Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2019/20

Prof. Dr. Michael Rohs, Tim Dünte, M.Sc.

Übungsblatt 12

Alle Übungen (bis auf die erste) müssen in Zweiergruppen bearbeitet werden. Beide Gruppenmitglieder müssen die Lösung der Zweiergruppe einzeln abgeben. Die Namen beider Gruppenmitglieder müssen sowohl in der PDF Abgabe als auch als Kommentar in jeglichen Quelltextabgaben genannt werden. Wenn Sie für Übungsblatt 1 noch keinen Gruppenpartner haben, geben Sie alleine ab und nutzen Sie das erste Tutorium dazu, mit Hilfe des Tutors einen Partner zu finden. Plagiate führen zum Ausschluss von der Veranstaltung.

Abgabe bis Donnerstag den 23.01. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2019/Programmieren1. Die Abgabe muss aus einer einzelnen zip-Datei bestehen, die den Quellcode, ein pdf bei Freitextaufgaben und alle weiteren nötigen Dateien (z.B. Eingabedaten oder Makefiles) enthält. Lösen Sie Umlaute in Dateinamen bitte auf.

Die Dokumentation der Prog1lib finden Sie unter: https://hci.uni-hannover.de/files/prog1lib/index.html

Aufgabe 1: Spielecharaktere und Objektlisten

Die für diese Aufgabe relevanten Dateien sind object.h, character.h, fighter.{h|c}, wizard.{h|c}, object_list.{h|c} und characters_test.c. Änderungen sind nur in fighter.c, wizard.c und characters_test.c nötig. Die Struktur `Character` soll einen Charaktertyp in einem Computerspiel repräsentieren, für den eine Angriffsstärke (attack) und eine Abwehrstärke (defense) berechnet werden kann. Die Strukturen `Fighter` und `Wizard` repräsentieren Spielecharaktere des jeweiligen Typs.

a) Machen Sie sich mit dem vorhandenen Quelltext vertraut. Die Definition von `Class` in object.h wurde in der Vorlesung besprochen. Die Struktur `CharacterClass` erweitert `Class` um Zeiger auf Funktionen vom Typ `AttackFun` und `DefenseFun`. Funktionen dieses Typs sollen der Berechnung von Angriffs- und Abwehrstärke eines Charakters dienen. Implementieren Sie `attack_fighter` und `defense_fighter` sowie `attack_wizard` und `defense_wizard` wie folgt:

$$attack_{Fighter} = 0.9 * sword * strength$$
$$defense_{Fighter} = 0.8$$
$$attack_{Wizard} = 1.0$$
$$defense_{Wizard} = 0.5 * potion * will^2$$

Überprüfen Sie Ihre Implementierung mit den (zu ergänzenden) Beispielaufrufen in characters_test.c.

b) Erstellen Sie in characters_test.c eine Objektliste mit den Objekten `f1`, `f2`, `w1` und `w2`. Diese Liste stellt eine Gruppe bestehend aus diesen vier Charakteren dar. Berechnen Sie mit einer `for`-Schleife über die Liste Gesamtangriffs- und Gesamtabwehrstärke der Gruppe. Geben Sie das Ergebnis aus.

c) Berechnen Sie nun in characters_test.c die Gesamtangriffsstärke der Objekte in der Liste mit Hilfe der `reduce_list`-Funktion und Ihrer Implementation von `add_attack` und geben Sie das Ergebnis aus.

d) Filtern Sie die Liste, so dass eine Ergebnisliste nur der `Wizards` gebildet wird. Verwenden Sie dazu die Funktion `filter_list` und definieren Sie eine Testfunktion `is_wizard`. Geben Sie die Ergebnisliste aus.

Aufgabe 2: Binärbäume

Die für diese Aufgabe relevanten Dateien sind integer_tree.{h|c}, integer_list.{h|c} und integer_tree_test.c. In dieser Aufgabe werden Operationen für einen Binärbaum implementiert. Der Binärbaum speichert ganze Zahlen.

a) In der `main`-Funktion von integer_tree_test.c findet sich das int-Array `values`. Fügen Sie die Elemente des Arrays geordnet in einen Binärbaum ein. Geben Sie die Elemente des Baums aus, so dass sie in aufsteigend sortierter Reihenfolge erscheinen.

b) Implementieren Sie die Funktion `free_tree` in integer_tree.c, die den Speicher des Baumes freigibt. Beachten Sie, dass der Baum nur ganze Zahlen speichert.

c) Die `print_tree`-Funktion in integer_tree.c gibt Bäume so aus, dass der Wert eines Knotens in der Mitte und die linken und rechten Unterbäume links und rechts davon erscheinen. Um die Struktur sichtbar zu machen wird jeder Knoten geklammert. Die Form ist also: (<left>, <value>, <right>). Der leere Baum wird als Unterstrich dargestellt. Beispiel:

```
(((_, 11, _), 1, (_, 12, _)), 0, (_, 2, (_, 7, _)))
```

Erweitern Sie die `print_tree`-Funktion so, dass Blätter in abgekürzter Form dargestellt werden. Für das Beispiel:

```
((11, 1, 12), 0, (_, 2, 7))
```

d) Implementieren Sie in integer_tree_test.c die Funktion `get_interval_rec`, die einen Baum, eine untere und eine obere Grenze übergeben bekommt. Die Funktion soll aus dem übergebenen Baum alle Elemente in der Ergebnisliste speichern und zurückgeben, die im Intervall [`lower`, `upper`] liegen.

e) Implementieren Sie nun das Filtern eines Intervalls auf andere Weise. Verwenden Sie die Funktion `filter_tree` und implementieren Sie die Funktion `in_interval`.

f) Implementieren Sie die Funktion `is_monotone`. Diese gibt genau dann wahr zurück, wenn für jeden Knoten im Baum gilt, dass die Werte der Kinder größer sind, als der Wert des Knotens. Für den leeren Baum soll die Funktion den Wert `true` zurückgeben.

Tipp: Es kann hilfreich sein, wenn Sie sich einen Baum aufmalen und prüfen, welche Grenzen für jeden Knoten gelten. Bedenken Sie, dass die Grenzen für Wurzelknoten nicht initialisiert sind, da er kein Nachfolgerknoten ist.

Aufgabe 3: Module

Ein Programmieranfänger hat ein Programm geschrieben, mit dem die in einer Textdatei enthaltenen Wörter und Zahlen extrahiert und in einem Array verfügbar gemacht werden. Ein Wort darf Klein- und Großbuchstaben, eine Zahl Vorzeichen und Ziffern enthalten. Alle anderen Zeichen (bis auf das terminierende `'\0'`-Zeichen) werden ignoriert. Jedes gefundene Wort bzw. jede gefundene Zahl wird als Zeichenkette zusammen mit einer Typangabe (ist es ein Word oder eine Zahl?) in einer `Token`-Struktur gespeichert. Die `Token` werden in der Reihenfolge, in der sie im Text auftreten, in einem `TokenArray` abgelegt. Leider hat der Programmieranfänger alles in eine einzige Datei geschrieben (`wordsnums_all.c`). Ihre Aufgabe besteht darin, das Programm zu modularisieren, also in mehrere Implementierungsdateien (.c) und Headerdateien (.h) aufzuteilen. Das Programm kann bereits kompiliert und ausgeführt werden:

```
make wordsnums_all && ./wordsnums_all example1.txt
```

a) Teilen Sie `wordsnums_all.c` in mehrere Module auf. Ein Modul wird hier verstanden als separat kompilierte Implementierungsdatei mit zugehöriger Headerdatei. In der .h-Datei soll die öffentlich sichtbare Schnittstelle des Moduls untergebracht werden (Funktionsdeklarationen, Typdefinitionen). In der .c-Datei sollen die Details der Implementierung platziert werden. Insbesondere sollen die Definitionen der Strukturen `Token` und `TokenArray` nicht in der Header-Datei sichtbar sein. Der Zugriff auf die Elemente soll nur durch geeignete Funktionen (wie z.B. `token_value`) erfolgen, die in der jeweiligen Header-Datei deklariert sind. Diese Zugriffsfunktionen sind in `wordsnums_all.c` bereits implementiert, aber in der `main`-Funktion nicht konsistent verwendet worden. Zur Modularisierung sollen die zu `Token` und `TokenArray` gehörigen Funktionen in das Modul `token_array.{c|h}` verschoben werden. Die Funktionen, die sich auf das Einlesen des Textes und die Bildung der Tokens beziehen (z.B. `parse_text`) sowie die `main`-Funktion sollen in das Modul `wordsnums.{c|h}` verschoben werden. Da dann die Strukturdefinitionen im Modul `wordsnums` nicht mehr sichtbar sind, muss in der `main`-Funktion die durch die Zugriffsfunktionen gegebene Schnittstelle des Moduls `token_array` verwendet werden.

b) Erläutern Sie als Kommentar im Quelltext, wie in `token_array_create` der dynamische Speicher verwendet wird.

c) Erläutern Sie als Kommentar im Quelltext, die Arbeitsweise der Funktion `skip` und die Bedeutung ihres zweiten Parameters.

d) Vereinfachen Sie die Funktion `count_tokens` durch Verwendung der (bereits implementierten) Hilfsfunktionen `eos`, `skip` und `is_*_char`.

e) Erweitern Sie das Modul `wordsnums` durch einen weiteren Token-Typ `LineBreak`, der einen Zeilenumbruch (`'\n'`) repräsentiert. Nehmen Sie die notwendigen Änderungen vor, so dass im resultierenden `TokenArray` jetzt auch Zeilenumbrüche als Tokens auftauchen.

f) **(optional)** Implementieren Sie im Modul `token_index.{c|h}` die Indexierung eines bestimmten Tokentyps in einem `TokenArray`. Es soll damit möglich sein, effizient z.B. auf alle Wörter zuzugreifen, andere Token aber zu ignorieren. Dazu soll der `TokenIndex` ein int-Array enthalten, in dem die Indizes der Tokens des gewählten Typs abgelegt sind. Die Indizes beziehen sich auf die Position des Tokens im `TokenArray`. Beim Zugriff auf ein Token mit `t = token_index_get(ti, i)` soll das Zieltoken durch eine konstante Zahl von Arrayzugriffen ermittelt werden. Ohne `TokenIndex`,

wäre es notwendig, im `TokenArray` zunächst alle Token anderen Typs zu überspringen. Die Verwendung von `TokenIndex` ist im unteren Teil der `main`-Funktion zu finden. `token_index.h` enthält bereits eine mögliche öffentliche Schnittstelle (Funktionsdeklarationen und Funktionsbeschreibungen) des Moduls. Die `TokenIndex`-Struktur enthält zwei Arrays: Der Wert `type2token[i]` ist der TokenArray-Index des i-ten Tokens des indexierten Typs (z.B. das i-te Wort). Hingegen gibt `token2type[i]` an, wie viele Token des Zieltyps im `TokenArray` im Intervall [0,i] auftauchen. Diese Information ist für `token_index_next` relevant. Es sind auch andere Implementierungen denkbar. Wesentlich ist für die Lösung dieser Teilaufgabe, dass ein wahlfreier Zugriff in einer konstanten Anzahl von Schritten auf jedes beliebige Token des indexierten Typs, unabhängig von der Anzahl der Elemente im `TokenArray`, möglich ist.

Aufgabe 4: Berechnungsbaum [1 Bonuspunkt]

Die für diese Aufgabe relevanten Dateien sind computational_tree.{h|c} und comp_tree_print.{h|c}. In dieser Aufgabe sollen Sie sich mit sogenannten Berechnungsbäumen auseinandersetzen. Eine `TreeNode` enthält entweder eine Konstante, einen Parameter, dessen Wert erst zur Evaluation bekannt ist, oder eine Funktion, die ausgeführt werden soll auf ihren nachfolgenden Knoten `left` und `right` in der Form: <left> <function> <right> Bspw.: 2 + 3. Die Knoten `left` und `right` können dabei wieder Teilberechnungsbäume sein. Ein Berechnungsbaum kann wie folgt ausehen:

```
                    ( * )
          ( + )                 ( / )
     ( + )     ( * )      ( z )     ( - )
(0.0)(3.0)( x )(0.0)           ( z )(0.0)
```

Der gegebene Baum lässt sich zu dem Baum, der nur aus der Node (3.0) besteht, reduzieren.

a) Implementieren Sie die Funktion `ComputationalResult evaluate(TreeNode* comp_tree, PNode* parameter_list)`. Die Funktion bekommt einen Berechnungsbaum übergeben, sowie eine Liste von Parametern, die die Werte für Parameter enthalten. Nach Ausführung der Funktion soll in der zurückgegebenen Struktur vom Typ `ComputationalResult` entweder der Wert der Berechnung stehen und das beinhaltende `bool valid true` enthalten oder falls eine Berechnung nicht möglich ist, so soll `valid` den Wert `false` enthalten. Es sind Testfälle vorgegeben. Zum besseren Verständnis werden die Testbäume auch auf der Konsole vor Ausführung des Testfalls ausgegeben.

b) Wenn die gleichen Berechnungen auf größeren Datenmengen durchgeführt werden, dann ist es sinnvoll, vorher die Berechnungen zu optimieren. Implementieren Sie hierzu die Funktion `TreeNode* precompute_tree(TreeNode* tree)`, die einen vorberechneten Berechnungsbaum zurückgibt, der eine äquivalente Berechnung ausführt wie der übergebene Baum. Folgende Vereinfachungen sollen implementiert werden:
   - Ein Teilbaum der nur aus Konstanten und Funktionen besteht kann schon berechnet werden.
   - Ein Baum, der aus zwei gleiche Parameter in `left` und `right` hat und die Funktion - oder / ist, kann durch einen konstante Node ersetzt werden. Bsp.: x – x = 0.0 bzw. x / x = 1.0

- Ein Baum, der einen Parameter und eine Konstante hat, kann unter folgender Bedingung vereinfacht werden:
  - Für * gilt a * 0 = 0, a * 1 = a (gilt auch, wenn Parameter und Konstante getauscht werden)
  - Für + gilt a + 0 = a (gilt auch, wenn Parameter und Konstante getauscht werden)
  - Für – gilt a – 0 = a (gilt nur in dieser Reihenfolge)
  - Für / gilt a / 1 = a, 0 / a = 0 (gilt nur in dieser Reihenfolge)

c) (optional) Was ist die maximal mögliche Baumtiefe, die die Funktion `print_tree` ausgeben kann? Hinweis: Es geht hier nicht um die Limitierung der Konsolenfensterbreite.

d) (optional) Implementieren Sie eine Funktion, die einen Baum in einen String umwandelt. Die Funktion soll einen Pointer auf die Zeichenkette zurückgeben und nur den Baum als Parameter erhalten. Die Ausgabe in der dynamisch allokierten Zeichenkette soll die folgende Form aufweisen: `(left function right)`, wobei `left` und `right` wieder geklammerte Ausdrücke sind, wenn Sie nicht ein Parameter oder eine Konstante sind. Berechnen Sie als erstes, wie viele Zeichen Sie benötigen, allokieren Sie dann den Speicher. Konstanten sollen immer nur mit einer Nachkommastelle ausgeben werden. Schreiben Sie dann die String Repräsentation des Baumes in den Speicher. Beispiel:

```
        ( + )
   ( * )     ( * )
(3.0)( y )( x )(4.0)
```

Wird zu:

```
((3.0 * y) + (x * 4.0))
```

Hinweise zum Editieren, Compilieren und Ausführen:

- mit Texteditor `file.c` editieren und speichern
- `make file` ← ausführbares Programm erstellen
- `./file` ← Programm starten (evtl. ohne `./`)
- Die letzten beiden Schritte lassen sich auf der Kommandozeile kombinieren zu: `make file && ./file`