Kodprov 2019-01-14

1 Instruktioner

Öppna en terminal och kör följande kommandon:

1. `cd` (detta tar dig till din hemkatalog)
2. `mkdir kodprov190114`
3. `cd kodprov190114`
4. `curl http://wrigstad.com/ioopm/bergamott.zip`
5. `unzip k.zip`

Nu har du fått ett antal filer och kataloger:

`uppgift1` – katalog med alla filer för uppgift 1
`uppgift2` – katalog med alla filer för uppgift 2
`Makefile` – en makefil för att lämna in kodprovet

1.1 Inlämning och rättning

Inlämning går till så här: ställ dig i katalogen `kodprov190114`. Om du har tappat bort dig i filsystemet kan du skriva `cd; cd kodprov190114`. Nu kan du skriva `make handin` för att lämna in kodprovet. När du kör detta kommando skapas en zip-fil med de filer som du har uppmanats att ändra i (inga andra), och denna fil sparas sedan på en plats där vi kan rätta provet. Vid behov kan du köra `make handin` flera gånger – endast den sista inlämningen räknas.

Den automatiska rättningen kommer att gå till så att vi kör dina inlämningar mot omfattande testfall. Du har fått ut mindre omfattande testfall eller motsvarande i detta prov som du kan använda som ett *stöd* för att göra en korrekt lösning. Experiment med att lämna ut mer omfattande tester har visat sig skapa mer stress än hjälp (tänk fler testfall som fallerar) [1].

Om du har löst uppgifterna på rätt sätt och testfallen som du får ut passerar är du förhoppningsvis godkänd.

1.2 Allmänna förhållningsregler

- Samma regler som för en salstenta gäller: inga mobiltelefoner, inga SMS, inga samtal med någon förutom vakterna oavsett medium.
- Du måste kunna legitimera dig.
- Du får inte på något sätt titta på eller använda gammal kod som du har skrivit.
- Du får inte gå ut på nätet.
- Du får inte använda någon annan dokumentation än man-sidor och böcker.
- Det är tillåtet att ha en bok på en läsplatta, eller skärmen på en bärbar dator. Inga andra program får köra på dessa maskiner, och du får inte använda dem till något annat än att läsa kurslitteratur.
- Du måste skriva all kod själv, förutom den kod som är given.
- Du får använda vilken editor som helst som finns installerad på institutionens datorer, men om 50 personer använder Eclipse samtidigt så riskerar vi att sänka servrarna.

Vi kommer att använda en blandning av automatiska tester och granskning vid rättning. Du kan inte förutsätta att den kod du skriver enbart kommer att användas för det driver-program som används för testning här. Om du t.ex. implementerar en länkad lista av strängar kan testningen ske med hjälp av ett helt annat driver-program än det som delades ut på kodprovet.

I mån av tid har vi ofta tillämpat ett system där vi ger rest för mindre fel. Det är oklart om detta system tillämpas för detta kodprov men det betyder att det är värt att lämna in partiella lösningar, t.ex. en lösning som har något mindre fel.

Lycka till!

---

2 C-uppgiften

En tree map är – precis som en hash map – en avbildning från nycklar till värden. I många fall kan en hash map och en tree map implementera samma gränssnitt. Den stora skillnaden ligger i hur datastrukturerna är implementerade internt (en genomgång följer strax).

Denna uppgift går ut på att implementera en tree map med följande gränssnitt:

```
void *treemap_insert(treemap_t *t, int key, void *value);
void *treemap_lookup(treemap_t *t, int key);
bool treemap_has_key(treemap_t *t, int key);
int *treemap_keys(treemap_t *t);
int treemap_size(treemap_t *t);
```

Därutöver följande funktioner för att skapa och riva ned en treemap.

```
treemap_t *treemap_create();
void treemap_destroy(treemap_t *);
```

Viss kod är redan given.

Som namnet antyder bygger en treemap på *träd*. Uppgiften utgår inte ifrån att du har jobbat med träd förut – istället följer här en genomgång. Ett träd är uppbyggt som en sorterad länkad lista vars noder (aka länkar) har en nyckel (`int key`), värde (`void *value`) och därutöver två `next` -pekare som kallas `left` och `right` .

Sorteringen fungerar på följande sätt: Om vi står i en nod med nyckelvärdet k och följer `left` -pekaren kommer vi enbart att kunna hitta noder vars nycklar är *strikt mindre än* k. Om vi istället följer `right` -pekaren kommer vi enbart att kunna hitta noder vars nycklar är *strikt större än* k.

Detta gör det effektivt att söka i ett träd, men det är inte så viktigt för detta kodprov.

Nedan följer ett exempel på ett träd där endast nycklarna ritats ut för enkelhelt skull. Det torde vara enkelt att se att noderna i trädet uppfyller invarianten ovan, dvs. till vänster om varje nod finns bara noder med mindre nyckelvärden, och till höger finns bara noder med större nyckelvärden.

```
  treemap_t
     |
    _5_
   /   \
  2     8
 / \   / \
1   4 6   32
```

Om jag skall *leta* efter t.ex. nyckelvärdet 4 i trädet ovan börjar jag i *roten* av trädet och jämför dess nyckelvärde k med 4. Om $k > 4$ vet vi att lösningen ligger till vänster (vi brukar säga “i vänster subträd”), och vi skall alltså följa `left` -pekaren. Om $k < 4$ vet vi att lösningen ligger till höger (höger subträd), och vi skall alltså följa `right` -pekaren. Om $k = 4$ vet vi att vi har hittat rätt nod! Observera att detta är en rekursiv algoritm, efter att vi gått ned i vänster subträd i vårt exempel kan vi starta från början av denna beskrivning med `2` som *rot* (den är ju roten av subträdet) jämföra $2 < 4$ och se att lösningen ligger till höger (etc.).

Om vi vill *lägga till* något till trädet – säg nyckeln 3 och något värde v – utför vi exakt samma operation som ovan, tills dess att vi inser att nästa rekursiva steg skulle få oss att “trilla ut ur trädet”. I exemplet med nyckelvärdet 3 kommer vi efter två steg att hamna i noden med nyckelvärdet 4 (gå vänster från 5 till 2, gå höger från 2 till 4). Eftersom $3 < 4$ kommer vi att vilja följa `left` -pekaren, men då märker vi att dess värde är `NULL` ! Det betyder att vi har hittat platsen där trädet skall utökas – vi skapar en ny nod n med nyckeln 3 och värdet v och tilldelar resultatet till `left` -pekaren i 4.

Observera att n saknar subträd – dvs. `left` och `right` är `NULL` .

Ett tomt träd:

```
treemap_t
   |
  null
```

Om man följer algoritmen för att lägga till något i ett träd som gavs ovan kommer trädet inte automatiskt att förgrena sig jämnt (vi brukar säga att ett träd är balanserat eller inte). Det träd som du skall skapa skall **inte** vara balanserat, så du behöver inte tänka på sådana aspekter. Det betyder att om man skapar ett träd med nycklarna 1, 2, 3, 4, 5 i den ordningen skapas ett träd som ser ut så här:

```
treemap_t
   |
   1
    \
     2
      \
       3
        \
         4
          \
           5
```

Detta träd fungerar precis som en länkad lista: alla `left`-pekare är `NULL` och `right`-pekarna pekar på en nod med nästa nyckelvärde, förutom den sista noden vars `left` är `NULL`.

2.1 Uppgiften

Uppgiften går ut på att skriva klart implementationen av en treemap som finns i filen `yourcode.c`.

Du skall skriva all kod i filen `yourcode.c`. Filen `treemap.h` innehåller signaturerna för alla publika funktioner som skall finnas i `yourcode.c` samt dokumentation för hur de olika funktionerna skall fungera. Det finns markeringar av typen `/// TODO` i filen som beskriver vad du skall göra.

Du kan förutsätta att `NULL` inte används som ett värde.

2.2 Icke-funktionella krav

- Du måste implementera hela programmet från grunden.
- Du får inte läcka minne eller läsa utanför initierat minne.
- Du skall skriva all din kod i `yourcode.c` som är den enda fil som lämnas in!

2.3 Testa din lösning

- `make compile` kompilerar `yourcode.c` mot testerna.
- `make test` kör testerna.
- `make memtest` kör testerna i valgrind för att hitta eventuella minnesfel.

3 Java-uppgiften

Denna uppgift bygger på AST-delen av den symboliska kalkylatorn (dvs. ingen parsning). Koden för ett abstrakt syntaxträd med följande konstruktioner är utdelad. *Du kommer inte att behöva läsa och förstå den mesta av denna kod!*

- Satser (`Statement`):
  - Tilldelning (`Assigment`), t.ex. `x = 42` som tilldelar resultatet av ett uttryck till en variabel.
  - Utskrift (`Print`), t.ex. `print(3 + 5)` som skriver ut resultatet av ett uttryck på terminalen.
  - Variabeldeklaration (`VariableDeclaration`), t.ex. `var x` som anger att en variabel finns och kan tilldelas.
  - Sekvenser (`Sequence`), t.ex. `var x; x = 42; print(x);` som grupperar `;`-separerade satser som evalueras i ett gemensamt environment.
  - Uttryck enligt nedan.
- Uttryck (`Expression`):
  - Variabler (`Variable`), t.ex. `x`.
  - Addition (`Addition`) och subtraktion (`Subtraction`) t.ex. `x + 4` och `y + (2 - z)`.
  - Konstanter av heltalstyp (`Integer`), flyttalstyp (`Float`) och strängkonstanter (`StringLiteral`).
  - Konstanten “bottom” (`Bottom`) som avser avsaknaden av ett värde (jmf. `null`).

Utöver dessa klasser finns ytterligare abstrakta klasser som ingår i en arvshierarki under `Statement`, samt klassen `Environment` som håller reda på variablers värden.

För enkelhets skull är dessa klasser deklarerade i två olika filer:

- `ASTFixed.java` – här ligger klasser som du **INTE FÅR** ändra i
- `YourCode.java` – här ligger klasser som du **FÅR** ändra i (detta är den enda fil som lämnas in!)

Testkod finns i `Driver.java`.

3.1 Uppgift

Din uppgift är att utöka AST-trädet med en ny sats: `IfThenElse`, motsvarande en if-sats i C eller Java. En `IfThenElse`-nod har tre attribut, ett expression, `guard`, samt två statements, `thenBody` och `elseBody`. En `IfThenElse`-nod evalueras enligt följande:

1. Först evalueras `guard` i det aktuella scopet
2. Om `guard` evalueras till ett värde som *inte* är noll evalueras `thenBody` *i ett nytt scope*
3. Annars evalueras `elseBody` *i ett nytt scope*
4. Sist returneras `Bottom`

Att evaluera något i det aktuella scopet betyder att det evalueras med det environment som skickades in till dess `eval()`-metod. Nu följer en förklaring av vad som avses med att evalueras i ett *nytt* scope. Vi börjar med ett exempel i C:

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

int main(void)
{
  int x = 42; /// yttre x deklareras
  printf("before: %d\n", x);
  if (rand() % 2 == 1) { /// thenbody
    x++;
    int x = 0; /// inre x deklareras
    x++;
    printf("thenBody: %d\n", x);
  } else { /// elsebody
    x++;
    printf("elseBody: %d\n", x);
  }
  printf("after: %d\n", x);

  return 0;
}
```

Om `rand() % 2 == 1` ovan är `true` kommer vi att köra "thenbodyn", dvs. det första blocket. Följande resultat syns då vid körning:

```
before: 42
thenBody: 1
after: 43
```

Vi kan se att `x` från början avser "yttre" x, men efter att vi introducerat det "inre" x med `int x = 0` ändras det så att `x` sedan avser det "inre" x. När vi når slutet av "thenbodyn" försvinner alla variabler deklarerade där, dvs. "inre" x försvinner. När vi når den sista utskriften avser x den "yttre" x igen.

Om `rand() % 2 == 1` ovan är `false` kommer vi att köra "elsebodyn", dvs. det andra blocket. Då finns bara den "yttre" x-variabeln. Resultat av körningen:

```
before: 42
elseBody: 43
after: 43
```

Observera att detta scope-beteende skiljer sig något från Inlupp 4 år 2018.

Att evaluera något i ett nytt scope betyder helt enkelt att vi skall använda ett nytt environment! Dock kan vi inte helt kasta bort det gamla environmentet, eftersom koden kanske vill läsa eller skriva till variabler i det gamla environmentet. Om `env` är det aktuella environmentet betyder det att vi kan evaluera `thenBody` enligt följande: `this.thenBody.eval(new Environment(env))`.

För att implementera stöd för scopes kan vi följa denna enkla design: Utöka `Environment` med en `next`-pekare så att varje environment effektivt blir en länkad lista av minst ett `Environment`. Sedan ändrar vi `hasVariable()`, `updateVariable()` och `readVariable()` så att de först kollar igenom det första environmentet i listan, och om variabeln inte finns går de vidare och kollar på nästa environment. När vi skall evaluera `thenBody`, t.ex. skapar vi först sålunda ett nytt tomt environment med det existerande environmentet som `next` och använder detta nya environment för anropet till `eval()` på `thenBody`.

3.2 Tester

För att underlätta felsökning finns ytterligare en ny sats: `dump()`. Denna sats skriver ut innehållet i ett environment så att nästlade scope blir synliga. Inre scopes skrivs ut nästlade i yttre, t.ex. `Scope{ Scope{ x => 2 } , x => 1 }`.

3.2.1 Test 1

Detta test prövar att skapa ett enda scope (ett vanligt environment) med en variabel `x` med värdet 1. Det skall skriva ut följande: `Scope{ x => 1 }`.

3.2.2 Test 2

Detta test prövar att skapa två nästlade scopes (som sker i en if-sats) med `x = 1` och `y = 1` ytterst och `x = 2` innerst. Det skall skriva ut följande: `Scope{ Scope{ x => 2 } , x => 1 , y = 1 }`.

3.2.3 Test 3

Skapar samma nästlade scopes som i Test 2 och printar sedan värdet på `x` och förväntar sig att få 2.

3.2.4 Test 4

Skapar samma nästlade scopes som i Test 2 och printar sedan värdet på `y` och förväntar sig att få 1.

3.2.5 Test 5

Skapar samma nästlade scopes som i Test 2 och tilldelar sedan `3` till `x` och `y` förväntar sig att få följande resultat: `Scope{ Scope{ x => 3 } , x => 1 , y = 3 }`.

3.2.6 Test 6

Skapar samma nästlade scopes som i Test 1 och exekverar sedan en if-sats `if (1) { x = "then" } else { x = "else" }` och förväntar att `x` har värdet `"then"`.

3.2.7 Test 7

Skapar samma nästlade scopes som i Test 1 och exekverar sedan en if-sats `if (0) { x = "then" } else { x = "else" }` och förväntar att `x` har värdet `"then"`.

3.2.8 Test 8

Skapar samma nästlade scopes som i Test 1 och exekverar sedan en if-sats `if (1) { var x; x = "then" } else { var x; x = "else" }` och förväntar att det yttre `x` har värdet `1` efteråt.

3.2.9 Test 9

Skapar samma nästlade scopes som i Test 1 och exekverar sedan en if-sats `if (0) { var x; x = "then" } else { var x; x = "else" }` och förväntar att det yttre `x` har värdet `1` efteråt.

3.2.10 Test 10

Integrerat test med följande program.

```
var x;
var y;
x = 42;
y = 0;
if (y) {
  x = x + 1;
  var x;
  x = 1;
  x = x + 1;
  print("thenBody: " + x)
  dump();
} else {
  x = x + 2;
  print("elseBody: " + x)
  dump();
}
dump();
```

Som synes startar variablerna `x` och `y` med värdena 42 och 0. Eftersom `y` är 0 går vi in i elsebodyn som ökar `x` värde med 2, och skriver ut `elseBody: 44`. Innan if-satsen tar slut använder vi `dump()` för att skriva ut scopes och får `Scope{ Scope{ } , x => 44 , y => 0 }`, dvs. det yttre scopet har värdet 44 på `x` och 0 på `y` och det innersta scopet är tomt. Efter if-satsen använder vi `dump()` igen och ser då att endast det yttre scopet är kvar, men samma värden som sist: `Scope{ x => 44 y => 0 }`.

3.2.11 Test 11

Detta test är exakt som test 10 men `y` har värdet 1 så vi går in i thenbodyn. Eftersom vi deklarerar en ny x-variabel i det inre scopet förväntar vi oss utskriften `thenBody: 2`. När vi dumpar scopet precis innan slutet på thenbodyn har vi två aktiva scopes. Det inre har en x-variabel med värde 2 och det yttre en x-variabel med värde 43 och y-variabel med värde 1: `Scope{ Scope{ x => 2 } , x => 43 , y => 1 }` Efter if-satsen använder vi `dump()` igen och ser då att endast det yttre scopet är kvar, men samma värden som sist: `Scope{ x => 43 y => 1 }`.

3.2.12 Test 12

Detta test prövar nästlade if-satser. Om du löser environments enligt förslaget kommer detta test att fungera direkt om test 2 och 3 fungerar. Om test 2 och 3 fungerar men inte detta har du förmodligen missat att gå vidare till nästa environment i `readVariable()` eller `updateVariable()` eller liknande.

3.2.13 Test 13

Prövar att ett scope returnerar bottom.

3.3 Att kompilera och köra testerna

Kompilera med `make compile` som kompilerar alla källkodsfiler och lägger resultatet i underkatalogen `classes`. För enkelhets skull finns inga paket.

Kör ditt program med `make run`.

Testa att ditt program ger rätt output med `make test`. Testerna är implementerade genom att programmets utdata stämmer överens med förväntat utdata som ligger i filen `expected_output.txt`. Om du vill stoppa in spårutskrifter i ditt program för så kallad “println debugging”, gör då det med `System.err.println(...)` så att testerna inte tolkar spårutskrifterna som en del av testresultaten.

---

[1] Att lämna ut exakt samma test som används vid rättning är heller inte lämpligt, då det har förekommit fall då studenter *försökt* simulera en korrekt lösning genom att bara hacka output för specifika testvärden.