Kodprov 2018-12-11 (förmiddag)

1 Instruktioner

Öppna en terminal och kör följande kommandon:

1. `cd` (detta tar dig till din hemkatalog)
2. `mkdir kodprov 181211`
3. `cd kodprov 181211`
4. `curl http://wrigstad.com/iooopm/ equinox.zip -o k.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 `kodprov 181211`. Om du har tappat bort dig i filsystemet kan du skriva `cd; cd kodprov 181211`. 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-uppgift

Uppgiften går ut på att slå samman två länkade listor A och B så att alla *länkar* och deras element förs över från A till B och att A "töms". **För enkelhetens skull kan du utgå från att alla element är unika.**

Anledningen till att vi inte bara vill flytta över elementen från A till B, utan också länkarna, är att vi är intresserade av att anropa `malloc()` (eller motsvarande) så lite som möjligt, dvs. av rena prestandaskäl – både den extra tiden det tar att allokera och frigöra minne, och det extra minnestrycket i att ha dubbelt så många länkar (som i A) en kort stund innan de togs bort (det skulle vi ha i en naiv implementation som flyttade över alla element i A till B och sedan tog bort alla länkar i A).

Utöver `list_merge()` måste du också implementera funktionerna `list_size()` som returnerar storleken på en lista och `list_destroy()` som tar bort en lista ur minnet, tillsammans med dess innehåll. Intruktioner finns i kommentarer i koden.

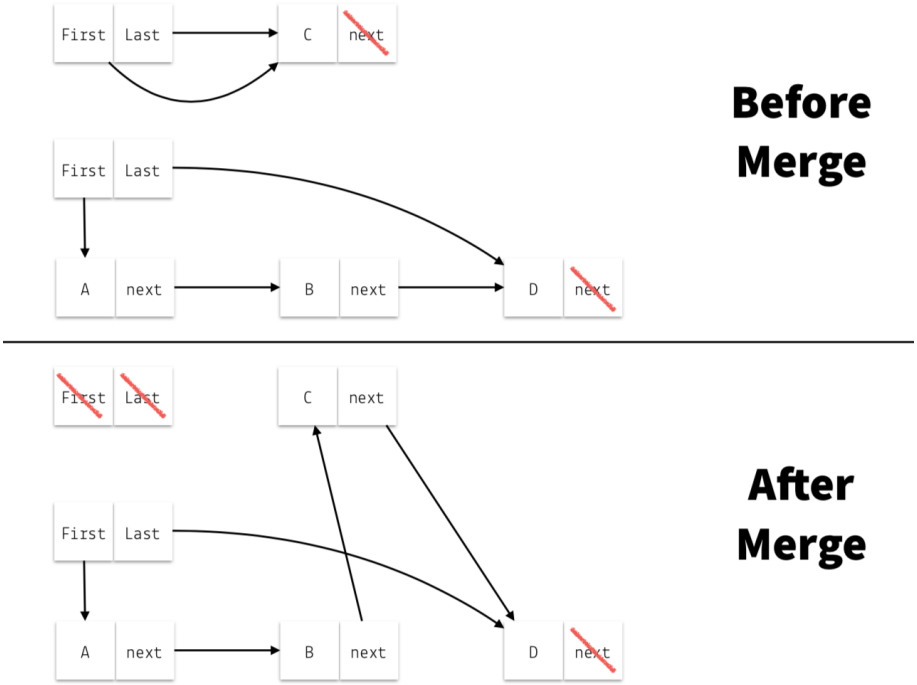
Kod för listorna är given, förutom funktionerna som du skall implementera. All kod du skall skriva är i `yourcode.c` och all testkod är i `driver.c`. **OBS!!** Endast `yourcode.c` lämnas in!

Du behöver inte ändra i `driver.c` men du kan vilja göra det för att t.ex. kommentera bort tester som inte fungerar, etc. Det kan också vara hjälpsamt att läsa koden i `driver.c` för ledning.

Som vanligt får du inte läcka minne, läsa oinitiera minne etc.

2.1 Exempel på merge

Ponera två länkade listor (OBS! Utan dummies, vilket de kan ha i implementationen.) Figuren nedan visa före och efter anrop till `list_merge(list1, list2)`.



Observera att “C” ligger kvar i samma länk, liksom “B” etc.

2.2 Testa din lösning

`make compile` kompilerar `driver.c` och `yourcode.c` till `a.out`.

`make test` kompilerar enligt `make compile` och kör sedan testerna.

`make memtest` kör testerna enligt `make test` under valgrind.

Testerna slår ihop två listor tre gånger, i tre olika test. **Du måste själv läsa outputen och tolka den för att se om allt gick bra!**

Här är ett exempel på ett test som passerar:

```
========================== TEST START ================================
First random list 'a'
'[k, r, v]'
Second random list 'b'
'[g, q, x]'
Merging lists
Expecting size of 'a' to be zero
0 == 0 ... PASSED
Expecting size of 'b' to be sum of 'a' and 'b' before merge
6 == 6 ... PASSED
Expecting the output of 'a' to be []
'[]'
Expecting the output of 'b' to be all elements of 'a' and 'b' before merge in ascending
order (e.g. a < b, b < c, etc.)
'[g, k, q, r, v, x]'
Checking that the links, not just elements were moved
... PASSED
========================== TEST STOP ================================
```

Som synes skapas två listor (samma skapas varje gång du kör testet!), i detta fall båda av längden 3, med unika data. Först skrivs listorna ut så vi ser vad som händer, sedan slås de ihop. Efter detta förväntar vi oss att a är tom, och att b har fått samtliga element. Sista testet kollar att det faktiskt är *länkarna* som flyttas över och inte enbart elementen.

3 Lösningsförslag till C-uppgiften

Det var några små skillnader mellan förmiddagens och eftermiddagens prov. T.ex. sorteringsordningen som användes av testerna och icke-funktionella krav på hur listans storlek skulle räknas ut (amorterad kostnas och size i konstant tid jmf. med size i linjär tid).

3.1 De vanligaste felen

1. Missa att det var länkarna som skulle flyttas över och bara flytta elementen
2. Förutsätta något om sorteringsordningen istället för att använda funktionspekarna
3. Ignorera sorteringen och bara konkatenera listorna
4. Blanda ihop listorna efter merge (båda listorna hade gemensamma länkar)
5. Missa att uppdatera `last`

3.2 Lösningsförslag `list_size()`

Visar endast i linjär tid:

```c
int list_size(list_t *list)
{
  int sum = 0;
  for (link_t *cursor = list->first; cursor->next; cursor = cursor->next)
    {
      ++sum;
    }
  return sum;
}
```

3.3 Lösningsförslag `list_destroy()`

```
void list_destroy(list_t *list)
{
  /// Ta fram en pekare till första länken efter dummyn
  link_t *cursor = list->first->next;

  while (cursor)
    {
      /// Spara undan next-pekaren
      link_t *tmp = cursor->next;
      /// Frigör elementet
      free(cursor->element);
      /// Frigör länken
      free(cursor);
      cursor = tmp;
    }
  /// Frigör dummy-länken (ej dess element eftersom det inte finns)
  free(list->first);
  /// Frigör själva listan
  free(list);
}
```

3.4 Lösningsförslag `list_merge()`

Lösningsförslaget loopar över listan `l1` i den yttre loopen (`for`). Varje varv flyttas en länk från `l1` över till `l2`. Pekaren `prev_l2` används för att hitta föregående länk i `l2` där länken från `l1` skall in. Den inre loopen (`while`) används för att positionera `prev_l2` korrekt.

Efter den inre loopen pekar `prev_l1->next` på den länk som skall flyttas över, och `prev_l2->next` är den plats dit länken skall flyttas.

```
void list_merge(list_t *l1, list_t *l2)
{
  /// Varje varv i loopen flyttas prev_l1->next över
  for (link_t *prev_l1 = l1->first; prev_l1->next;)
    {
      /// Bekvämt för jämförelsen i while-loopen
      void *element = prev_l1->next->element;

      link_t *prev_l2 = l2->first;

      /// Flytta prev_l2 till platsen innan där prev_l1->next skall stoppas in
      while (prev_l2->next && l2->compare(prev_l2->next->element, element) < 0)
        {
          prev_l2 = prev_l2->next;
        }

      /// Låt link_l1 vara länken som skall flyttas
      link_t *link_l1 = prev_l1->next;

      /// Länka link_l1 ur l1
      prev_l1->next = link_l1->next;

      /// Länka in link_l1 i l2
      link_l1->next = prev_l2->next;
      prev_l2->next = link_l1;

      /// Uppdatera last-pekaren
      if (prev_l2 == l2->last)
        {
          l2->last = link_l1;
        }
    }

    /// Uppdatera l1's last-pekare
    l1->last = l1->first;
}
```

Om vi hade haft amorterad `list_size()` hade vi också behövt följande kod:

```
l2->elements += l1->elements; /// Eftersom vi kunde utgå från inga dubletter
l1->elements = 0;
```

4 Java-uppgift

4.1 Översikt över utdelad kod

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 satser separerade med `;` , som kan 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`)
  - 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` . **OBS!!! Du behöver ändra på rad 12 i `Driver.java` för att köra testerna.**

Nedan följer några exempel på valida "program" i form av sekvenser av statements, för tydlighets skull skrivna på olika rader. Följande program skriver ut "49" som sitt resultat.

```
var x;        // anger att variabeln x finns och har värdet "bottom"
var y;        // anger att variabeln y finns och har värdet "bottom"
var z;        // anger att variabeln z finns och har värdet "bottom"
x = 7;
y = 42;
z = x + y;
print(z);
```

Följande program skriver ut "<bottom>" som sitt resultat. Analogt att printa `null` .

```
var z;        // anger att variabeln z finns och har värdet "bottom"
print(z);
```

Följande program skriver ut "A/An java.lang.RuntimeException was thrown (Bottom used as a value!)". Analogt med att avreferera `null` .

```
var x;       // anger att variabeln x finns och har värdet "bottom"
var y;       // anger att variabeln y finns och har värdet "bottom"
print(x + y);
```

Följande program skriver ut "A/An java.lang.RuntimeException was thrown (Access to undeclared variable!)" eftersom `z` inte har deklarerats innan den användes.

```
print(z);
```

4.2 Uppgift

Uppgiften går ut på att utöka systemet med flyttalskonstanter i klassen `Float` som du måste skriva själv från grunden och i filen `YourCode.java` (varför klassen inte får vara publik!). Uppgiftens karaktär är "läsa mycket, skriva lite". Du har ledning av existerande kod i alla delar av uppgiften.

1. Förutom numeriken (flyttal istf. heltal) skall flyttalskonstanter fungera som heltalskonstanter, dvs. de skall gå att skriva ut, de skall stöda addition och subtraktion, det skall gå att spara flyttal i variabler, etc.
2. Addition eller subtraktion mellan två flyttal eller ett heltal och ett flyttal skall returnera ett flyttal.

För att avgöra när du är klar finns några enkla tester som beskrivs nedan.

OBS! Du får inte använda `instanceof` i din lösning. Inte heller får du "lösa" problemet genom att generera och matcha strängar.

4.2.1 Enkla test (fler finns i `Driver.java`)

```
print(42.0);
```

Ovanstående program skriver ut "42.0" som sitt resultat. (`test0()` i `Driver.java`.)

```
var x;
x = 1.0;
print(x);
```

Ovanstående program skriver ut "1.0" som sitt resultat. (`test1()` i `Driver.java`.)

```
var x;
var y;
x = 1.0;
y = 5.0;
print(x + y);
```

Ovanstående program skriver ut "6.0" som sitt resultat. (`test2()` i `Driver.java`.)

```
var x;
var y;
x = 2;
y = 5.0;
print(x + y);
```

Ovanstående program skriver ut "7.0" som sitt resultat. (`test3()` i `Driver.java`.)

```
var x;
var y;
x = 2.0;
y = 6;
print(x + y);
```

Ovanstående program skriver ut "8.0" som sitt resultat. (`test4()` i `Driver.java`.)

```
var x;
var y;
x = 2;
y = 6;
print(x + y);
```

Ovanstående program skriver ut "8" som sitt resultat (använder ej flyttal!). (`testRegression()` i `Driver.java`.)

```
var x;
var y;
x = 1.0;
print(x + y);
```

Ovanstående program skriver ut "A/An java.lang.RuntimeException was thrown (Bottom used as a value!)". (`test5()` i `Driver.java`.)

4.3 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`. **OBS!!! Du behöver ändra på rad 12 i `Driver.java` för att skapa flyttalskonstanter.**

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 såkallad "println debugging", gör då det med `System.err.println(...)` så att testerna inte tolkar spårutskrifterna som en del av testresultaten.

4.4 Klassen `java.lang.Number`

Klassen `java.lang.Number` är superklass till klassen `java.lang.Integer` (bland annat). Du kan se hur den används i klassen `Integer` i `ASTFixed.java` och aritmetik-metoderna `sub()` och `add()` i `Calculation` i `YourCode.java`. Här är några metoder i klassen `Number` som kan vara av intresse.

- `abstract double doubleValue()` Returns the value of the specified number as a double.
- `abstract float floatValue()` Returns the value of the specified number as a float.
- `abstract int intValue()` Returns the value of the specified number as an int.
- `abstract long longValue()` Returns the value of the specified number as a long.
- `short shortValue()` Returns the value of the specified number as a short.

5 Lösningsförslag till Java-uppgiften (fm + em)

Eftermiddagens Java-uppgift skiljde sig från förmiddagens – istället för att lägga till en flyttalskonstant gick uppgiften istället ut på att lägga till en ny abstrakt klass för numeriska konstanter i ett system där flyttals- och heltalskonstanter redan fanns.

Att lägga till en flyttalskonstant är tämligen enkelt:

```
class Float extends Constant {
    public Float(double value) {
        super(value);
    }

    public boolean isFloat() {
        return true;
    }
}
```

Båda kodproven krävde utökning av klassen `Calculation` för korrekt implementation av addition och subtraktion. Förmiddagens kodprov behövde hantera heltal och flyttal emedan eftermiddagens kodprov också behövde hantera strängar. Nedan följer unionen av båda kodproven, och det torde vara enkelt att bara blunda för den del som inte rörde eftermiddagen/förmiddagen vid behov.

```
class Calculation {
    public static Constant add(Constant a, Constant b) {
        if (a.isInteger() && b.isInteger()) {
            return new Integer(a.intValue() + b.intValue());
        } else if (a.isString() && b.isString() || (a.isString() && b.isInteger()) ||
(b.isString() && a.isFloat())) {
            return new StringLiteral(a.stringValue() + b.stringValue());
        } else if (a.isFloat() && b.isFloat() || (a.isFloat() && b.isInteger()) ||
(b.isFloat() && a.isInteger())) {
            return new Float(a.floatValue() + b.floatValue());
        } else {
            throw new RuntimeException("Bottom used as a value!");
        }
    }

    public static Constant sub(Constant a, Constant b) {
        if (a.isInteger() && b.isInteger()) {
            return new Integer(a.intValue() - b.intValue());
        } else if (a.isFloat() && b.isFloat() || (a.isFloat() && b.isInteger()) ||
(b.isFloat() && a.isInteger())) {
            return new Float(a.floatValue() - b.floatValue());
        } else if (a.isString() || b.isString()) {
            throw new RuntimeException("Strings do not support subtraction!");
        } else {
            throw new RuntimeException("Bottom used as a value!");
        }
    }
}
```

Många lösningar gav prov på en mer läsbar implementation, men med mer upprepning:

```
public static Constant sub(Constant a, Constant b) {
    if (a.isInteger() && b.isInteger()) {
        return new Integer(a.value().intValue() - b.value().intValue());
    }
    else if (a.isInteger() && b.isFloat()) {
        return new Float(a.value().floatValue() - b.value().floatValue());
    }
    else if (a.isFloat() && b.isInteger()) {
        return new Float(a.value().floatValue() - b.value().floatValue());
    }
    else if (a.isFloat() && b.isFloat()) {
        return new Float(a.value().floatValue() - b.value().floatValue());
    } else {
        throw new RuntimeException("Bottom used as a value!");
    }
}
```

För subtypspolymorfins skull (eftersom `a.isFloat()` etc. ovan sker mot en statisk `Constant`-typ) behövdes en `isFloat()` i konstant-klassen, vars standardimplementation var `return false`:

```
public boolean isFloat() {
    return false;
}
```

5.1 Lösningsförslag till eftermiddagens kodprov

Utöver `Calculation`-klassen som var delvis delad med förmiddagns kodprov gick eftermiddagens kodprov ut på att lägga till en ny klass `NumberConstant` för numeriska konstanter. Eftersom den ärver av en klass för godtyckliga konstanten (t.ex. även strängliteraler) bestod en av svårigheterna i att inse att en numerisk konstant alltid har ett `value` som är ett `Number`. Så här kunde denna klass exempelvis implementeras:

```
abstract class NumberConstant extends Constant {
    /// TODO
    public NumberConstant(Number value) {
        super(value);
    }

    public Number value() {
        return (Number) super.value();
    }

    public int intValue() {
        return this.value().intValue();
    }

    public double floatValue() {
        return this.value().doubleValue();
    }
}
```

Observera att i och med att konstruktorn alltid tar in ett `Number` som argument vet vi att `value()` som finns i `Constant`-klassen kommer att returnera ett `Number`. Det möjliggör typomvandligen i `value()` vilket i sin tur möjliggör de smidigar implementationerna av `intValue()` och `floatValue()`.

Mindre eleganta (i mitt tycke) lösningar som räknades som helt godkända var struntade t.ex. i att override:a `value()` och istället implementera ex. `intValue()` så här:

```
public int intValue() {
    return ((Number) this.value()).intValue();
}
```

6 Statistik

	C	Java
Godkända	5	58
Godkända efter rest	0	21
Rester ej kompletterade	1	1
Underkända	0	7

Observera att statistiken inte innefattar studenter som inte lämnade in någonting.

Questions about stuff on these pages? Use our Piazza forum.

Want to report a bug? Please place an issue here. Pull requests are graciously accepted (hint, hint).

Nerd fact: These pages are generated using org-mode in Emacs, a modified ReadTheOrg template, and a bunch of scripts.

Ended up here randomly? These are the pages for a one-semester course at 67% speed on imperative and object-oriented programming at the department of Information Technology at Uppsala University, ran by Tobias Wrigstad.

---

[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.