



Hausaufgabe 10 – Patternmatching und S-Expressions

Verwenden Sie zur Bearbeitung dieser Hausaufgabe die Sprache BSL mit Listen-Abkürzungen („How to Design Programs — Anfänger mit Listen-Abkürzungen“) und aktivieren Sie `match`:

```
(require 2htdp/abstraction)
```

Bearbeiten Sie dieses Blatt, außer Aufgabe 4, alleine. Stellen Sie dabei sicher, dass Sie die gesetzten Lernziele erreichen.
Viel Erfolg!

Lernziele

Nach der Bearbeitung dieses Arbeitsblattes haben Sie gelernt

- ✓ Pattern Matching für Fallunterscheidungen zu nutzen,
- ✓ Pattern Matching als Alternative für Selektoren zu nutzen,
- ✓ die verschiedenen Arten Listen zu programmieren ineinander zu übersetzen.

Nach dieser Übung sollen Sie in der Lage sein, Pattern Matching und Listenabkürzungen gezielt einzusetzen.

Aufgabe 1: Programmieren mit Pattern Matching

Beginnen Sie mit diesem Code aus der 8. Hausaufgabe:

```
(define-struct literal (value))
(define-struct addition (lhs rhs))
(define-struct subtraction (lhs rhs))
(define-struct multiplication (lhs rhs))
(define-struct division (lhs rhs))

; An Expression is one of:
; - (make-literal Number)
; - (make-addition Expression Expression)
; - (make-subtraction Expression Expression)
; - (make-multiplication Expression Expression)
; - (make-division Expression Expression)
; interp. abstract syntax of arithmetic expressions
```

Sie brauchen auch diese drei Beispiele:

```
(define EXPRESSION-1 (make-literal 42))
(define EXPRESSION-2 (make-addition (make-literal 4)
                                     (make-literal 4)))
(define EXPRESSION-3 (make-division (make-subtraction
                                     (make-literal 42)
                                     (make-literal 4))
                                     (make-literal 4)))
```

In dieser Aufgabe geht es um zwei Funktionen, die mit der Datendefinition von Expression zu tun haben. Hier ist der Anfang für die Lösung dieser Aufgabe:

```
; Any -> Boolean
; determines whether e is an Expression or not.
(check-expect (expression? EXPRESSION-1) #true)
(check-expect (expression? EXPRESSION-2) #true)
(check-expect (expression? (make-literal "42")) #false)
(define (expression? e) ...)

; A Value is a Number
; interp. the result of evaluating an expression.
```

Bewertungsschema (Gesamtpunktzahl 15 pt)

- ☐ 6 pt Es wurde Pattern Matching eingesetzt, um `expression?` korrekt zu programmieren.
- ☐ 3 pt Es wurde Pattern Matching eingesetzt, um `evaluate` korrekt zu programmieren.
- ☐ 3 pt Die Testabdeckung für die beiden Funktionen ist gut
- ☐ 3 pt Die Funktionen sind gut strukturiert, wie es von der Anwendung des Entwurfsrezeptes und DRY zu erwarten ist.

```
; Expression -> Value
; computes the value of an arithmetic expression
(define (evaluate expr) ...)
```

Programmieren Sie die Funktionen `expression?` und `evaluate` fertig.

1. Die Funktion `expression?` soll erkennen, ob `expr` der Datendefinition für `Expression` entspricht oder nicht.
2. Die Funktion `evaluate` soll den numerischen Wert von `expr` ausrechnen.

Verwenden Sie Pattern Matching.

Aufgabe 2: Programmieren mit S-Expressions

In dieser Aufgabe geht es darum, zwischen Expression-Werten und S-Expressions hin und her zu übersetzen. Zum Beispiel entsprechen sich der Expression-Wert

```
(make-addition (make-literal 1) (make-literal 2))
```

und `'(+ 1 2)`. Hier der Anfang zur Lösung dieser Aufgabe:

```
; examples converted to s-expressions
(define S-EXPRESSION-1 ...)
(define S-EXPRESSION-2 ...)
(define S-EXPRESSION-3 ...)

; Expression -> S-Expression
; converts an arithmetic expression to a S-Expression
(define (expr->sexpr expr) ...)

; S-Expression -> Expression
; tries to convert an S-Expression to an arithmetic
; expression or throws an error if not possible.
(check-expect (sexpr->expr '(+ 1 2))
  (make-addition (make-literal 1) (make-literal 2)))
(check-error (sexpr->expr '("Hello" 42)))
(define (sexpr->expr sexpr) ...)

; expr->sexpr and sexpr->expr fit together
(check-expect (expr->sexpr (sexpr->expr '(* 2 21)))
  '(* 2 21))
```

Programmieren Sie die drei Konstanten und die Funktionen `expr->sexpr` und `sexpr->expr` fertig.

1. Die Konstanten `S-EXPRESSION-1` bis `S-EXPRESSION-3` sollen S-Expressions entsprechend `EXPRESSION-1` bis `EXPRESSION-3` enthalten.
2. Die Funktion `expr->sexpr` soll einen `Expression`-Wert in die entsprechende S-Expression übersetzen.
3. Die Funktion `sexpr->expr` soll eine S-Expression in einen `Expression`-Wert übersetzen. Falls die S-Expression keinem arithmetischen Ausdruck entspricht, soll ein Laufzeitfehler erzeugt werden.

Tipp: Schlagen Sie die Datendefinition von S-Expression im Skript nach und lesen Sie die Zusatzinformation über das Programmieren mit Fehlermeldungen auf diesem Aufgabenblatt.

Bewertungsschema (Gesamtpunktzahl 30 pt)

- ☐ 5 pt Die drei Konstantendefinitionen wurden richtig in entsprechende S-Expressions übersetzt.
- ☐ 6 pt Die Funktion `expr->sexpr` wurde richtig programmiert.
- ☐ 6 pt Die Funktion `sexpr->expr` wandelt S-Expressions korrekt in Expressions um;
- ☐ 4 pt Die Funktion `sexpr->expr` erzeugt eine sinnvolle Fehlermeldung, wenn eine S-Expression nicht umgewandelt werden kann.
- ☐ 3 pt Es wurde Pattern Matching verwendet, um die beiden Funktionen zu programmieren.
- ☐ 3 pt Die Testabdeckung für die beiden Funktionen ist gut.
- ☐ 3 pt Die Funktionen sind gut strukturiert, wie es von der Anwendung des Entwurfsrezeptes und DRY zu erwarten ist.

Aufgabe 3: Arithmetische Ausdrücke mit Wahrheitswerten

In dieser Aufgabe geht es darum, neben Zahlen und den vier Grundrechenarten auch Wahrheitswerte, den Vergleich zweier Zahlen mittels `=` und aus der BSL bekannten Operationen für Wahrheitswerte zu unterstützen. Hierzu definieren wir zwei Sprachen:

1. Eine *Kernsprache*, also S-Expressions, die neben Zahlen und `+`, `-`, `*` und `/` nun auch die Wahrheitswerte `#true`, `#false` und die Operationen `=` und `if` enthalten darf. Wir unterstützen die Kernsprache, indem wir zusätzliche Strukturdefinitionen programmieren und die Datendefinition für Expression sowie `evaluate`, `expr->sexpr` und `sexpr->expr` erweitern.
2. Eine *Erweiterte Sprache*, also S-Expressions, die zusätzlich zur Kernsprache auch noch `and`, `or`, `not` und `cond` enthalten dürfen. Wir unterstützen die Erweiterte Sprache als syntaktischen Zucker, indem wir eine Funktion `desugar` schreiben, die von der Erweiterten Sprache in die Kernsprache übersetzt.

Wenn Sie damit fertig sind, können Sie Ausdrücke der Erweiterten Sprache auswerten, indem Sie zuerst `desugar`, dann `sexpr->expr` und schließlich `evaluate` aufrufen:

```
> (evaluate (sexpr->expr (desugar '(and (= 1 1) #true)
#true
```

Hier sind die zusätzlichen Strukturdefinitionen und die erweiterte Datendefinition für Expression:

```
(define-struct equality (lhs rhs))
(define-struct conditional (condition true-branch false-branch))

; An Expression is one of:
; - (make-literal Number)
; - (make-literal Boolean)
; - (make-addition Expression Expression)
; - (make-subtraction Expression Expression)
; - (make-multiplication Expression Expression)
; - (make-division Expression Expression)
; - (make-equality Expression Expression)
; - (make-conditional Expression Expression Expression)
; interp. abstract syntax of arithmetic and boolean expressions

(define S-EXPRESSION-4 '(if (= 1 1) 2 3))
(define S-EXPRESSION-5 '(if (= 1 0) 2 #true))
```

Einige neue Testfälle für bestehende Funktionen:

```
(check-expect (evaluate (sexpr->expr S-EXPRESSION-4)) 2)
(check-expect (evaluate (sexpr->expr S-EXPRESSION-5)) #true)
(check-expect (expr->sexpr (sexpr->expr S-EXPRESSION-5)) S-EXPRESSION-5)
```

Und der Anfang für die Funktion `desugar`:

```
; S-Expression -> S-Expression
; Desugars all uses of and, or, not and cond into (potentially nested) ifs.
```

Bewertungsschema (Gesamtpunktzahl 35 pt)

- ☐ 8 pt Daten-, Struktur- und Funktionsdefinitionen berücksichtigen `#true`, `#false`, `=` und `if`.
- ☐ 3 pt Auch die Tests wurden um neue Testfälle für die Erweiterung mit Wahrheitswerten erweitert.
- ☐ 6 pt `and` wird korrekt durch `desugar` in einen äquivalenten Ausdruck mit `if` umgewandelt.
- ☐ 5 pt `or` und `not` werden korrekt durch `desugar` in äquivalente Ausdrücke mit `if` umgewandelt.
- ☐ 10 pt `cond` wird korrekt durch `desugar` in einen äquivalenten Ausdruck mit `if` umgewandelt.
- ☐ 3 pt Die Funktion `desugar` (und evtl. Hilfsfunktionen) sind gut strukturiert, wie es von der Anwendung des Entwurfsrezeptes und DRY zu erwarten ist.

```
(check-expect (desugar '(and #true #false)) '(if #true (if #false #true #false) #false))
(define (desugar sexpr)
  ...)
```

Programmieren Sie die Erweiterung um Wahrheitswerte fertig.

1. Alle Datendefinitionen, Funktionsdefinitionen und Signaturen sollen die *Kernsprache* unterstützen.
2. Die Funktion `desugar` soll von der *Erweiterten Sprache* in die *Kernsprache* übersetzen.

Tipps:

- Beachten Sie, dass der Rückgabewert von `evaluate` jetzt nicht mehr einfach nur eine Zahl sondern nun auch ein Wahrheitswert sein kann. Ändern Sie hierzu die Datendefinition von `Value`.
- Lesen Sie die Zusatzinformation über Transformationsregeln auf diesem Aufgabenblatt.
- Programmieren Sie zunächst eine Variante von `desugar`, die eine S-Expression mit Pattern Matching auseinander nimmt und genau gleich wieder zusammenbaut. Fügen Sie dann nach und nach Fälle hinzu, um die einzelnen Transformationsregeln zu programmieren. Fügen Sie vorher auch jedes Mal entsprechende Tests hinzu, die überprüfen, ob Ihre Transformation funktioniert.

Kombinieren der Funktionen aus diesem Übungsblatt

Herzlichen Glückwunsch, wenn Sie dieses Übungsblatt erfolgreich bearbeitet haben, haben Sie alle nötigen Bestandteile zusammen, um schon einen etwas größeren Teil der BSL auszuwerten. Hierzu können Sie zuerst `desugar`, dann `sexpr->expr` und dann `evaluate` aufrufen:

Aufgabe 4: Freies Programmieren (Transferaufgabe)

Im Rahmen der Hausaufgaben haben Sie bisher vier größere, zusammenhängende Programme entwickelt: Portoberechnung in der fünften Hausaufgabe, die Katze die die Maus verfolgt in der sechsten Hausaufgabe, das "Breakout"-Spiel in der neunten Hausaufgabe und die Auswertung eines Teils der BSL auf diesem Hausaufgabenblatt.

Diese Aufgabe soll es Ihnen nun ermöglichen, ein größeres, zusammenhängendes Programm zu einem Thema Ihrer Wahl zu programmieren, und sich so in der Weihnachtspause mit dem Stoff der Informatik 1 auseinanderzusetzen, wenn Sie das wollen.

Überlegen Sie sich dazu eine einfache Programmidee, die Sie mit den in der Informatik 1 bisher gelernten Mitteln umsetzen können. Die vermutete Programmgröße sollte dabei einige hundert Zeilen nicht überschreiten. Wenn Sie sich ein ehrgeizigeres Ziel setzen wollen, fangen Sie am besten mit einer einfacheren Version Ihrer Programmidee an und fügen dann nach und nach weitere Aspekte ihres Programmes hinzu.

1. Beschreiben Sie in ca. einem bis drei Absätzen, was das Programm tun soll. Konzentrieren Sie sich darauf, wie ein Benutzer Ihr Programm wahrnimmt, nicht darauf, wie das Programm intern funktioniert. Erstellen Sie in der Kategorie "Freies Programmieren" im Forum der Veranstaltung ein neues Thema über Ihre Programmidee mit dieser Beschreibung.
2. Beschreiben Sie in ca. zwei bis fünf Absätzen die ungefähre technische Struktur Ihres Programmes (was sind die wichtigsten Datendefinitionen und Funktionen, welche Funktion benutzt welche anderen Funktionen, was ist der *WorldState*, falls Sie *big-bang* verwenden wollen, wollen sie *top-down* oder *bottom-up* vorgehen, ...). Stellen Sie die Beschreibung der technischen Struktur ebenfalls in das Forumthema aus Aufgabenteil 1.
3. Programmieren Sie das Programm. Sie können das Forumthema aus den anderen beiden Teilaufgaben wie einen Blog nutzen, um über Ihre Fortschritte zu berichten oder um Unterstützung zu bitten. Laden Sie dann auch Ihr fertiges Programm ins Forum.

In dieser Aufgabe dürfen Sie sich im Forum (oder sonstwo) mit den anderen Studierenden der Informatik 1 über Ihre Programmideen und Ihren Fortschritt bei der Programmierung austauschen. Unterstützung kann sowohl bedeuten, Tipps zu geben, als auch, etwas selber herausfinden zu lassen. Überlegen Sie sich, was für Ihren eigenen Lernprozess und den Lernprozess Ihrer Mitstudierenden am besten ist! Sie dürfen diese Aufgabe auch ausnahmsweise in einer kleinen Gruppe gemeinsam bearbeiten -- dies sollte dann in dem Thema im Forum ersichtlich sein. Jedes Mitglied der Gruppe sollte den Link zu Ihrem Thema in die eigene Abgabe für Aufgabe 1 bis 3 schreiben. Bitte erwähnen Sie zusätzlich Ihre Tutoren in dem Thema im Forum.

Bewertungsschema (Gesamtpunktzahl 20 pt)

- ☐ 5 pt Programmidee im Forum beschrieben
- ☐ 5 pt Technische Struktur im Forum beschrieben
- ☐ 10pt Programmidee umgesetzt.

Zusatzinformationen

Programmieren mit Fehlermeldungen

Bereits in der ersten Woche haben Sie gelernt, dass Funktionen Laufzeitfehler erzeugen können. So entsteht bei der Auswertung von `(/ 1 0)` zum Beispiel die Fehlermeldung: `/: division by zero`.

Wenn Sie Funktionen schreiben, können Sie die Funktion `error` verwenden, um eigene Fehlermeldungen zu erzeugen und die Auswertung der Funktion abubrechen. Versuchen Sie einmal auf der Konsole folgenden Ausdruck auszuwerten:

```
> (error "Test")
```

Test

In der BSL ist es üblich, dass Fehlermeldungen mit dem Namen der Funktion gefolgt von einem Doppelpunkt beginnen. Das lässt sich auch im obigen Beispiel zu `(/ 1 0)` sehen.

Wenn Funktionen Fehler "werfen", dann ist es auch wichtig darauf zu testen. Sie können die Funktion `check-error` verwenden, um zu testen, dass auch tatsächlich ein Fehler aufgetreten ist:

```
(define (invert x)
  (if (zero? x)
      (error "invert: Cannot invert number 0")
      (/ 1 x)))
```

```
> (check-error (invert 0))
```

Der eine Test ist bestanden!

```
> (check-error (invert 4))
```

check-error erwartete einen Fehler, bekam aber den Wert `0.25`

Transformationsregeln

Hier sind die Regeln für die Umformungen von `and` und `cond` (aus der Erweiterten Sprache) zu äquivalenten Ausdrücken, die nur `if` verwenden (die Kernsprache).

Bevor Sie anfangen zu Programmieren: Schreiben Sie sich auch die Regeln für `or` und `not` auf. Probieren Sie ihre umgeformten Ausdrücke anhand von Beispiel in DrRacket aus. Bedenken Sie auch Randfälle in denen z.B. die Teilausdrücke keine booleschen Werte sind.

`and` durch `if` ausdrücken

```
'(and x y)
->
'(if x (if y #true #false) #false)
```

`cond` durch `if` ausdrücken

Sie dürfen hierbei davon ausgehen, dass immer ein `else` Bedingungsausdruck vorhanden ist.

```
'(cond [c1 e1] ... [cn-1 en-1] [else en])
->
'(if c1 (if ... (if cn-1 en-1 en)))
```