Fakultät IV Elektrotechnik und Informatik
Einführung in die Programmierung WS 2023/2024
Manfred Hauswirth
Damien Foucard / Uwe Kuehn / Aljoscha Meyer

Programmierblatt 04

Ausgabe: *30.11.2023 16:00*
Abgabe: *08.12.2023 20:00*

Thema: *Stacks, Reverse Polish Notation*

Abgabemodalitäten

1. Alle abzugebenden Quelltexte müssen ohne Warnungen und Fehler auf Ihrem 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 Sie an Ihrem Test einen grünen Haken sehen.
3. Sie können bis zur Abgabefrist beliebig oft neue Versionen abgeben. Lesen Sie sich die Hinweise der Tests genau durch, denn diese helfen Ihnen die Abgabe zu korrigieren.
Bitte beachten Sie, 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. Geben Sie für jede Aufgabe die Quellcodedatei(en) gemäß der Vorgabe ab. Im ISIS-Kurs werden zum Teil Vorgabedateien bereitgestellt. Nutzen Sie diese zur Lösung der Aufgaben.
6. Die Abgabefristen werden vom Server überwacht. Versuchen Sie Ihre Abgabe so früh wie möglich zu bearbeiten. Damit minimieren Sie auch das Risiko, die Abgabefrist auf Grund von „technischen Schwierigkeiten“ zu versäumen. Eine Programmieraufgabe gilt als bestanden, wenn alle bewerteten Teilaufgaben bestanden sind.
7. Sofern die Aufgabenstellenstellung nichts gegenteiliges besagt, dürfen keine weiteren `include` Direktiven verwendet werden, d.h., es dürfen keine zusätzlichen Bibliotheksfunktionen verwendet werden. Eigene Funktionen zu implementieren und verwenden ist hingegen legitim und häufig eine gute Idee für besser lesbaren Code.

Aufgabe 1 Postfix Taschenrechner (bewertet)

In dieser Aufgabe sollen Sie einen einfachen Taschenrechner implementieren, der Addition, Subtraktion und Multiplikation auf Fließkommawerten beherrscht. Um das Einlesen der Eingabe zu vereinfachen, verwendet dieser anstatt der Infix Notation die Postfix[1] Notation:

Postfix Notation: 38 4 +
Infix Notation: 38 + 4

Diese Notation bietet die Vorteile, dass die Eingabe schrittweise mit Hilfe eines Stacks abgearbeitet werden kann und die Ausdrücke auch ohne Klammern eindeutig an die Operatoren gebunden 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

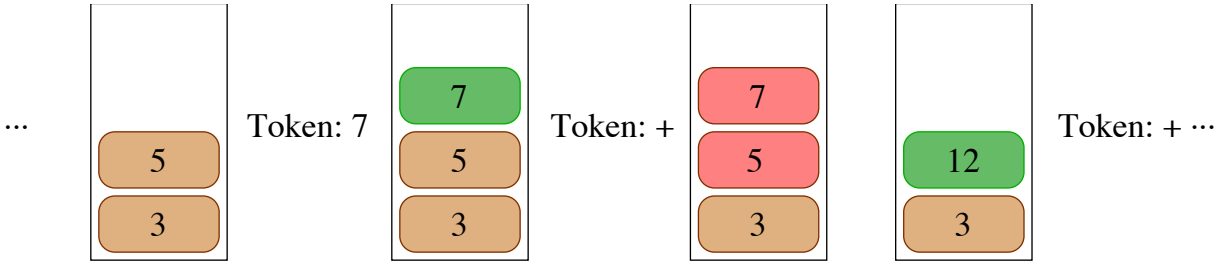

Abbildung 1: Stack während der Berechnung von „3 5 7 + +“
Grüne Zahlen werden in dem Schritt auf dem Stack abgelegt und rote Zahlen vom Stack genommen.

Ein in Postfix Notation geschriebener Ausdruck kann nun mit den folgenden Schritten berechnet werden. Zunächst wird der als Zeichenkette bestehende Ausdruck (z.B. „`0.3 -2 +`“) in die von Leerzeichen getrennten Zeichenketten („`0.3`“, „`-2`“ und „`+`“) unterteilt. Diese *Token* werden dabei nacheinander von links nach rechts bei Unterscheidung folgender Fälle verarbeitet:

[1] aka Reverse Polish Notation

- Fall 1: Wenn es sich bei dem Token um eine (Fließkomma-)Zahl handelt, dann konvertiere die Zeichenkette in eine Zahl (siehe `atof()`) und platziere sie mit `push()` auf dem Stack.
- Fall 2: Wenn es sich bei dem Token um einen Operator (`+`, `-` und `*`) handelt, dann nehme die obersten zwei Elemente mit `pop()` vom Stack, führe die zu dem Operator gehörende Operation auf diesen beiden Operanden aus und lege das Resultat der Operation wieder auf den Stack.
- Sonderfall: Wenn der Stack zur Zeit der Abfrage keine Elemente enthält, dann gibt `pop()` ein `NAN` („not-a-number“) zurück. `NAN` ergibt bei jeder Berechnung (d.h. unter Verwendung beliebiger Operatoren auf Operanden die mindestens ein `NAN` enthalten) ebenfalls ein `NAN` und ermöglicht es damit, Fehler schnell zu finden (siehe auch Hinweise).
- Ignoriere alle anderen 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. Der Quellcode ist wegen des Umfanges hier nicht abgebildet.

Listing 1: Programmbeispiel

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

Wie bereits angedeutet, benötigen Sie 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);
```

Ihr Programm soll des Weiteren die folgenden Bedingungen erfüllen:

- **Funktionen:**
  Implementieren Sie die folgenden vier Funktionen und verändern Sie die anderen Funktionen in der Vorgabe **nicht**:
  - `stack_push(stack*, float)`
    Füge Element mit dem Wert `value` am Anfang des Stacks ein.
  - `stack_pop(stack*)`
    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*, char*)`
    Verarbeite die Token wie oben beschrieben.
- **Datenstrukturen:**
  Nutzen Sie 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 Sie noch nicht kennengelernt haben. Falls Sie wissen möchten, was diese tun, so können Sie das mit dem folgendem Befehl[2] 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: $\text{NAN} + x = \text{NAN}$, $\text{NAN} - x = \text{NAN}$ und $\text{NAN} * x = \text{NAN}$. Diese Eigenschaft machen wir uns in dieser Aufgabe zu Nutze, um Fehler zu finden. Allerdings hat `NAN` noch eine weitere Eigenschaft: $\text{NAN} \neq \text{NAN}$, 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.

Nutzen Sie zur Lösung der Aufgabe die Vorgaben aus unserem ISIS-Kurs. Fügen Sie Ihre Lösung als Datei `introprog_stacks-rpn.c` im entsprechenden Abgabebranch in Ihrem persönliches Repository ein und übertragen Sie die Lösung an die Abgabeplattform.

[2] man erfordert, dass für die entsprechenden Befehle und Funktionen die man-pages installiert sind