Funktionale Programmierung

E-Portfolio Julian Glänzer

Dieses E-Portfolio dient als Prüfungsleistung für das Modul "Funktionale Programmierung" an der Technischen Hochschule Mittelhessen (THM)

Aufgabenblock I

Dieser Aufgabenblock wurde in Zusammenarbeit mit Daniel Spahn und Ari Diehl erstellt.

Aufgabe 1

> Welche Ausdrücke sind strikt? Begründen Sie Ihre Antwort.

```
(define (square x) (* x x))

; 1
(square 9)

; 2
(if (= 10 2) (square 10) (square 0))

; 3
(and (= 10 2) (< 2 10))
```

1. `(square 9)` ist **strikt**, da das Argument nicht ausgewertet werden kann, da es ein Literal ist.
2. `(if (= 10 2) (square 10) (square 0))` ist **nicht strikt**, da die test-expr `#f` ist und somit nur die else-expr `(square 0)` ausgewertet wird.
3. `(and (= 10 2) (< 2 10))` ist **nicht strikt**, da `(= 10 2)` `#f` zurückgibt was dazu führt, dass `and` die Auswertung abbricht. Das zweite Argument wird somit nicht ausgewertet.

Aufgabe 2

> Welche der folgenden Ausdrücke sind Listen? Begründen Sie Ihre Antwort.

```
1: '(1 (2 3) . 4)

2: (list 1 2 3)

3: (cons 1 (cons 2 (cons 3 '())))

4: (cons 'a 'b)
```

1. Keine Liste, da das zweite Element keine Liste ist
2. Liste, da diese Funktion eine gültige Liste mit den gegebenen Parametern erstellt
3. Liste, weil es eine Verkettung von Elementen darstellt, die alle mit "cons" verbunden sind. Das erste Element ist 1, gefolgt von der Liste (2 3). Das letzte Element der Liste ist eine leere Liste.
4. (cons 'a 'b) ist keine Liste. Dieser Ausdruck repräsentiert ein Paar, dessen erste Element a ist und dessen zweites Element b ist.

Aufgabe 3

> Gegeben folgender Code:

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))
```

> Was erhält der Nutzer, wenn er ‘(test 0 (p))‘ eingibt, bei applicative order, bei normal order?

Applicative Order
Bei der Applicative Order wird zuerst die Funktion (p) ausgewertet, bevor das Ergebnis an die Funktion (test) übergeben wird. Da (p) ein unendlicher Rekursionsprozess ist, erhält der Benutzer niemals ein Ergebnis.
Normal Order
Bei der Normal Order wird zuerst die Funktion `(test)` ausgewertet, dann werden die Argumente `0` und `(p)` übergeben. Da `0` das erste Argument ist, erhält der Benutzer `0` als Ergebnis.

Aufgabe 4

`new-if` funktioniert nicht genau so wie `if`, da bei `new-if` die Applicative Order gilt. Dies bedeutet, dass immer alle Ausdrücke ausgewertet werden, auch wenn die Test-Expression `#f` ist. Bei `if` wird die Normal Order angewendet, was bedeutet, dass nur der Ausdruck ausgewertet wird, der auch benötigt wird.

```
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
(display "IF:")
(newline)
(if #f
    (display "True Branch\n")
    (display "False Branch"))
```

```
(newline)
(display "NEW-IF:")
(newline)
(new-if #f
        (display "True Branch\n")
        (display "False Branch"))

;IF:
;False Branch
;NEW-IF:
;True Branch
;False Branch
```

Aufgabe 5

> Was macht die folgende Funktion und wie funktioniert die Auswertung:

```
(define (a-plus-abs-b a b)

((if (> b 0) + -) a b))
```

Diese Funktion addiert `a` mit dem Absolutwert von `b`. Darauf wird überprüft, ob `b` größer als 0 ist.
Falls dies wahr ist, wird b von a subtrahiert, ansonsten wird b zu a addiert.

Aufgabe 6

> Umdrehen einer Liste: Schreiben Sie eine Funktion my-reverse, die eine Liste erhält und die Elemente umdreht, also bspw. (my-reverse '(1 2 3)) führt zu (3 2 1).

```
(define (my-reverse lst)
  (cond
    ((null? lst) '())
    (else (append (my-reverse (cdr lst)) (list (car lst))))))
```

Beispiel:

```
(my-reverse '(1 2 3))
; Ausgabe: (3 2 1)
```

Aufgabe 7

> Finden in einer Liste: Schreiben Sie eine Funktion my-find, die eine Liste lst und ein Argument x erhält und mittels equal? prüft, ob das Element in der Liste enthalten ist. Die Funktion soll `#t` zurückgeben, wenn dem so ist, oder `#f` wenn nicht.

```
(define (my-find lst x)
  (cond
    ((null? lst) #f)
    ((equal? (car lst) x) #t)
    (else (my-find (cdr lst) x))))
```

Beispiel:

```
(my-find '(1 2 3 4) 4)
; Ausgabe: #t
```

Aufgabe 8

> Quadrieren der Elemente einer Liste: Schreiben Sie eine Funktion square-my-list, die eine Liste mit Zahlen erhält und eine neue Liste der Quadrate dieser Zahlen zurückliefert.

```
(define (square-my-list lst)
  (cond
    ((null? lst) '())
    (else (cons (* (car lst) (car lst))
                (square-my-list (cdr lst))))))
```

Beispiel:

```
(square-my-list '(1 2 3 4 5))
; Ausgabe: '(1 4 9 16 25)
```

Aufgabe 9

> Addieren der Werte in einer Liste: Schreiben Sie eine Funktion sum-my-list, die eine Liste mit Zahlen erhält und diese aufaddiert.

```
(define (sum-my-list lst)
  (cond
    ((null? lst) 0)
    (else (+ (car lst) (sum-my-list (cdr lst))))))
```

Beispiel:

```
(sum-my-list '(1 2 3 4 5))
; Ausgabe: 15
```

Aufgabe 10

> Schreiben Sie endrekursive Fassungen der beiden letzten Funktionen.

```
; Quadriert die Elemente einer Liste endrekursiv
(define (square-my-list lst)
  (define (f lst acc)
    (if (null? lst) acc (f (cdr lst) (append acc (list (sqr (car lst)))))))
  (f lst '()))

; Berechnet die Summe der Elemente einer Liste endrekursiv
(define (sum-my-list lst)
  (define (f lst acc)
    (if (null? lst) acc (f (cdr lst) (+ (car lst) acc))))
  (f lst 0))
```

Aufgabe 11

> Schreiben Sie map unter Verwendung von foldl/foldr (wählen Sie eine geeignete fold-Variante).

```
(define (my-map func lst)
  (foldr (lambda (x y) (cons (func x) y)) '() lst))
```

Aufgabe 12

> Wie erkennen Sie endrekursive Funktionen in Racket und welchen Vorteil haben diese?

Endrekursive Funktionen lassen sich daran erkennen, dass die letzte Operation in der Funktion der Aufruf der Funktion selbst ist.
Dies hat den Vorteil, dass weniger Stack-Frames benötigt werden. Dadurch wird Speicherplatz gespart und die Funktion kann schneller ausgeführt werden.

Aufgabe 13

> Wann liefern foldr und foldl das gleiche Ergebnis? Geben Sie jeweils ein selbst gewähltes Beispiel für beide Fälle (ergibt das gleiche, ergibt etwas anderes) an. Wie unterscheiden sich beide Funktionen in ihrem Laufzeitverhalten (Geschwindigkeit, Speicherbedarf etc.)?

`foldr` und `foldl` liefern das gleiche Ergebnis, wenn die Funktion `f` assoziativ ist. Das heißt, dass `f` für alle Elemente `x`, `y` und `z` gilt:

```
(f x (f y z)) = (f (f x y) z)
```

Beispiel:

```
(define (f x y) (+ x y))
(foldr f 0 '(1 2 3 4 5))
; Ausgabe: 15
(foldl f 0 '(1 2 3 4 5))
; Ausgabe: 15
```

`foldr` und `foldl` unterscheiden sich im Laufzeitverhalten, wobei `foldr` in der Regel schneller ist. Dies liegt daran, dass `foldr` die Liste von links nach rechts durchläuft, während `foldl` die Liste von rechts nach links durchläuft. Darüber hinaus benötigt `foldl` mehr Speicher, da es einen Teil der Liste beim Durchlaufen im Speicher speichert, während `foldr` nur einen Stack-Frame erstellt. [1]

Aufgabe 14

> Zeigen Sie an zwei selbst erstellten Beispielen die Nutzung von Funktionen höherer Ordnung auf.

Eine Funktion ist höherer Ordnung, wenn diese eine Funktion als Argument erhält oder eine Funktion zurückliefert.

```
; Berechnet die Mehrwertsteuer
(define (calculate-vat price)
  (* price 1.19))

; Beispiel:
(calculate-vat 100) ; 119
```

```
; FizzBuzz: Gibt "Fizz" aus, wenn die Zahl durch 3 teilbar ist, "Buzz", wenn sie
durch 5 teilbar ist und "FizzBuzz", wenn sie durch 15 teilbar ist. Ansonsten wird
die Zahl ausgegeben.

(define (fizzbuzz)
  (lambda (n)
    (cond
      [(= (modulo n 15) 0) "FizzBuzz"]
      [(= (modulo n 3) 0) "Fizz"]
      [(= (modulo n 5) 0) "Buzz"]
      [else n])))


((fizzbuzz) 15) ; "FizzBuzz"
((fizzbuzz) 3) ; "Fizz"
((fizzbuzz) 5) ; "Buzz"
((fizzbuzz) 7) ; 7
```

Aufgabe 15

> Überführen Sie die Funktion sum in eine endrekursive Fassung. Die Funktion wurde wie folgt definiert:

```
(define (sum term a next b)
(cond
```

```
((> a b) 0)
(else (+ (term a)
         (sum term (next a) next b)))))
```

Endrekursive Fassung:

```
(define (my-sum term a next b)
  (define (f term a next b acc)
    (cond
      [(> a b) acc]
      [else (f term (next a) next b (+ (term a) acc))]))
  (f term a next b 0))
```

Aufgabe 16

> Schreiben Sie eine Funktion ‘repeated, die eine Funktion f und eine natürliche Zahl n erhält und dann f n-mal ausführt. Beispiel:

```
((repeated sqr 2) 3)
81

((repeated add1 10) 1)
11
```

Funktion `repeated`:

```
(define (repeated f n)
  (if (= n 0)
      (lambda (x) x)
      (lambda (x) (f ((repeated f (- n 1)) x)))))
```

Aufgabenblock 2

Aufgabe 1

> Schreiben Sie eine Funktion insert-sort, die eine übergebene Liste von Zahlen mittels Sortieren durch Einfügen sortiert und das Ergebnis zurückliefert. Beispiele:

```
> (insert-sort '(2 4 1 6 4))

'(1 2 4 4 6)

> (insert-sort '())

'()
```

Funktion `insert-sort`:

```
(define (insert-sort l)
  (define (insert x l)
    (cond
      [(null? l) (list x)]
      [(< x (car l)) (cons x l)]
      [else (cons (car l) (insert x (cdr l)))]))
  (cond
    [(null? l) '()]
    [else (insert (car l) (insert-sort (cdr l)))]))
```

Aufgabe 2

> Schreiben Sie eine Funktion flatten, die eine beliebig tief verschachtelte Listen-Struktur in eine Liste "glättet":

```
> (flatten '((a) b (c (d)  e) ()))

'(a b c d e)
```

Funktion `flatten`:

```
(define (flatten lst)
  (define (f lst acc)
    (cond
      [(null? lst) acc]
      [(pair? lst) (f (car lst) (f (cdr lst) acc))]
      [else (cons lst acc)]))
  (f lst '()))
```

Aufgabe 3

> Beschreiben Sie den Unterschied zwischen der Nutzung von thunk und den Promises (delay und force).

Der Unterschied zwischen thunk und Promises besteht darin, dass thunk die übergebenen Ausdrücke bei jedem Aufruf evaluiert, während Promises die Ausdrücke nur einmal evaluieren und das Ergebnis für die zukünftigen Aufrufe zwischenspeichern.

> Wann ist das Ergebnis gleich und wann unterscheidet es sich?

Das Ergebnis von thunk und Promises ist dasselbe, aber das Ergebnis von Promises wird gecached, so dass zukünftige Aufrufe von force dasselbe Ergebnis erzeugen. Bei thunk ist das nicht der Fall, da der Code jedes Mal, wenn er aufgerufen wird, ausgeführt wird.

> Für welchen Anwendungszweck würden Sie welche Version einsetzen?

Thunks werden am besten verwendet, wenn der Code sofort ausgeführt werden muss. Promises werden am besten verwendet, wenn der Code nicht sofort ausgeführt werden muss, aber bei Bedarf auf Abruf ausgeführt werden muss. Beispiele für die Verwendung von Promises sind die Vermeidung von Code-Redundanz, die Vermeidung von Wiederholungen und die Optimierung von Performanz.

Die Verwendung von thunk ist dann besser, wenn der Output stark unterschiedlich sein kann. Ein Beispiel dafür ist die Statusabfrage eines Servers.

Aufgabe 4

> Warum kann man delay nicht als Funktion schreiben?

Die Argumente, die `delay` übergeben werden, werden nur ausgewertet, wenn `force` auf den erzeugten Promise angewandt wird. Dies entspricht dem Prinzip der *lazy evaluation*. Im Gegensatz dazu werden bei einer Funktion alle Argumente *vor* dem Funktionsrumpf ausgewertet (*applicative order*).

Aufgabe 5

> Implementierten Sie eine Funktion fib-stream, die einen Strom von zwei-elementigen Listen (n (fib(n))) erzeugt, wobei n die natürlichen Zahlen durchläuft. Die Funktion soll nicht jede Fibonacci-Zahl unabhängig berechnen.

```
> (define a (fib-stream))
> a
((0 0) . #<promise>)
> (tail a)
((1 1) . #<promise>)
> (tail (tail a))
((2 1) . #<promise>)
> (tail (tail (tail a)))
((3 2) . #<promise>)
> (tail (tail (tail (tail a))))
((4 3) . #<promise>)
> (tail (tail (tail (tail (tail a)))))
((5 5) . #<promise>)
```

Funktion `fib-stream`:

```
; Erstellt einen Stream
(define (fib-stream)
  (let fib-helper ([n 0] [prev 0] [cur 1])
    (cons (list n prev) (delay (fib-helper (add1 n) cur (+ prev cur))))))

; Gibt den ausgewerteten Rest eines Streams zurück
(define (tail s) (force (cdr s)))
```

Aufgabe 6

> Gegeben folgende Racket-Code, der ein simples Bankkonto realisieren soll. Es soll möglich sein, ein neues Bankkonten einzurichten und Geld ein- und auszuzahlen.

```
(define (make-account money)
  (lambda (movement)
    (set! money (+ money movement))
    money))

(define a (make-account 10))
(define b (make-account 100))
```

> Und gegeben die folgende Interaktion mit der Racket-REPL:

```
> a
#<procedure>
> b
#<procedure>
> (a 10)
20
> (b 10)
110
> (b 10)
120
```

> Erläutern Sie anhand der Interaktion und des Programmcodes, wie in Racket Abschlussobjekte und zustandsorientierte Programmierung genutzt werden können, um Eigenschaften objektorientierter Programmierung umzusetzen.

In Racket können Abschlussobjekte und zustandsorientierte Programmierung genutzt werden, um Eigenschaften objektorientierter Programmierung umzusetzen. Im obigen Beispiel wird das Konzept eines Bankkontos veranschaulicht. Es werden zwei Bankkonten angelegt, a und b. Jedes Konto hat ein Anfangskapital, a von 10 und b von 100. Mit der Anweisung (a 10) wird das Konto a um 10 erhöht und auf 20 erhöht, während mit (b 10) das Konto b um 10 erhöht und auf 110 erhöht wird.

Das obige Beispiel zeigt, wie Racket Abschlussobjekte und zustandsorientierte Programmierung nutzt, um Eigenschaften der objektorientierten Programmierung zu implementieren. Mit der Funktion make-account wird ein Bankkonto erstellt. Diese Funktion gibt eine Lambda-Funktion zurück, die den Zustand des Kontos speichert. Jedes Mal, wenn die Lambda-Funktion aufgerufen wird, wird der Zustand des Kontos durch den übergebenen Parameter geändert.

> Sehen Sie Grenzen in der Umsetzung objektorientierter Programmierung in Racket? Wenn ja, welche und warum?

Ja, es gibt einige Grenzen bei der Umsetzung objektorientierter Programmierung in Racket. Eine der Einschränkungen besteht darin, dass Racket keine spezifischen Objekte unterstützt, die in anderen objektorientierten Sprachen wie Java oder C++ vorhanden sind. Daher ist es schwieriger, Klassen und Methoden zu definieren, die eine objektorientierte Programmierung ermöglichen.

Darüber hinaus unterstützt Racket keine Vererbung, was bedeutet, dass es schwierig ist, eine Hierarchie von Objekten zu erstellen. Dies macht es schwieriger, eine objektorientierte Programmierung zu implementieren.

Aufgabe 7

> Implementieren Sie eine Funktion mk-mp3-control, die ein Objekt zurückliefert, das die Kontrolleinheit eines MP3-Spielers repräsentiert. (Es sollen aber nicht wirklich Dateien abgespielt werden.) Das Objekt soll folgende Informationen speichern/zurückliefern: Eine Liste, der gespeicherten MP3-Dateien (mit Dateiname und Dauer des Stücks), die Anzahl der Titel, der aktuelle Titel, der gerade abgespielt wird oder vorgewählt ist, den Abspielstatus, also ob derzeit ein Titel (welcher?) abgespielt wird oder stop, wenn kein Titel abgespielt wird. Folgende Botschaften soll das Objekt verstehen: laden: Hinzufügen einer neuen MP3-Datei an das Ende der Titelliste, loeschen: Löschen einer Datei aus der Liste, Übergabe der Nummer des zu löschenden Titels, abspielen/stop: Ändern des Abspielstatus, vor/zurück: Titel erhöhen oder erniedrigen, sind keine Titel vorhanden oder ist das Ende oder 1 erreicht, so wird die Nachricht ignoriert, unbekannte Nachrichten werden ignoriert.

Diese Aufgabe wurde in Zusammenarbeit mit Daniel Spahn bearbeitet.

```
; Erstellt ein Objekt, das die Kontrolleinheit eines MP3-Spielers repräsentiert
(define (mk-mp3-control)
  (define mp3-files '())
  (define selected-file-index 0)
  (define playing? #f)

  ; load: Laden einer Datei in die Liste (Übergabe des Dateinamens und der Dauer
des Stücks)
  (define (load song-name song-length)
    (set! mp3-files (append mp3-files (list (cons song-name song-length)))))

  ; delete: Löschen einer Datei aus der Liste (Übergabe des Indexes der Datei in
der Liste)
  (define (delete index)
    (cond [(= index selected-file-index) (set! selected-file-index 0)])
    (set! mp3-files (remove (list-ref mp3-files index) mp3-files)))

  ; toggle-play: Umschalten zwischen Play und Stop
  (define (toggle-play)
    (set! playing? (not playing?)))
```

```
  ; next: Wechsel auf den nächsten Titel
  (define (next)
    (define next-index (+ selected-file-index 1))
    (cond [(> (length mp3-files) next-index) (set! selected-file-index next-
index)]))

  ; prev: Wechsel auf den vorherigen Titel
  (define (prev)
    (define prev-index (- selected-file-index 1))
    (cond [(>= prev-index 0) (set! selected-file-index prev-index)]))

  ; die folgenden Methoden sind nur für die Ausgabe der Daten gedacht
  (define (get-mp3-files) mp3-files)
  (define (get-file-count) (length mp3-files))
  (define (get-selected-file)
    (cond [(= (length mp3-files) 0) null]
          [else (list-ref mp3-files selected-file-index)]))

  ; play-state: Gibt den aktuellen Status zurück (Play oder Stop)
  (define (play-state)
    (if playing?
        (get-selected-file)
        "stop"))

  (define (dispatch m)
    (cond
      ((equal? m 'load) load)
      ((equal? m 'delete) delete)
      ((equal? m 'toggle-play) toggle-play)
      ((equal? m 'next) next)
      ((equal? m 'prev) prev)
      ((equal? m 'get-mp3-files) get-mp3-files)
      ((equal? m 'get-file-count) get-file-count)
      ((equal? m 'get-selected-file) get-selected-file)
      ((equal? m 'play-state) play-state)))

  dispatch)
```

Aufgabe 8

> Gegeben sei die folgende Funktion in Typed Racket:

```
(: bar (-> (U Integer Boolean String) Integer))
(define (bar x)
(cond
((number? x) (string-length (number->string x)))
(else (string-length x))))
```

> Wenn man diese Funktion versucht zu kompilieren, erhält man eine Fehlermeldung vom Type Checker. Erläutern Sie die Fehlermeldung und korrigieren Sie die Funktion.

Dies bedeutet, dass die Funktion einen anderen Typ als den erwarteten erhält. Die Funktion erwartet einen Integer, aber der Typ des übergebenen Parameters x ist der universell Typ (U Integer Boolean String). Dieser Typ erlaubt es, dass sowohl Integer, Boolean und Strings als Argumente übergeben werden können.

```
(: bar (-> (U Integer Boolean String) Integer))
(define (bar x)
  (cond
    [(number? x) (string-length (number->string x))]
    [(boolean? x) 1]
    [else (string-length x)]))
```

Aufgabe 9

> Überführen Sie einige (4-5) Ihrer Programme aus den vorherigen Aufgaben in Typed Racket.

```
(: square-my-list (-> (Listof Number) (Listof Number)))
(define (square-my-list lst)
  (if (null? lst) '() (cons (sqr (car lst)) (square-my-list (cdr lst)))))


(: sum-my-list :  (Listof Integer) -> Integer)
(define (sum-my-list lst)
  (foldl + 0 lst))

(: my-find :  (Listof Any) Any -> Boolean)
(define (my-find lst x)
  (if (null? lst)
      #f
      (if (equal? (car lst) x)
          #t
          (my-find (cdr lst) x))))

(: my-reverse :  (Listof Any) -> (Listof Any))
(define (my-reverse lst)
  (cond
    [(empty? lst) empty]
    [else (append (my-reverse (rest lst))
                  (list (first lst)))]))
```

Aufgabe 10

> Schreiben Sie ein map und ein foldr in Typed Racket.

```
(: my-map (All (T U) (-> (-> T U) (Listof T) (Listof U))))
(define (my-map f lst)
  (if (null? lst) '() (cons (f (car lst)) (my-map f (cdr lst)))))

(: my-foldr (All (T U) (-> (-> T U U) U (Listof T) U)))
(define (my-foldr f init lst)
  (if (null? lst) init (f (car lst) (my-foldr f init (cdr lst)))))
```

Aufgabe 11

> Schreiben Sie ein Makro infix, mit dem Sie die Addition zweier Zahlen in Infix-Schreibweise errechnen lassen können. Anwendungsbeispiel: `(infix 1 + 2)`

```
(define-syntax-rule (infix x op y)
  (op x y))
```

Aufgabenblock 3

Aufgabe 1 (Freie Aufgabe)

Als Freie Aufgabe habe ich mich für Tic-Tac-Toe entschieden. Dies habe ich mit Hilfe der von Racket bereitgestellten GUI Bibliothek umgesetzt. [2]

Hierzu wird zunächst eine 3x3 Button Matrix erzeugt. Jedem dieser Buttons wird hierbei eine on-click (callback) Methode zugewiesen, die bei einem Klick auf den Button ausgeführt wird:

```
(define (create-grid parent)
  (let ([rows-panel (new horizontal-panel% [parent parent])])
    (create-row rows-panel 0))
  )

(define (create-row parent i)
  (when (< i 3)
    (let ([row-panel (new vertical-panel% [parent parent])])
      (create-button row-panel 0 i))
    (create-row parent (+ i 1))))

(define (create-button parent j i)
  (when (< j 3)
    (let ([btn (new button% [parent parent] [min-width 75] [min-height 75]
                    [label ""]
                    [callback handle-button-click])])
      (vector-set! grid (+ (* i 3) j) btn))
    (create-button parent (+ j 1) i)))

(create-grid frame)
```

Um das Spielfeld möglichst effizient zu erstellen, wird eine rekursive Methode verwendet, was für die funktionale Programmierung typisch ist. Diese Methode erzeugt zunächst eine Reihe von Buttons. Anschließend wird die Methode erneut aufgerufen, um die nächste Reihe zu erzeugen. Dabei wird jedem Button ein Index zugewiesen, der die Position des Buttons im Spielfeld repräsentiert.

Die Spiellogik wird in der Methode handle-button-click implementiert. Diese Methode wird bei jedem Klick auf einen Button ausgeführt.

```
(define (handle-button-click b e)
  (when (equal? (send b get-label) "")
    (send b set-label turn) ; Button Text = Current Turn
    (set! button-count (+ button-count 1)) ; Keep track of how many buttons are
clicked to determine a tie
    (change-turns)

    (unless (or (check-win) (not (send bot-checkbox get-value)))
      (play-bot (get-empty-buttons)))
    ))
```

Diese Methode prüft zunächst, ob der Button bereits geklickt wurde. Wenn dies nicht der Fall ist, wird der Button mit einem X oder O befüllt. Zudem wird geprüft, ob der Spieler den Bot aktiviert hat. In diesem Fall übernimmt der Bot den nächsten Zug:

```
(define (play-bot empty-buttons)
  (sleep 0.25) ; Sleep to make it look like the bot is thinking
  (when (not (empty? empty-buttons))
    (let ([random-index (random (length empty-buttons))])
      (send (vector-ref grid (list-ref empty-buttons random-index)) set-label
"O")
      (set! button-count (+ button-count 1))
      (change-turns)
      (check-win)))
  )
```

Dem Bot wird zunächst eine Liste mit allen freien Buttons übergeben. Anschließend wird ein zufälliger Button Index aus dieser Liste ausgewählt. Der zufällig ausgewählte Button wird dann mit einem O befüllt, wodurch der Bot seinen Zug beendet.

Am Ende jedes Zuges wird check-win aufgerufen. Diese Methode prüft anhand vordefinierten Gewinnkombinationen, ob ein Spieler gewonnen hat. Ist dies der Fall, wird eine Meldung ausgegeben und das Spiel durch deaktivieren der Buttons beendet.

check-win sieht wie folgt aus:

```
(define (check-win)
  (define (check-row indices)
```

```
    (let ([text (get-button-text-by-index (car indices))])
      (if (and (not (equal? text ""))
               (andmap (lambda (index) (equal? text (get-button-text-by-index
index))) (cdr indices)))
          text
          #f)))
  (let ((winner (or (check-row '(0 3 6))
                    (check-row '(1 4 7))
                    (check-row '(2 5 8))
                    (check-row '(0 1 2))
                    (check-row '(3 4 5))
                    (check-row '(6 7 8))
                    (check-row '(2 4 6))
                    (check-row '(0 4 8)))))
    (when winner
      (disable-buttons)
      (send turn-label set-label (string-append "Winner: " winner)))
    (when (and (not winner) (= button-count 9))
      (disable-buttons)
      (send turn-label set-label "Tie!"))
    winner))
```

Jedem Button wird ein Index zugewiesen, der die Position des Buttons im Spielfeld repräsentiert. Diese Methode prüft anhand der Indizes, ob ein Spieler gewonnen hat. Dazu wird zunächst überprüft, ob die Buttons an den Indizes 0, 3 und 6 den gleichen Text (X oder O) haben. Wenn dies der Fall ist, wird der Text zurückgegeben. Ansonsten wird die Methode erneut aufgerufen, um die nächsten Indizes zu prüfen.

Im Falle eines Unentschiedens wird eine entsprechende Meldung ausgegeben und die Buttons deaktiviert.

Zudem wurden noch einige Hilfsmethoden implementiert um den Code übersichtlicher zu gestalten.

Um den Code möglichst funktional zu gestalten, wurde wenn möglich auf globale Bindungen verzichtet. So wird beispielsweise der `play-bot`-Methode eine Liste mit allen freien Buttons übergeben, anstatt diese in der Methode zu ermitteln. Weiterhin wird der `disable-buttons`-Methode eine Liste mit allen Button-Indices übergeben, sodass diese Methode nicht auf globale Bindungen zugreifen muss.

Da dieses Projekt nur eine kleine Anwendung ist und eng mit der GUI verflochten ist, konnte jedoch nicht vollständig Seiteneffekt-frei programmiert werden. So wird in einigen Methoden auf die GUI zugegriffen, um den Button Text zu prüfen, ändern oder den Button zu deaktivieren. Weiterhin werden die globalen Bindungen `button-count` und `turn` verwendet, um den aktuellen Spielstand zu speichern.

Aufgabe 2

> Wie wird in Racket/Scheme lexical scope umgesetzt und warum ist dynamic scope eine schlechte Idee? Illustrieren Sie Ihr Argument an einem selbst entwickelten Beispiel.

Lexical scope bedeutet, dass die Gültigkeit einer Bindung von der Stelle ihres Auftretens abhängt. Das heißt, dass eine Bindung nur innerhalb des Blocks gültig ist, in dem sie definiert wurde.

Ein Beispiel für lexischen Scope wäre folgendes:

```
(define x 5)

(define (beispiel)
  (define x 3)
  (+ x 2))

(define (beispiel2)
  (+ x 2))

(beispiel) ; 5
(beispiel2) ; 7
```

In diesem Beispiel ist die Bindung `x` innerhalb der Funktion `beispiel` gültig, da sie innerhalb des Blocks definiert wurde. Die Bindung `x` in der Funktion `beispiel2` ist nicht gültig, da sie außerhalb des Blocks definiert wurde. Außerdem ist es im dynamic scope nicht möglich, eine Bindung neu zu definieren, wie es in der Funktion `beispiel` der Fall ist.

Dynamic scope ist eine schlechte Idee, da es zu unerwartetem Verhalten führen kann. Außerdem ist es schwerer zu verstehen und somit schwerer zu debuggen.

Aufgabe 3

> Nehmen Sie zum Zitat "Objects are a poor man’s closures. Closures are a poor man’s objects." Stellung und vergleichen die vorgestellten Möglichkeiten der Objektorientierung mit einer Ihnen bekannten objektorientierten Programmiersprache.

Das Zitat besagt, dass sowohl Objekte als auch Closures ähnliche Funktionalitäten haben, aber auf unterschiedliche Art und Weise genutzt werden können. Objekte sind eine einfachere Möglichkeit, Daten und Funktionen zusammenzufassen und zu organisieren. Closures dagegen ermöglichen es, Daten und Funktionen abzuschirmen und zu kapseln. Eine objektorientierte Programmiersprache wie Java bietet beides, das heißt man kann sowohl Klassen verwenden, um Objekte zu erstellen, als auch Lambda-Ausdrücke, die als Closures dienen. Beide Möglichkeiten sind nützlich und es kommt darauf an, welche Art der Datenstruktur und Funktionalität für das jeweilige Projekt am besten geeignet ist.

Aufgabe 4

> Nennen und erläutern Sie mindestens drei Gründe, warum es sinnvoll sein kann, ein Typsystem zu verwenden. Erläutern Sie anschließend den Unterschied zwischen einer dynamischen Typprüfung und einer statischen Typprüfung. Sie können für Ihre Antwort neben Racket auch noch auf andere Ihnen bekannte Programmiersprachen zurückgreifen.

1. Bessere Lesbarkeit des Codes: Durch die Verwendung von Typen können die Programmierregeln und -konventionen eindeutig festgelegt werden, damit der Code leichter lesbar und verständlich ist.

2. Erhöhte Fehlersicherheit: Typen können dazu beitragen, dass Programmfehler entdeckt werden, bevor sie zu Laufzeitfehlern führen.

3. Erhöhte Wartbarkeit: Da Typen im Code eindeutig definiert sind, ist es einfacher, den Code zu debuggen und zu verstehen.

Der Unterschied zwischen einer dynamischen Typprüfung und einer statischen Typprüfung ist, dass eine dynamische Typprüfung erst zur Laufzeit stattfindet, während eine statische Typprüfung während des Kompilierungsprozesses stattfindet. Beispielsweise kann Racket eine dynamische Typprüfung verwenden, während Java eine statische Typprüfung verwendet.

Aufgabe 5

> Beschreiben Sie, was man im Lambda-Kalkül unter einer Abstraktion und einer Applikation versteht und geben hierfür jeweils ein Beispiel an.

Unter einer Abstraktion versteht man, eine Variable in einem Ausdruck zu binden. Der λ-Operator dient dazu, eine Variable zu abstrahieren und eine Funktion zu definieren. Beispielsweise wird bei `λx.x^2` die Variable `x` gebunden.

Unter einer Applikation versteht man die Anwendung einer Funktion auf ein bestimmten Wert. Ein Beispiel dafür wäre `(λx.x^2)3` was `3^2=9` entspricht.

Aufgabe 6

> Erklären Sie den Unterschied zwischen der Reduktionsstrategie normal order und applicative order. Führen beide immer zum gleichen Ergebnis? Begründen Sie Ihre Antwort an einem selbst erstellten Beispiel.

Normal Order und Applicative Order sind zwei verschiedene Strategien zur Auswertung von Funktionsaufrufen in der Computeralgebra. Applicative Oder wertet zunächst alle Parameter aus, bevor der Rumpf der Funktion ausgeführt wird. Normal Order hingegen evaluiert nicht immer alle Parameter aus, sondern nur diese, die benötigt werden.

Ein Beispiel für Applicative Oder wäre die folgende Funktion:

```
(define (multiply x y) (* x y))

(multiply (+ 1 2) (- 4 2))
; 6
```

Die Funktion (+ 1 2) und (- 4 2) werden zuerst evaluiert und dann die Funktion multiply selbst. Alle Parameter werden also zuerst evaluiert, bevor die Funktion selbst ausgeführt wird.

Ein Beispiel für Normal Order ist `if` in Racket. `if` wertet nicht zwangsweise alle übergebenen Parameter aus, sondern evaluiert nur, was benötigt wird.

Aufgaben zur Reflexion

> Was ist Ihr erster Gedanke gewesen, als Sie die Syntax von Racket gesehen haben? Wie sehr beeinflusst die Syntax einer Programmiersprache Ihr Lernen? Ist es hinderlich, dass Sie sich an eine neue Syntax gewöhnen müssen oder fällt es Ihnen leichter, neue Konzepte mit einer ungewohnten Syntax zu erlernen?

Mein Erster Eindruck von der Racket Syntax war sehr einschüchternd, da kaum eine Ähnlichkeit zu den mir bekannten (von C abgeleiteten) Sprachen besteht.

Die Syntax hat einen großen Einfluss darauf, wie ich eine neue Sprache lerne. Die meisten Sprachen nutzen eine von C inspirierte Syntax, die ich bereits kenne. Die Syntax von Racket ist für mich jedoch sehr ungewöhnlich.

> Haben Sie bereits mit einer funktionalen Programmiersprache Kontakt gehabt? Privat, im Betrieb? Wenn ja, welche war es? Wie hat diese Sprache die vorgestellten Konzepte (statische vs. dynamische Bindung, eager vs. lazy evaluation, strikt vs. nicht strikt) umgesetzt?

Ich habe bisher noch keinen Kontakt mit funktionalen Sprachen gehabt.

> Auf welche Schwierigkeiten (wenn überhaupt) stoßen Sie beim Erlernen des Programmierparadigmas der funktionalen Programmierung? Worin sehen Sie die Ursachen?

Beim Lernen der der funktionalen Programmierung stieß ich auf einige Schwierigkeiten. Da ich noch keine Erfahrung mit funktionalen Programmiersprachen hatte, war es für mich schwierig, diese Konzepte zu verstehen und in der Praxis anzuwenden. Die Ursache für diese Schwierigkeiten lag meiner Meinung nach daran, dass funktionales Programmieren ein sehr spezielles Programmierparadigma ist, das ein anderes Denkmodell erfordert als andere Programmiersprachen.

> Sehen Sie in Typed Racket Vorteile oder Nachteile gegenüber dem "normalen" Racket? Wenn ja, welche?

Typed Racket bietet aus meiner Sicht einige Vorteile gegenüber dem normalen Racket. Entwicklern wird es durch Typsicherheit erleichtert, Fehler zu vermeiden und ihren Code zu debuggen. Weiterhin macht es den Code lesbarer und somit leichter zu verstehen.

Einige Racket Bibliotheken sind unter typed Racket zudem performanter. So ist beispielsweise die math/matrix library in typed Racket bis zu 50x schneller als in untyped Racket. [3]

> Vergleichen Sie das Typsystem von Typed Racket mit einem anderen Ihnen bekannten Typsystem. Welche Stärken und Schwächen der beiden Systeme können Sie ausmachen?

Typed Racket hat ein statisches Typsystem, bei dem Typen vor der Ausführung des Programms auf Richtigkeit überprüft werden. Es erleichtert das Debugging, da Fehler frühzeitig erkannt werden. Die statischen Typen helfen, Fehler zu reduzieren, und ermöglichen den Programmierern, vor der Ausführung eines Programms bestimmte Eigenschaften zu überprüfen.

Ein dynamisches Typsystem verzichtet auf diese Prüfungen vor der Ausführung, sodass die Laufzeit schneller ist. Dies ermöglicht auch eine größere Flexibilität bei der Entwicklung von Programmen, da Programmierer bestimmen können, welche Typen sie verwenden möchten, während sie das Programm schreiben. [4] [5]

> Halten Sie Programmiersprachen mit dynamischer oder statischer Typprüfung für "besser"? Für welchen Einsatzzweck würden Sie die eine oder andere Typprüfung empfehlen?

Ich halte weder dynamische noch statische Typprüfungen für „besser“, da beide ihre jeweiligen Vor- und Nachteile haben. Dynamische Typprüfungen bieten meiner Meinung nach den Vorteil, dass sie schneller entwickelt werden können, da der Programmierer nicht jeder Variable einem Datentyp zuweisen muss. Dies kann jedoch zu Logikfehlern führen, wenn Variablen falsch verwendet werden. Statische Typprüfungen bieten den Vorteil, dass Logikfehler vermieden werden, da der Programmierer jede Variable einem bestimmten Datentyp zuweisen muss. Dies kann jedoch zu einer längeren Entwicklungszeit führen. Ich würde daher empfehlen, dynamische Typprüfungen für schnell zu entwickelnde Anwendungen zu verwenden und statische Typprüfungen für komplexe Anwendungen, die eine optimale Performance benötigen.

> Welche Inhalte haben Sie besonders interessiert?

Ich fand die Idee der Funktionalen Programmierung sehr interessant, insbesondere den Einsatz von Funktionen, um problematische Code-Fragmente zu vereinfachen. Auch der Fokus auf Wiederverwendbarkeit und die Verwendung von Funktionen als eine Art "Bausatz" war für mich sehr faszinierend.

> Welche Inhalte haben Einfluss auf Ihr zukünftiges Programmieren?

Ich werde in Zukunft versuchen, mehr Funktionen zu verwenden, um Code zu vereinfachen und zu wiederverwenden. Weiterhin werde ich meinen Code zukünftig wenn möglich ohne Seiteneffekte schreiben.

> Ich hatte Ihnen am Anfang des Semesters das Zitat von Alan Perlis präsentiert: "A language that doesn't affect the way you think about programming is not worth knowing." Hat das Modul und Racket Ihre Art über Programmieren zu denken verändert? Wenn ja, in welcher Weise? Wenn nein, warum denken Sie, dass es das nicht getan hat?

Ja, das Modul hat meine Vorgehensweise beim Programmieren nachhaltig geprägt. Über die Bedeutung von funktionalen Konzepten wie zum Beispiel Rekursion, Seiteneffekte und Modularisierung habe ich mir zuvor nur wenig Gedanken gemacht, werde diese nun aber bei zukünftigen Projekten berücksichtigen.

Quellen

GitHub-Repository - Funktionale Programmierung: https://github.com/Julian0021/functional-programming

Racket Dokumentation: https://docs.racket-lang.org/

[1] Folien "04-funktionen-hoeherer-ordnung" Seite 15

[2] https://docs.racket-lang.org/gui/

[3] https://docs.racket-lang.org/math/matrices.html?q=matrix

[4] https://docs.racket-lang.org/ts-guide/

[5] https://stackoverflow.com/questions/42499613/advantages-of-typed-racket-over-racket