Fakultät IV - Elektrotechnik und Informatik
Einführung in die Programmierung WS 2014/15
Feldmann / Rost / Streibelt / Lichtblau / Deimel / Lucas

Aufgabenblatt 6

letzte Aktualisierung: 10. Dezember, 16:27 Uhr

(e4eaa9c34d064aec0db9d68197232b9a5a688902)

Ausgabe: Mittwoch, 10.12.2014
Abgabe: spätestens Freitag, 19.12.2014, 18:00

Thema: *Trees, tree traversal, reverse polish notation*

Abgabemodalitäten

- Alle abzugebenden Quelltexte müssen ohne Warnungen und Fehler auf den Rechnern des tubIT/IRB mittels `gcc -std=c99 -Wall` kompilieren.
- Abgaben erfolgen prinzipiell immer in Gruppen à 2 Personen, welche in den Tutorien festgelegt wurden. Einzelabgaben sind explizit als solche gekennzeichnet.
- Die Abgabe erfolgt ausschließlich über SVN. Die finale Abgabe
  - für Gruppenabgaben erfolgt im Unterordner
    `Tutorien/t<xx>/Gruppen/g<xx>/Blatt<xx>/submission/`
  - für Einzelabgaben erfolgt im Unterordner
    `Tutorien/t<xx>/Studierende/<tuBIT-Login>/Blatt<xx>/submission/`
- Benutzen Sie für alle Abgaben von Programmcode das folgende Namensschema: `introprog_blatt0X_aufgabe0Y_Z.c`, wobei X durch die Blattnummer, Y durch die Aufgabe und Z durch die Unteraufgabe ersetzen ist.
  Beispiel: `introprog_blatt01_aufgabe01_1.2.c`
  Für jede Unteraufgabe geben Sie maximal eine Quellcodedatei ab, es sei denn, die Aufgabenstellung erfordert explizit die Abgabe mehrerer Dateien pro Aufgabe.

  Alle anderen Abgaben (Pseudocode, Textaufgaben) benennen Sie wie oben beschrieben. Die zugelassenen Abgabeformate sind PDF, ODT und Text (txt). Auch hier verwenden Sie eine Datei pro Aufgabe.

1. Aufgabe: Implementieren eines Taschenrechners (22 Punkte)

In dieser Aufgabe programmieren Sie einen einfachen Taschenrechner, der Addition, Subtraktion und Multiplikation auf Fließkommawerten beherrschen soll. Um das Einlesen der Eingabe einfach zu gestalten, verwendet dieser anstatt der geläufigen Infix Notation die auch als *Reverse Polish Notation* bekannte Postfix Notation:

Postfix Notation: $38\ 4\ +$
Infix Notation: $38 + 4$

Diese Notation bietet den Vorteil, dass die Eingabe schrittweise mit Hilfe eines Stacks abgearbeitet werden kann und keine Klammern benötigt werden, um Eindeutigkeit zu gewährleisten.

Postfix Ausdruck	equivalenter Infix Ausdruck		Wert
`1 2 3 + +`	`1 + (2 + 3)`	=	6
`1 2 + 3 +`	`(1 + 2) + 3`	=	6
`1 2 3 + *`	`1 * (2 + 3)`	=	5
`3.1415 2 3 - *`	`3.1415 * (2 - 3)`	=	-3.1415
`4.0 0.2 + 3.0 * 5 +`	`((4.0 + 0.2) * 3.0) + 5`	=	17.6

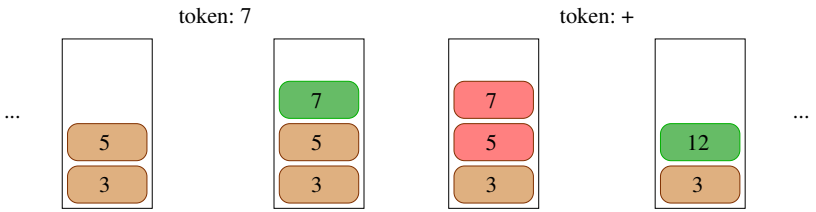

Abbildung 1: Stack während der Berechnung von 3 5 7 + +
Grüne Zahlen werden in dem Schritt auf dem Stack abgelegt und rote Zahlen vom Stack genommen.

Ein Ausdruck in Postfix Notation kann mit zwei Regeln berechnet werden:

1. Ein eingelesender Operand wird auf den Stack gelegt
2. Ein eingelesener Operator wird auf die obersten Elemente des Stacks angewendet, diese Operanden werde vom Stack entfernt und das neue Ergebnis wieder auf den Stack gelegt.
   In dieser Aufgabe verwenden wir nur die binären Operatoren `+`, `-` und `*`, d.h. diese Operatoren werden immer auf die obersten beiden Elemente auf dem Stack angewendet. Das Ergebnis wird dann wieder auf den Stack gelegt.

Ihr Programm soll dabei die folgenden Bedingungen erfüllen:

- **Ein- und Ausgabe:**
  In Ihrem Programm soll die Eingabe über die Standardeingabe erfolgen.
  Als Beispiel, der String `10 4 6 3 + + -` wird als eine Zeile dem Programm übergeben.

  Die Ausgabe des Wertes erfolgt dann mit dem Formatstring `"%4.3f"` über die Standardausgabe. Die Ein- und Ausgabe erfolgt zeilenweise. Fehlerhafte Eingaben erzeugen den Ausgabestring `Error`.

- **Funktionen:**
  Programmieren Sie drei Funktionen `stack_push(stack_top *stacktop, float value)`, `stack_pop(stack_top *stacktop)` und `void process(stack_top *stacktop, char* token)` und die dazugehörige Datenstruktur `struct sStackElement`, die zusammen einen Stack für Fließkommawerte implementieren.

- **Codekonventionen:**
  Verwenden Sie **keine** externen Bibliotheken, die gelinkt werden müssen (`-l` Flag).
  Strukturieren Sie Ihr Programm mit Hilfe von Funktionen!
  Geben Sie diesen aussagekräftige Namen und erklären deren korrekte Verwendung in Form von Kommentaren.

  Geben Sie insbesondere in Form von Kommentaren an, welche Argumentwerte erwartet werden, und welche Bedeutung und Eigenschaften die Rückgabewerte besitzen.

- **Stack:**
  Implementieren Sie den Stack so, dass er beliebig tief werden kann. Orientieren Sie sich dabei an den Datenstrukturen der letzten Aufgaben.

Verwenden Sie folgende Vorgabe und fügen Sie Ihren Code an den entsprechenden Stellen ein:

Listing 1: Vorgabe `introprog_blatt06_aufgabe01_vorgabe.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <malloc.h>
5  #include <string.h>
6  #include <math.h>   // Definiert den speziellen Fließkommawert NAN ("Not A
       ↪ Number")
7
8  struct sStackElement{
9      // Hier Code einfügen
10 };
11
12 /*
13   Der Typ stack_top ist ein pointer auf ein Stack Element
14 */
15 typedef struct sStackElement* stack_top;
16
17 /*
18    Lege Element auf den Stack
19
20    Wenn *stacktop auf NULL zeigt, ist kein Element auf dem Stack
21 */
22 void stack_push(stack_top *stacktop, float value)
23 {
24     // Hier Code einfügen
25 }
26
27 /*
28    Nimm Element auf den Stack
29
30    Wenn stack leer wird, setze *stacktop auf NULL
31
32    Ist der Stack leer, gib den Fließkommawert NAN zurück
33 */
34 float stack_pop(stack_top *stacktop)
35 {
36     // Hier Code einfügen
37     return NAN;
38 }
39
40 /*
41    Identifiziere das Token. Unterscheide folgende Fälle:
42    - Das Token ist eine Zahl: Lege es auf den Stack.
43    - Das Token ist ein Operator (+, -, *):
44        1. Nimm die obersten beiden Stackelemente
45        2. Wende den Operator auf diese Werte an
46        3. Lege das Resultat auf dem Stack ab
47
48    Implementiere hier die Rechenoperationen (+, -, *) und lege das
49    Ergebnis zurück auf den Stack. Beachte, dass du mit Floatingpointwerten
50    arbeitest, z.B. auch negative Kommazahlen.
51 */
52 void process(stack_top *stacktop, char* token)
53 {
54     assert(token != NULL);
55     // Hier Code einfügen
56 }
57
58
59 const int MAX_LINE_LENGTH=1024;
60 /*
61  main() liest die Eingabe zeilenweise ein und zerlegt diese in einzelne
       ↪ getrennte Zeichenketten (Token)
62
63  Diese Funktion muss nicht geändert werden.
64 */
65 int main(int argc, char** args)
66 {
67     stack_top stacktop = NULL;
68     char line[MAX_LINE_LENGTH]; // Eingabezeile
69     char* token;                // Pointer auf das aktuellen token;
70
71     while ( fgets(line, MAX_LINE_LENGTH, stdin) ) {
72         token = strtok(line, "␣");
73
74         while (token != NULL) {
75             process(&stacktop, token);  // Stackoperationen durchführen
76             token = strtok(NULL, "␣");  // Neues Token einlesen
77         }
78
79         float result = stack_pop(&stacktop);    // Letztes Ergebnis vom Stack
               ↪ holen
80
81         if (stacktop != NULL) { // Mehr Operanden als benötigt angegeben.
```

```
            ↪ Fehler.
82              while (stacktop != NULL) {
83                  stack_pop(&stacktop);   //Räume Stack auf
84              }
85              printf("Error\n");
86          } else if (result != result) {  // result ist NAN: Berechnung
               ↪ fehlgeschlagen  (z.b. zu wenig Operanden)
87              printf("Error\n");
88          } else {
89              printf("%4.3f\n", result); // Gib Resultat aus
90          }
91      }
92  }
```

Hinweis: Diese Aufgabe verwendet einige Funktionen, die Sie so noch nicht kennengelernt haben. Die Erklärungen dazu finden Sie in den meisten Fällen mit dem folgendem Befehl in der Kommandozeile:

```
# man <Funktionsname>
```

Die für diese Aufgabe relevanten Funktionen werden in der Manpage wie folgt kurz erklärt (für detailierte Informationen schauen Sie bitte in der manpage nach):

`char *fgets(char *s, int size, FILE *stream);`
fgets() liest höchstens *size* minus ein Zeichen von *stream* und speichert sie in dem Puffer, auf den *s* zeigt. Das Lesen stoppt nach einem **EOF** oder Zeilenvorschub. Wenn ein Zeilenvorschub gelesen wird, wird er in dem Puffer gespeichert. Ein **'\0'** wird nach dem letzten Zeichen im Puffer gespeichert.

`char *strtok(char *s, const char *trenner);`
extrahiert Token aus der Zeichenkette *s*, die durch eines der Byte in *trenner* unterteilt werden.

`double atof(const char *nptr);`
Die Funktion **atof()** wandelt den Anfang der Zeichenkette, auf die *nptr* zeigt, in eine Double-Zahl um.

Geben Sie den Quelltext in einer Datei mit folgendem Namen ab:
`introprog_blatt06_aufgabe01.c`

2. Aufgabe: Binärbäume (6 Punkte)

Geben Sie Ihre Lösung der Aufgabe in einem Textdokument mit einem der folgenden Namen ab:
`introprog_blatt06_aufgabe02.{txt|odt|pdf}`

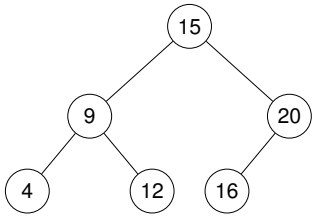

Abbildung 2: Binärbaum

2.1. In-order-tree-walk (2 Punkte) Durchlaufen Sie den gegebenen Baum nach dem in der Vorlesung vorgestellten Verfahren. Geben Sie die Elemente in der Reihenfolge an, in der sie besucht werden.

Geben Sie Ihre Lösung der Teilaufgabe in einem Textdokument mit einem der folgenden Namen ab:
`introprog_blatt06_aufgabe02_1.{txt|odt|pdf}`

2.2. Einfügen in Bäume (4 Punkte) Welcher Baum ergibt sich nachdem Sie die folgenden Elemente in der gegebenen Reihenfolge in den Baum einfügen?

(11), (22), (15), (21)

Zeichnen Sie den resultierenden Baum.

3. Aufgabe: Beweise auf Bäumen (10 Punkte)

3.1. Beweis: Knotenanzahl der k-ten Schicht (5 Punkte) Beweisen Sie mittels vollständiger Induktion: Die Zahl der Knoten in der k-ten Schicht eine Binärbaums ist höchstens 2^{k-1}.

Geben Sie Ihre Lösung der Teilaufgabe in einem Textdokument mit einem der folgenden Namen ab:
`introprog_blatt06_aufgabe03_1.{txt|odt|pdf}`

3.2. Beweis: Anzahl Knoten in einem Baum der Höhe h (5 Punkte) Beweisen Sie mit Hilfe von 3.1: Die Zahl der Knoten in einem Binärbaum der Höhe h ist höchstens $2^h - 1$.

Geben Sie Ihre Lösung der Teilaufgabe in einem Textdokument mit einem der folgenden Namen ab:
`introprog_blatt06_aufgabe03_2.{txt|odt|pdf}`