Datateknik C, Kompilatorer och interpretatorer, provkod 0100

Hjälpmedel:	Inga hjälpmedel.
Poängkrav:	Maximal poäng är 30. För godkänt betyg (3 respektive G) krävs 15 poäng.
Resultat:	Meddelas på kursens hemsida eller via e-post senast lördag 7 januari 2012.
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!

Uppgift 1: Grunder om kompilatorer (5 p)

Här är ett enkelt C++-program:

```
 1  #include <stdio.h>
 2
 3  int main() {
 4      int a, b, c;
 5
 6      printf("Ange a: ");
 7      scanf("%d", &a);
 8      printf("Ange b: ");
 9      scanf("%d", &b);
10
11      c = a/b;
12      printf("c = %d\n", c);
13
14      return 0;
15  }
```

Här är några fel som kan uppstå med detta C-program:

1. Användaren kan mata in **0** som värde på variabeln **b**, och så kraschar programmet pga division med noll.
2. Användaren kan mata in ett mycket litet tal (men inte **0**) som värde på variabeln **b**, och så kraschar programmet pga att resultatet av divisionen blir för stort.
3. Programmeraren kan råka stava fel till **main**
   ```
   3  int mein() {
   ```
4. Programmeraren kan råka glömma ett kommatecken på rad 4:
   ```
   4      int a, b c;
   ```
5. Programmeraren kan glömma ett **"**-tecken på rad 6:
   ```
   6      printf("Ange a: );
   ```
6. Programmeraren kan glömma en parentes på rad 8:
   ```
   8      printf("Ange b: " ;
   ```
7. Programmeraren kan råka skriva **+** i stället för / på rad 11:
   ```
  11      c = a+b;
   ```

Vilka av dessa fel kan kompilatorn upptäcka? I vilka faser av kompileringen upptäcks respektive fel?

Uppgift 2: Mellankod (6 p)

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

```
x = 1;
y = 3;
while (x < y) {
    z = (x + y) / 2;
    x = z;
    if (x < y)
        y = y - 0.1;
    else
        x = x + 0.1;
}
```

Översätt ovanstående programavsnitt till *två* av följande tre typer av mellankod.

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

b) postfixkod för en stackmaskin

c) treadresskod

Observera: Det finns tre deluppgifter i uppgiften ovan. Välj ut och besvara (högst) *två* av dessa. (Skulle du svara på alla tre, räknas den med högst poäng bort.)

Ett scenario

Vi ska skriva ett program som fungerar som en enkel miniräknare, men som inte arbetar med vanliga tal, utan med listor av heltal.

Ett komplett körexempel, med användarens inmatning **understruken**:

```
{ 1, 2, 17 }
Resultat: { 1, 2, 17 }
{ 1, 2, 17 } + { 4, 5 }
Resultat: { 1, 2, 17, 4, 5 }
{ 1, 2, 17 } + { 4, 5 } + { 1, 2 } + { 9, 9, 9 }
Resultat: { 1, 2, 17, 4, 5, 1, 2,  9, 9, 9 }
a = { 1, 2, 17 }
b = { 4, 5 } + { 5, 1}
b
Resultat: { 4, 5, 5, 1}
a + b + { 4, 5 }
Resultat: { 1, 2, 17, 4, 5, 5, 1, 4, 5 }
c = a + b + { 4, 5 }
```

Man ska alltså kunna mata in en lista:

{ 1, 2, 17 }

Listan tolkas som ett värde, och vi får det värdet som svar:

Resultat: { 1, 2, 17 }

Man ska kunna addera två eller flera listor, och vi får en sammanfogad lista som svar, som här:

{ 1, 2, 17 } + { 4, 5 } + { 1, 2 } + { 9, 9, 9 }
Resultat: { 1, 2, 17, 4, 5, 1, 2, 9, 9, 9 }

Man ska också kunna spara listor i variabler. De listor som ska sparas anges med uttryck av samma typ som vi sett ovan:

a = { 1, 2, 17 }
b = { 4, 5 } + { 5, 1}

Variablerna kan sen användas i uttryck. Även uttrycken i variabeltilldelningar kan innehålla variabler.

c = a + b + { 4, 5 }

Uppgift 3: Grammatiker (5 p)

a) (1p) Vilka tokentyper ingår i inmatningsspråket från scenariot?

b) (4p) Skriv en grammatik som beskriver hur *en inmatningsrad* ser ut i språket. Använd terminalerna från deluppgift a. Om grammatiken inte lämpar sig för implementation i form av en prediktiv recursive-descent-parser, ska du också transformera den så den passar.

Uppgift 4: Parsning (6 p)

Skriv en prediktiv recursive-descent-parser för inmatningsspråket ovan. Gör detta i ett språk som åtminstone liknar något vanligt känt programmeringsspråk, till exempel C. 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. Förklara gärna sådant som kan antas vara oklart för läraren.

Du kan anta att det redan finns en funktion som heter **scan**, och att den returnerar typen på nästa token.

Uppgift 5 (3 p)

Här är ett C-program:

```
#include <stdlib.h>

int a;
int b;

void fun1(int c, int d) {
  int e;
  int* p = malloc(sizeof(int));
  e = 10;
  *p = 11;
}

void fun2(int h) {
  int i;
  int* p = malloc(sizeof(int));
  i = 12;
  *p = 13;
  // Här!
}

int main(void) {
  int k;
  k = 15;
  a = 16;
  b = 17;
  fun1(k, 18);
  fun2(19);
  return 0;
}
```

När programexekveringen kommer fram till raden med kommentaren "Här!", så antar vi att vi stoppar programmet och tittar på vilka olika variabler som finns i minnet.

I minnet kommer det då att finnas ett antal lagringsutrymmen för heltal, som innehåller olika tal mellan 10 och 19. Rita upp en skiss över dessa variabler, och förklara vad som är vad. Din förklaring bör bland annat innehålla termerna "statiska data", "stack", "heap", "aktiveringspost" och "parametrar".

Uppgift 6 (5 p)

Vi tänker oss också att vi ska införa en ny konstruktion i C, den så kallade **while-else**-satsen. Tanken är att göra en vanlig **while**-loopen, men med en **else**-gren som körs om inga varv körs i **while**-loopen. Som exempel skulle det här (vanliga) C-programmet:

```
#include <stdio.h>

int main(void) {
    int n;
    printf("Ange antal: ");
    scanf("%d", &n);
    if (n > 0) {
        while (n > 0) {
            printf("*");
            n = n - 1;
        }
    }
    else {
        printf ("Inga rader!");
    }
    printf("\n");
    return 0;
}
```

kunna skrivas så här med while-else:

```
#include <stdio.h>

int main(void) {
    int n;
    printf("Ange antal: ");
    scanf("%d", &n);
    while (n > 0) {
        printf("*");
        n = n - 1;
    }
    else {
        printf ("Inga rader!");
    }
    printf("\n");
    return 0;
}
```

I en syntaxstyrd översättning (på engelska: "syntax-directed translation") associerar man grammatikregler med vad som ska hända. Den finns i två varianter: syntax-styrd definition (på engelska: "syntax-directed definition"), där produktionerna associeras med semantiska regler, som anger hur attribut ska få värden, och syntax-styrt

översättningsschema (på engelska: "syntax-directed translation scheme"), där man stoppar in semantiska aktioner.

a) Skriv en grammatikregel för den här nya while-else-satsen i C.

b) Välj en form av mellankod (bland dem från uppgift 2 ovan), och tala om vilken du valde. Skriv sedan en syntax-styrd definition eller ett syntaxstyrt översättningsschema (och tala om vilken du valde) för översättning av while-else-satsen till mellankod.
