Informatik 1

Forum: https://forum-db.informatik.uni-tuebingen.de/c/ws2021-info1

Übungsblatt 12 (10.02.2021)

Abgabe bis: Mittwoch, 17.02.2021, 14:00 Uhr

Relevante Videos: bis einschließlich `Informatik 1 - Chapter 12 - Video #058`.

https://tinyurl.com/Informatik1-WS2021

Sprachebene „Die Macht der Abstraktion“

Aufgabe 1: [10 Punkte]

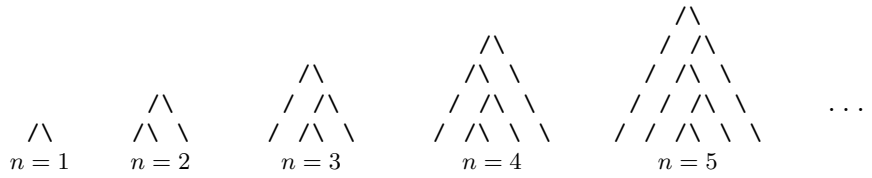


Abbildung 1: Mountain-Peaks: Ausgabe als ASCII-Grafik.

Baut eine Funktion `(: mountain-peaks (natural -> (list-of string)))`, die einen Gebirgszug der gegebenen Größe n zeichnet. Die größeren Berge liegen dabei jeweils hinter den vorderen kleineren Bergen (vgl. Abbildung 1).

Hinweise für die Implementierung:

- Beachtet, dass sich der Gebirgszug der Größe n sehr systematisch aus dem Gebirgszug der Größe $n-1$ konstruieren lässt.
- Die Zeichnungen der Gebirge sind aus einzelnen Zeilen zusammengesetzt (bspw. besteht die Zeichnung für $n = 4$ aus vier Zeilen). Stellt eine Zeichnung daher als `(list-of string)` dar, in der jedes Listenelement eine Zeile darstellt.
- Die Funktionen `unlines` und `print` (siehe unten) sind bereits vorgegeben. Der Aufruf von `(print (mountain-peaks` n`))` erzeugt in der REPL dann das gewünschte Bild des Gebirges: `print` gibt eine Liste von Zeilen untereinander aus.
- Das Zeichen \ (backslash) wird in Racket durch den String `"\\"` notiert.

```
(: unlines ((list-of string) -> string))
(define unlines
  (lambda (ys)
    (fold "" (lambda (x xs) (string-append x "\n" xs)) ys)))

(: print ((list-of string) -> %nothing))
(define print
  (lambda (ss)
    (write-string (unlines ss))))
```

Diese Aufgabe entstammt direkt aus einer sog. “Code Golfing Challenge”, einer Variante von Programmier-Wettbewerben, in denen die Teilnehmer versuchen, sich in der Kürze ihrer Lösungen zu unterbieten:
http://codegolf.stackexchange.com/questions/98588/draw-some-mountain-peaks
(Programmlänge ist in unserer Aufgabe aber kein Kriterium.)

Aufgabe 2: [10 Punkte]

In dieser Aufgabe soll eine Funktion implementiert werden, die für eine gegebene Liste alle möglichen Anordnungen (oder: **Permutationen**) ihrer Elemente erzeugt.

(a) Schreibe zuerst eine Funktion `splits` mit der folgenden Signatur:

```
(: splits ((list-of %a) -> (list-of (tuple-of (list-of %a) (list-of %a))))).
```

Die Funktion erhält eine Liste beliebiger Elemente, teilt sie an allen möglichen Stellen in zwei Listen auf und gibt alle diese Listenpaare zurück.

Beispiel:

```
(splits (list 1 2 3)) ⇝ (list (make-tuple empty (list 1 2 3))
                               (make-tuple (list 1) (list 2 3))
                               (make-tuple (list 1 2) (list 3))
                               (make-tuple (list 1 2 3) empty))
```

(b) Schreibe nun eine Funktion `permutations`, die eine Liste beliebiger Elemente erhält und basierend darauf alle Permutationen dieser Liste zurück gibt. Die Funktion hat folgende Signatur:

```
(: permutations ((list-of %a) -> (list-of (list-of %a)))).
```

Nutze hierfür die in Teilaufgabe (a) implementierte Funktion `splits`. Eine Liste mit n Elementen hat übrigens $n!$ Permutationen. Die Reihenfolge der Permutationen in der Ergebnisliste spielt keine Rolle.

Beispiele:

```
(permutations (list 1 2 3)) ⇝ (list (list 1 3 2)
                                     (list 3 1 2)
                                     (list 3 2 1)
                                     (list 1 2 3)
                                     (list 2 1 3)
                                     (list 2 3 1))
(permutations (list "a" "b" "b")) ⇝ (list (list "a" "b" "b")
                                          (list "b" "a" "b")
                                          (list "b" "b" "a")
                                          (list "a" "b" "b")
                                          (list "b" "a" "b")
                                          (list "b" "b" "a"))
```


Aufgabe 3: [20 Punkte]

In der Vorlesung wurde vorgeführt, wie sich arithmetische Ausdrücke mittels Binärbäumen darstellen lassen (Chapter 11, Folie 19). Das funktioniert, ist allerdings recht unflexibel: Die Binärbaumstruktur eignet sich zwar problemlos für *binäre* Operationen (+, −, ...) mit zwei Operanden, macht es aber z.B. nicht möglich, *unäre* Operationen (Vorzeichen-Minus, Quadratwurzel, ...) darzustellen.

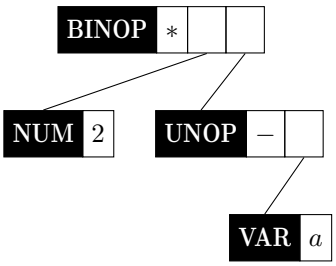
Eine in der Praxis wesentlich mächtigere Repräsentation arithmetischer Ausdrücke ist ein eigens dafür definierter *Abstract Syntax Tree*, kurz AST (siehe Abbildung 2).[1] Statt nur einer Knotenart (`node`), besitzt ein solcher AST für jede Kategorie von Werten bzw. Operationen eine spezielle Knotenart.

Für den Arithmetik-AST `term` sind das:

- binäre Operationen (`binop`) mit einem Operator (`+`, `-`, `*`, `/` oder `^`) und zwei Operanden,
- unäre Operationen (`unop`) mit einem Operator (`-`, √) und einem Operand,
- Variablen (`var`) mit einem Variablennamen und
- Zahlen (`num`) mit einem Wert.

Um bei der Auswertung unserer Arithmetik-ASTs den darin referenzierten Variablen konkrete Werte zuweisen zu können, werden wir eine *Umgebung* (`environment`) verwenden:

[1] Hintergrund: ASTs kommen aus der Welt der Programmiersprachen und Compiler. Jedes Mal, wenn du in DrRacket auf "Start" klickst, wird dein Racket-Code zunächst in einen AST umgewandelt, der dann intern evaluiert wird. (Dieser Racket-AST ist natürlich wesentlich komplexer als der Arithmetik-AST, mit dem wir uns in dieser Aufgabe befassen.)




```
(: term1 term)
(define term1
  (make-binop "*"
              (make-num 2)
              (make-unop "-"
                         (make-var "a"))))
```

Abbildung 2: AST für den Ausdruck $2 * (-a)$.

```
(define environment
  (signature (list-of (tuple-of string number))))

; Eine Umgebung, die a=7 und b=42 zuweist
(: env1 environment)
(define env1
  (list (make-tuple "a" 7)
        (make-tuple "b" 42)))
```

In den folgenden Teilaufgaben wirst du vorrangig AST-Transformationen implementieren, die zusammengenommen eine automatisierte Vereinfachung von arithmetischen Ausdrücken ermöglichen.

Wichtig: Die Datei `ast-arithmetic.rkt` enthält:

- Alle für die Arbeit mit Arithmetik-ASTs `term` und der Umgebung `environment` nötigen Definitionen.
- Einen *Parser* `term-parse`, der einen als String gegebenen Ausdruck in einen AST übersetzt.
- Einen *Pretty Printer* `term-prettyprint`, der einen AST in einer menschenlesbaren Form ausgibt.

Mache dich nun zunächst mit den in `ast-arithmetic.rkt` vorgegebenen Definitionen vertraut. Dabei kannst du den Parser ignorieren, den *Pretty Printer* kann dir jedoch als Schablone für die folgenden Teilaufgaben dienen.

(a) Damit du Arithmetik-ASTs zu einem konkreten Zahlenwert auswerten kannst, muss sich ermitteln lassen, welche Werte die darin enthaltenen Variablen haben. Schreibe also eine Funktion

```
(: lookup (environment string -> (maybe-of number)))
```

die in der übergebenen Umgebung die dem übergebenen Variablennamen zugeordnete Zahl nachschlägt. Ist der Variablenname nicht in der Umgebung definiert, soll `#f` zurückgegeben werden – deswegen gibt diese Funktion einen Wert der Signatur `(maybe-of number)` zurück.

Beispiele:

```
(lookup env1 "b") ⇝ 42
(lookup env1 "c") ⇝ #f
```

(b) Formuliere eine Funktion

```
(: eval (term environment -> number))
```

die einen AST (Signatur `term`) auswertet, d.h. den hinter dem arithmetischen Ausdruck stehenden Wert "ausrechnet". Dabei sollen die den Variablen zugeordneten Werte in der übergebenen Umgebung (Signatur `environment`) nachgeschlagen werden. Erzeuge eine `violation`, falls eine Variable im AST referenziert wird, aber nicht in der Umgebung definiert ist. Andere Fehler, die bei der Berechnung auftreten können (wie etwa die Division durch 0), müssen nicht gesondert behandelt werden.

Hinweis: Eine Quadratwurzel $\sqrt{a}$ lässt sich mittels `(sqrt a)` berechnen, eine Potenz `a^b` mittels `(expt a b)`.

Beispiele:

```
(eval term1 env1) ⇝ -14
(eval term1 (list (make-tuple "a" -21))) ⇝ 42
(eval term1 empty) ⇝ (violation "Variable nicht definiert")
(eval (term-parse "√(a^2 + b^2)") env1) ⇝ 42.579
(eval (term-parse "√(7^2 + 42^2)") empty) ⇝ 42.579
```

(c) Definiere ein Prädikat

```
(: constant? (term -> boolean))
```

das darüber Aufschluss gibt, ob der übergebene AST (Signatur `term`) *konstant* ist. Das sei genau dann der Fall, wenn darin keine Variablen referenziert werden (siehe Abbildung 3).

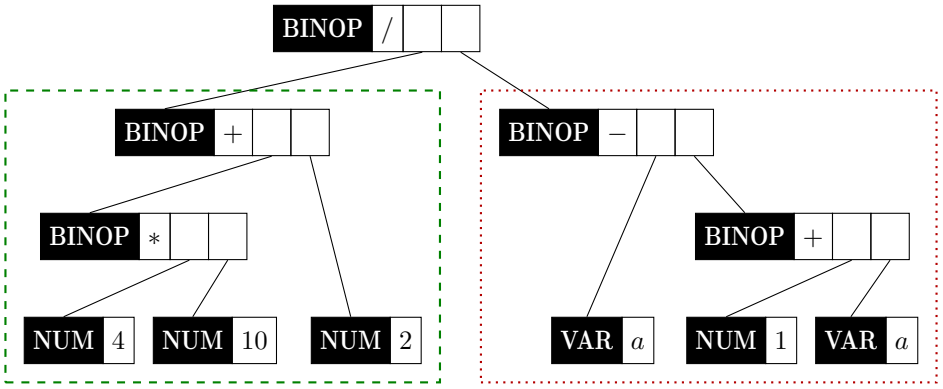


Abbildung 3: AST `term2` zum Ausdruck $((4 * 10) + 2)/(a - (1 + a))$. Der grün gestrichelt umrandete Teilbaum (`term2-left`) ist konstant; der rot gepunktet umrandete Teilbaum (`term2-right`) nicht, da dort die Variable a refenziert wird.

Beispiele:

```
(constant? term2-left) ⇝ #t
(constant? term2-right) ⇝ #f
```


(d) Schreibe eine Funktion

```
(: constant-folding (term -> term))
```


die konstante Teilausdrücke/-bäume eines ASTs mittels `eval` auswertet und durch einen `num`-Knoten mit dem Ergebniswert ersetzt.[2] Als Umgebung kannst du dabei problemlos `empty` übergeben, weil das Prädikat `constant?` garantiert, dass keine Variablen referenziert werden. Die Rückgabe ist der dabei entstehende (im alter Regel simplere) AST.

Beispiel:

```
(constant-folding term2) ⇝ (make-binop "/" (make-num 42) term2-right)
```

(e) Formuliere eine Funktion

```
(: normalize (term -> term))
```

die folgende einfache Äquivalenzen ausnutzt, um einen als AST (Signatur `term`) gegebenen arithmentischen Ausdruck zu vereinfachen:

$$a + 0 \equiv a \text{ und } 0 + a \equiv a \tag{1}$$
$$a - 0 \equiv a \tag{2}$$
$$a * 1 \equiv a \text{ und } 1 * a \equiv a \tag{3}$$
$$a * 0 \equiv 0 \text{ und } 0 * a \equiv 0 \tag{4}$$
$$a \,/\, 1 \equiv a \tag{5}$$

Implementiere alle obigen sowie **mindestens drei weitere** Äquivalenzen.

Beispiele:

```
(normalize (term-parse "0 + 1 * (a - 0)")) ⇝ (make-var "a")
(normalize (term-parse "0 + (42/1)*1 + 7 * 0")) ⇝ (make-num 42)
```

[2] Hintergrund: *Constant folding* ist eine der elementarsten Optimierungen, die ein Compiler durchführt.

(f) Schreibe eine Funktion

```
(: partial-application (term environment -> term))
```

die in der Umgebung (Signatur `environment`) definierte Variablen im AST (Signatur `term`) einsetzt, d.h. die Referenzierungen der Variablen mit den jeweiligen Zahlenwerten ersetzt, den Term ansonsten aber unverändert lässt.

Beispiel:[3]

```
(define term-add (make-binop "+" (make-var "a") (make-var "b")))

(partial-application term-add (list (make-tuple "a" 1)))
 ⇝ (make-binop "+" (make-num 1) (make-var "b"))
```

(g) Formuliere abschließend eine Funktion

```
(: simplify (term environment -> term))
```

die die AST-Transformationen `constant-folding`, `normalize` und `partial-application` kombiniert, um einen gegebenen AST (Signatur `term`) basierend auf einer gegebenen Umgebung (Signatur `environment`) so weit wie möglich auszuwerten und zu vereinfachen.

Achte darauf, die drei genannten Funktionen in der korrekten Reihenfolge anzuwenden, damit du am Ende einen möglichst einfachen arithmetischen Ausdruck erhältst.

Beispiel:

```
(simplify (term-parse "((-2) + (3*4) + 4/(1/8)) * z * ((x+y)/(x-y)) + (x-x)")
          (list (make-tuple "x" 1) (make-tuple "y" 0)))
 ⇝ (make-binop "*" (make-num 42) (make-var "z"))
```


[3] Cool: `partial-application` ist eng verwandt mit der aus der Vorlesung bekannten Funktion `curry`. Vergleiche das Beispiel mit:

```
(define add
  (lambda (a b)
    (+ a b)))
((curry add) 1) ⇝ (lambda (b) (+ 1 b)
```