Vorkus
Programmierung 2 (SS 2021)
Universität des Saarlandes
Fakultät MI

Blatt 3
Projekttag 3

Daniel Weber
Jessica Schmidt,
Luise Puhl,
Pascal Lauer

Der Vorkurs ist ein Angebot der (teilweise ehemaligen) Programmierung 2 Tutoren. Der Kurs ist keine offizielle Lehrveranstaltung. Es gibt keine CP, die Teilnahme ist optional, und eine HISPOS-Anmeldung ist weder möglich noch erforderlich. Bei Fragen zum Vorkurs und Programmierung 2 könnt ihr euch gerne an die Dozenten und Tutoren wenden. Wir wünschen euch ein erfolgreiches Semester und freuen uns, euch in Programmierung 2 wieder zu sehen.

Aufgabe 3.0: Stack

In dieser Aufgabe geht es darum, einen einfachen Stack für `direction` zu implementieren. Wie ein Stack funktioniert, haben Sie bereits in der Vorlesung gelernt. In unserem Projekt verwenden wir das globale Array, um den Speicher, in dem wir den Stack speichern, zu simulieren. Die "Adresse" des Stacks ist daher eine Position im Array, im Folgenden `stack_base` genannt. An dieser Position speichern wir den `stack_pointer`. Dieser beinhaltet die Position, an der das oberste Element auf dem Stack liegt. Jede Operation bekommt als Argument 0 die `stack_base` übergeben. Die folgenden Operationen soll der Stack unterstützen:

a) `StackInit`: Initialisiert den `stack_pointer`.

b) `StackPush`: Legt das Element aus Argument 1 auf den Stack.

c) `StackPop`: Löscht das aktuelle Element vom Stack und gibt es zurück.

d) `StackEmpty`: Gibt `1` zurück, falls der Stack leer ist, sonst `0`.

e) `StackPeek`: Gibt das oberste Element vom Stack zurück, ohne es zu entfernen. Implementieren Sie `StackPeek` auf zwei Arten:

   (a) Greifen Sie nicht direkt auf das Array zu, indem Sie die anderen Operationen des Stacks verwenden.

   (b) Implementieren Sie `StackPeek` ohne die anderen Operationen zu verwenden.

   Welchen Vorteil und welchen Nachteil hat die erste Implementierung gegenüber der zweiten?

Hinweis:
Rückgabewerte können mit `set_arg position value` gesetzt und mit `get_arg variable position` vom Aufrufer gelesen werden.

Aufgabe 3.1: Tiefensuche

In dieser Aufgabe implementieren Sie eine Tiefensuche (Depth-First-Search, dfs), um das Labyrinth zu lösen.

a) Schreiben sie ein Unterprogramm `move_to`, welches eine Himmelsrichtung als Argument erhält und dann die Eule in diese Himmelsrichtung bewegt. Dabei sollen jeweils der genutze Ausgang des Startfeldes, sowie der genutze Eingang des Zielfeldes markiert werden.

b) Schreiben sie ein Unterprogramm `get_next_move`, welches die StartAddresse eines Stacks als Argument erhält und die nächste Bewegungsrichtung der Eule wie folgt berrechnet:

- Falls es ein betretbares Nachbarfeld gibt **und** der Weg zu diesem Feld nicht markiert ist, gehe in diese Richtung. (Es ist prinzipiell egal in welcher Reihenfolge die Nachbarfelder überprüft werden, allerdings empfehlen wir die Reihenfolge: **Norden Osten**, **Süden**, **Westen**, da dadurch der Test aussagekräftiger wird).
- Falls es kein solches Feld gibt, gehe auf das Feld zurück, von dem aus das aktuelle Feld betreten wurde.

c) Implementieren sie den Algorithmus Tiefensuche entweder durch Zusammenführen der Unterprogramme aus a) & b) oder mit einer alternativen Herangehensweise. Sie dürfen für die Lösung die Tatsache benutzen, dass ihr Algorithmus automatisch terminiert, wenn die Eule das Zielfeld betritt.

Aufgabe 3.2: **Bonus:** Queue

In dieser Aufgabe geht es darum eine Queue mithilfe von Stacks zu implementieren. Eine Queue funktioniert nach dem FIFO-Prinzip (First in, first out). Ihre Queue soll folgende Operationen unterstützen: `QueueInit`, `QueuePush`, `QueuePop`, und `QueueEmpty`, wobei die Argumente und return-Werte denen des Stacks entsprechen.