Fakultät IV Elektrotechnik und Informatik
Einführung in die Programmierung WS 2021/2022
Manfred Hauswirth / Damien Foucard / Christoph Seifert / Uwe Kuehn

Programmierblatt 05

Ausgabe: *02.12.2021 12:00*
Abgabe: *14.12.2021 08:00*

Thema: *Stacks, Reverse Polish Notation*

Abgabemodalitäten

1. Alle abzugebenden Quelltexte müssen ohne Warnungen und Fehler auf Deinem Rechner mit dem Befehl `clang -std=c11 -Wall -g` kompilieren.
2. Die Abgabe für den Quellcode erfolgt ausschließlich über unser Git im entsprechenden Branch. Nur wenn ein Ergebnis im ISIS-Kurs angezeigt wird, ist sichergestellt, dass die Abgabe erfolgt ist. Die Abgabe ist bestanden, wenn Du an Deinem Test einen grünen Haken siehst.
3. Du kannst bis zur Abgabefrist beliebig oft neue Versionen abgeben. Lies Dir die Hinweise der Tests genau durch, denn diese helfen Dir Deine Abgabe zu korrigieren.
Bitte beachte, dass ausschließlich die letzte Abgabe gewertet wird.
4. Die Abgabe erfolgt, sofern nicht anders angegeben, in folgendem Branch: `iprg-b<xx>-a<yy>`, wobei `<xx>` durch die zweistellige Nummer des Aufgabenblattes und `<yy>` durch die entsprechende Nummer der Aufgabe zu ersetzen sind.
5. Gib für jede Aufgabe die Quellcodedatei(en) gemäß der Vorgabe ab. Im ISIS-Kurs werden zum Teil Vorgabedateien bereitgestellt. Nutze diese zur Lösung der Aufgaben.
6. Die Abgabefristen werden vom Server überwacht. Versuche Deine Abgabe so früh wie möglich zu bearbeiten. Du minimierst so auch das Risiko, die Abgabefrist auf Grund von „technischen Schwierigkeiten“ zu versäumen. Eine Programmieraufgabe gilt als bestanden, wenn alle bewerteten Teilaufgaben bestanden sind.

Aufgabe 1 Postfix Taschenrechner (bewertet)

In dieser Aufgabe sollst Du einen einfachen Taschenrechner implementieren, der Addition, Subtraktion und Multiplikation auf Fließkommawerten beherrscht. 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 die Ausdrücke auch ohne Klammern eindeutig sind.

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

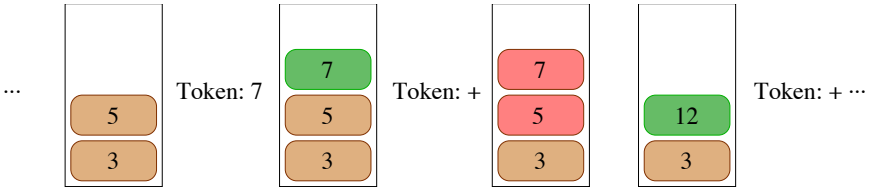

Abbildung 1: Stack während der Berechnung von “3 5 7 + +”
Grüne Zahlen werden in dem Schritt auf dem Stack abgelegt und rote Zahlen vom Stack genommen.

Die Berechnung eines in Postfix Notation geschriebenen Ausdrucks erfolgt wie folgt. Zunächst wird der als Zeichenkette bestehende Ausdruck (z.B. “`0.3 -2 +`”) in die von Leerzeichen getrennte Abschnitte unterteilt (“`0.3`”, “`-2`” und “`+`”). Jede dieser Einheiten, die selbst wieder eine Zeichenkette ist, bezeichnen wir als ein *Token*. Die Token werden dabei nacheinander, von links nach rechts, verarbeitet:

- Wenn es sich bei dem Token um eine, als Zeichenkette repräsentierte, Zahl handelt, dann konvertiere die Zahl in ein Zahlenformat (siehe `atof`) und platziere sie auf dem Stack (engl. push).
- Wenn es sich bei dem Token um einen Operator (`+`, `-` und `*`) handelt, dann nehme die obersten zwei Elemente vom Stack (engl. pop), führe die dem Operator entsprechende Operation aus und lege das Resultat der Operation wieder auf den Stack.
  - Wenn der Stack zur Zeit der Abfrage keine Elemente enthält, dann gibt er ein `NAN` (“not-a-number”) zurück. `NAN` ergibt bei jeder Berechnung (d.h., unter Verwendung jegliches Operators auf `NAN` und jegliche andere Zahl) ebenfalls ein `NAN` und ermöglicht es damit, Fehler schnell zu finden.
- Wenn es sich bei dem Token um ein anderes (als die genannten) Zeichen (oder Zeichenketten, bspw. Kommazahlen) handelt, dann ignoriere das Token und fahre mit dem nächsten Token fort.

Das Programm endet, wenn alle Token abgearbeitet wurden.

Um die Programmierung zu vereinfachen, muss die Vorgabe verwendet werden. Zusätzlich muss die Bibliothek `introprog_input_stacks-rpn.c` und `introprog_input_stacks-rpn.h` wie angegeben eingebunden werden. (Da die Ausgabe zu umfangreich ist, wird sie hier nicht gezeigt.)

Listing 1: Programmbeispiel

```
> clang -std=c11 -Wall introprog_stacks-rpn.c \
      introprog_input_stacks-rpn.c -o introprog_stacks-rpn
> ./introprog_stacks-rpn
```

Zusätzlich benötigst Du die Funktion `double atof(const char* string)`. Sie wandelt die Zeichenkette, auf die `string` zeigt, in eine Zahl des Formats `double` um.

Listing 2: Beispiel für atof

```
char* string = "-1.2";
float number = atof(string);
printf("String: %s Zahl: %d\n", string, number);
```

Dein Programm soll die folgenden Bedingungen erfüllen:

- **Funktionen:**
  Programmiere die folgenden vier Funktionen und verändere **nicht** die anderen Funktionen in der Vorgabe:
  - `stack_push(stack* astack, float value)`
    Füge Element mit dem Wert `value` am Anfang des Stacks ein.
  - `stack_pop(stack* astack)`
    Nehme das zuletzt eingefügte Element vom Stack und gebe den enthaltenen Wert zurück. Gebe `NAN` zurück, wenn der Stack leer ist.
  - `stack* stack_erstellen()`
    Erstelle einen Stack (dynamische Speicherreservierung & Initialisierung) und gebe einen Pointer auf den Stack zurück.
  - `void process(stack* astack, char* token)`
    Verarbeite die Token wie oben beschrieben.
- **Datenstrukturen:**
  Nutze die vorhandenen Datenstrukturen aus der Vorgabe.
- **Speicherverwaltung:**
  Der Speicher soll zum Ende des Programms vollständig freigegeben sein.

Hinweise:

- Diese Aufgabe verwendet gerade in der Vorgabe einige Funktionen, die Du noch nicht kennengelernt hast. Falls Du wissen möchtest, was diese tun, so kannst Du das mit dem folgendem Befehl in der Kommandozeile in Erfahrung bringen:

  ```
  > man <Funktionsname>
  ```

- In dieser Aufgabe wird der Wert `NAN` verwendet. Diese Nicht-Zahl wird in der Bibliothek `math.h` definiert. Für eine beliebige Zahl x gilt: NAN $+ x =$ NAN, NAN $- x =$ NAN und NAN $* x =$ NAN. Diese Eigenschaft machen wir uns in dieser Aufgabe zu Nutze, um Fehler zu finden. Allerdings hat `NAN` noch eine weitere Eigenschaft: NAN $\neq$ NAN, d.h. der Wert `NAN` ist nicht nur ungleich jeder Zahl, sondern unterscheidet sich ebenso von sich selbst. Diese Eigenschaft benötigen wir in dieser Aufgabe nicht, aber sie kann, wenn nicht beachtet, zu Problemen führen.

Nutze zur Lösung der Aufgabe die Vorgaben aus unserem ISIS-Kurs. Füge Deine Lösung als Datei `introprog_stacks-rpn.c` im entsprechenden Abgabebranch in Dein persönliches Repository ein und übertrage die Lösung an die Abgabeplattform.