Kodprov 2018-12-19

1 Instruktioner

Öppna en terminal och kör följande kommandon:

1. `cd` (detta tar dig till din hemkatalog)
2. `mkdir kodprov181219`
3. `cd kodprov181219`
4. `curl http://wrigstad.com/ioopm/patata.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 `kodprov181219` Om du har tappat bort dig i filsystemet kan du skriva `cd; cd kodprov181219`. 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

Denna uppgift går ut på att implementera en *string tokenizer* i C, analogt med den som redan finns och som användes i Inlupp 1 (2018) i ordfrekvensräknaren. En string tokenizer delar upp en sträng i delsträngar givet vissa avdelare. Om vi tänker att vår avdelare är `','` (kommatecknet) kan vi exemplifiera med följande strängar (**för tydlighets skull skriver vi `_` istället för mellanslag i exemplen!**):

- `"Fee,_foo,_fi,_fum!"` delas upp i *fyra* strängar: `"Fee"`, `"_foo"`, `"_fi"` och `"_fum!"`.
- `"Han_var,_som_laxhandlare,_oöverträffad."` delas upp i *tre* strängar: `"Han_var"`, `"_som_laxhandlare"`, och `"_oöverträffad."`.
- `"ett,,,två"` delas upp i *två* strängar: `"ett"` och `"två"`.
- `"ett,"`, `",ett"` och `",ett,"` delas alla upp i *en* sträng: `"ett"`.

Observera att endast avdelaren försvinner vid uppdelning, dvs. i ovanstående exempel är alla inledande mellanslag kvar i de resulterade strängarna. Detta går emellertid att lösa genom att ha flera avdelare! Vi skickar in våra avdelare som en *sträng* av tecken som alla är avdelare (ordningen i strängen spelar ingen roll). Om vi har som avdelare "_," (dvs. mellanslag och kommatecken) och återbesöker ovanstående exempel blir vi helt av med mellanslagen:

- `"Fee,_foo,_fi,_fum!"` delas upp i *fyra* strängar: `"Fee"`, `"foo"`, `"fi"` och `"fum!"`.
- `"Han_var,_som_laxhandlare,_oöverträffad."` delas upp i *fem* strängar: `"Han"`, `"var"`, `"som"`, `"laxhandlare"`, och `"oöverträffad."`.

Målet med kodprovet är att vi skall återskapa funktionaliteten hos `strtok()` (och lite mer). Vi börjar med att gå igenom hur `strtok()` fungerar. Följande kodsnutt skriver ut om den kompileras och körs.

```
'Fee'
'_foo'
'_fi'
'_fum!'
```

Kodsnutten:

```
char *str = strdup("Fee,_foo,_fi,_fum!");
char *delimiters = ",";
char *substring = strtok(str, delimiters);

while (substring)
  {
    printf("'%s'\n", substring);
    substring = strtok(NULL, delimiters);
  }

free(str);
```

Observera det "märkliga" anropet `strtok(NULL, delimiters)` inuti loopen. Funktionen fungerar så att vid första anropet sparas strängen som skall delas upp och innan `strtok()` returnerar sparas internt en pekare till var nästföljande anrop skall börja. Ett anrop vars första argument är `NULL` anses vara ett efterföljande anrop och ett vars första argument inte är `NULL` "installerar" en ny sträng som skall tokeniseras. När det inte finns några fel delsträngar returnerar funktionen `NULL`, vilket vi använder ovan som stoppargument.

Observera även att `strtok()` är en destruktiv funktion! Med detta menas att den förstör strängen som skall tokeniseras. Varje delsträng som returneras är faktiskt en delsträng i den ursprungliga strängen, vilket betyder att `strtok()` kör i konstant minne. Om `"Fee,_foo,_fi,_fum!"` är strängen som skickades in och avdelarna är "‚_" kommer vi efter att vi är färdiga (dvs. efter sista anropet i `while`-loopen) att ha följande sträng: `"Fee\0_foo\0_fi\0_fum!"`.

Observera att vi faktiskt aldrig tagit bort några mellanslag ut strängen ovan – vi hoppar istället över dem genom att returnera en pekare till det första tecken som inte är en avdelare: första anropet returnerar en pekare till första (och enda) F:et; andra anropet till första f:et, tredje till andra f:et och fjärde till tredje f:et.

Idag skall vi utöka funktionaliteten hos `strtok()` med att spara undan information så att vi kan återskapa originalsträngen igen. För strängen `"Fee\0_foo\0_fi\0_fum!"` sparar vi en array av int:ar som ser ut så här: `[',', 3, ',' 8, ',', 12, '\0', 18]`; Dessa intar kommer i par: `','` och 3 betyder "ersätt 3:e positionen (indexat från 0) med `','`". Att återskapa originalsträngen betyder då att gå igenom arrayen göra ersättningar tills den första inten i paret är `\0`. Totalt 3 ersättningar skall alltså appliceras för att återfå `"Fee,_foo,_fi,_fum!"` från `"Fee\0_foo\0_fi\0_fum!"`.

2.1 Uppgift del 1

Implementera funktionen `ioopm_strtok()` som tar som argument en sträng `src` som skall tokeniseras, en sträng `delimiters` av avdelare, samt en array av heltal `replacements` som sparar information för att kunna återskapa den ursprungliga `src` efter tokeniseringen.

Du kan förutsätta att det finns nog med plats i `replacements`.

2.1.1 Ledning

- Rita innan du kodar (men titta gärna på koden först så att du ser om det finns begränsningar eller redan givna delar).
- Använd några mycket enkla strängar som exempel.
- Minns att du kan hoppa över alla avdelare utom den som avslutar en delsträng – den avdelaren ersätts istället med `NULL`.
- Ignorera replacements tills du har en fungerande tokenisering.

2.2 Uppgift del 2

Skriv klart funktionen `ioopm_undo_strtok()` som tar som argument en tokeniserad sträng `src` samt en array av heltal `replacements` och återskapar den ursprungliga `src` samt returnerar antalet ersättningar som gjorts.

Glöm inte att se till att `replacements` termineras ordentligt. Ovan föreslog jag att det sista parets första int är 0, men du kan använda en annan lösning om du vill.

2.3 Icke-funktionella krav

- Du måste implementera hela programmet från grunden.
- Då får inte använda några hjälpfunktioner utöver de som är givna – förutom `malloc()`, `calloc()` och `free()` – om du behöver dem?
- 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!

2.4 Utdelad kod (mer kan finnas i `yourcode.c`)

- Funktionen `ioopm_strtok()` är deklarerad och innehåller given kod.
- Funktionen `is_delimiter(c, delimiters)` returnerar `true` om `c` är en avdelare.

2.5 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. *Observera att `Environment` hanteras lite annorlunda än i inlupp 3 – för enkelhets skull har den blivit global.*

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`.

Nedan följer några exempel på valida "program" i form av sekvenser av satser, 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 med 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 deklarerats innan den användes.

```
print(z);
```

3.1 Uppgift

Din uppgift är att utöka AST-trädet med två nya uttryck: `Quote` och `Unquote` [2]. `Quote` returnerar strängrepresentationen av ett uttryck. Här är exempel på användande av `Quote`:

- `Quote(3)` returnerar strängen `'3'`
- `Quote(3 + x)` returnerar strängen `'3 + x'` oavsett vad det är för värde på `x` eller om `x` inte ens är definierad.
- `Quote('Hej' - 4)` returnerar strängen `'\'Hej\' - 4'` – observera att detta uttryck inte går att evaluera, men det är tillåtet att ta dess strängrepresentation

(Notera att `'\'x\''` avser en sträng med en sträng inuti – `\'` är escapade “strängfnuttar”, analogt med `\"` i Java och C.)

`Unquote` kan bara förekomma inuti ett `Quote`-uttryck och “slår på” evaluering. Om `x` sparar värdet `5` kan vi modifiera exempel två ovan enligt följande:

- `Quote(Unquote(3 + x))` returnerar strängen `'8'`
- `Quote(3 + Unquote(x))` returnerar strängen `'3 + 5'`

Det går utmärkt att nästla `Quote` och `Unquote`, men flera nästlande `Unquote` ger upphov till exception:

- `Quote(Quote(x))` returnerar strängen `'Quote(x)'`
- `Quote(Unquote(Quote(x)))` returnerar strängen `'\'x\''`
- `Quote(Quote(Unquote(x)))` returnerar strängen `'Quote(5)'`
- `Quote(Unquote(Unquote(x)))` kastar ett Runtime-exception “Unquote appeared outside of Quote” (det är det inre som kastar exception)

Som test används (minst) exemplen ovan (i någon ordning).

3.2 Ledning

Finns inuti `YourCode.java`.

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.

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.

[2] Kända från bl.a. Lisp/Scheme/Racket/m.fl., där semantiken kan vara lite annorlunda.