Örebro universitet
Institutionen för teknik
Thomas Padron-McCarthy (Thomas.Padron-McCarthy@oru.se)

Tentamen i

Kompilatorer och interpretatorer

för Dataingenjörsprogrammet m fl

lördag 7 november 2009

Gäller som tentamen för:
DT3004 Datateknik C, Kompilatorer och interpretatorer, provkod 0100
TDK104 Kompilatorer och interpretatorer, provkod 0100

Hjälpmedel:	Inga hjälpmedel.
Poängkrav:	Maximal poäng är 40. För godkänt betyg (3 respektive G) krävs 20 poäng.
Resultat:	Meddelas på kursens hemsida eller via e-post senast lördag 28 november 2009.
Återlämning av tentor:	Efter att resultatet meddelats kan tentorna hämtas på institutionen. Man kan också få sin rättade tenta hemskickad.
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!

Scenario till (en del av) uppgifterna

En *träningsdagbok* är en dagbok där man antecknar sina träningspass, som till exempel löpning och styrketräning. Vi vill att man ska kunna skriva sin träningsdagbok som en fil, och så ska ett program läsa filen och göra olika typer av bearbetning.

Här är ett exempel på hur en fil med en sådan träningsdagbok ska kunna se ut:

```
typ löpning;
typ motionscykling;
2009-10-19 löpning;
       typ  styrketräning ;
2009-10-19 styrketräning;

2009-10-20 löpning 43:12.2;
       2009-10-21 löpning 1:44:00.62    ;
2009-10-21
styrketräning 1:00:00
;
2009-10-22 motionscykling 1:09:22.1;
slut;
```

Som synes ska filen kunna skrivas på fritt format, vilket betyder att blanktecken och radslut inte spelar någon roll, annat än för att skilja orden från varandra.

Man kan skriva in vilka sporter man ägnar sig åt med "satser" som börjar med nyckelordet **typ**. Man kan skriva in träningspass med "satser" som börjar med ett datum, följt av namnet på en sport (som tidigare måste ha deklarerats med en **typ**-sats), eventuellt följt av en tid. Dessutom finns en **slut**-sats som markerar slutet på träningsdagboken. Alla satser avslutas med semikolon.

Det finns alltså ett "träningsdagboks-språk" som beskriver hur träningsdagboken ser ut, och ett program som ska kunna läsa och förstå träningsdagboken måste kunna läsa och förstå det språket.

Uppgift 1 (5 p)

Vi antar att **datum** och **tid** är två egna tokentyper (även kallade terminaler).

a) (2p) Vilka övriga tokentyper ingår i träningsdagboks-språket?

b) (1p) Ett exempel på hur ett datum kan se ut är **2009-10-19**. Datum ska bestå av årtal med fyra siffror, ett bindestreck, månad med två siffror, ännu ett bindestreck, och till sist datum med två siffror. Skriv ett reguljärt uttryck som beskriver tokentypen **datum**.

c) (2p) Den träningstid som man kan ange för en träning kan bestå av timmar, minuter, sekunder och decimaldelar av en sekund, som till exempel **1:09:22.1** och **117:00:09.6293845**. Minuterna och sekunderna måste vara med, men timmarna och sekunddecimalerna kan uteslutas, som till exempel **09:22.17**, **1:09:22** och **0:09**. Skriv ett reguljärt uttryck som beskriver tokentypen **tid**.

Uppgift 2 (5 p)

a) (3p) Skriv en grammatik för träningsdagboks-språket. Använd terminalerna från uppgift 1. Startsymbolen ska heta **dagbok**, och ska beskriva hela innehållet på filen, enligt scenariot.

b) (2p) Om grammatiken från a-uppgiften inte lämpar sig för implementation i form av en prediktiv recursive-descent-parser, så transformera den så den passar. (Om den redan är lämplig, behöver du inte göra om den, utan du får 2 poäng på den här deluppgiften i alla fall.)

Uppgift 3 (3 p)

Ett parse-träd (ibland kallat "konkret syntaxträd") innehåller noder för alla icke-terminaler. I ett syntax-träd (ibland kallat "abstrakt syntaxträd") har man tagit bort alla "onödiga" inre noder, och flyttat upp operatorerna.

Här är en kort träningsdagbok:

```
typ vila;
2009-10-19 vila 24:00:00;
2009-10-20 fika 1:00:00;
slut;
```

Rita upp ett parse-träd för denna träningsdagbok, med din grammatik från (b-)uppgiften ovan.

Uppgift 4 (5 p)

Skriv en prediktiv recursive-descent-parser för träningsdagboks-språket. 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 (4 p)

Här är ett C-program:

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

int x = 1;

int apa(int a, int b) {
    int* p = malloc(sizeof(int));
    *p = a;
    x = a;
    if (a <= 2) {
        /* Här! */
        return 3;
    }
    else {
        return apa(a - 4, b);
    }
}

void svamp(int a, int y) {
    int z;
    z = 5;
    z = apa(6, x);
}

int main(void) {
    int a;
    int y;
    a = 7;
    y = 8;
    svamp(a, y);
    a = 9;
    return 0;
}
```

Ett programs adressrymd kan delas upp i fyra delar: programkod och konstanter, statiska data, heap och stack. Rita upp hur stacken, heapen och statiska data ser ut när programkörningen kommer till kommentaren "Här!".

Uppgift 6 (7 p)

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

```
i = 0;
j = 10;
n = 0;
while (i < j) {
    if (i % 2 == 0)
         i = i + 2;
    else
         j = j - 1;
    n = (j - i - 1) / 2;
}
```

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

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

b) postfixkod för en stackmaskin

c) treadresskod

Uppgift 7 (5 p)

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 while-satsen i C.

b) Välj en form av mellankod (bland dem från uppgift 6 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-satsen till mellankod.

Uppgift 8 (6 p)

En intresserad (men ibland kanske något förvirrad) person ställer frågor om kompilatorer. Svara på frågorna! Det kan kanske behövas en del förklaringar för att reda ut missuppfattningar som personen har.

a) Kompilatorn har ju faser som "scanning", "parsning" och så vidare. Varför finns det ingen fas i kompilatorn som heter "skräpsamling"? Skräpsamling är ju ett ändå ganska viktigt begrepp i modern programmering, och man vill slippa krångla med manuella **free** och **delete**.

b) Jag tycker att man borde sluta med kompilatorer och bara använda interpretatorer, för interpretatorer måste ju vara mycket snabbare. En kompilator översätter ju först källkoden, innan den kan köras, medan en interpretator börjar köra programmet direkt!

c) Bison säger att jag har en massa "shift/reduce-konflikter" här i min grammatik. Vad är det för nåt? Vad beror det på? Är det farligt?
