Prof. Dr. T. Grust, B. Dietrich, C. Duta, D. Hirn WS 2020/2021

Computer science 1

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

Exercise sheet 12 (02/10/2021)

Submission by: Wednesday, February 17th, 2021, 2 p.m.

Relevant Videos: Up to and including Computer Science 1 - Chapter 12 - Video # 058.
https://tinyurl.com/Informatik1-WS2021

Language level „ The power of abstraction "

Exercise 1: [10 points]

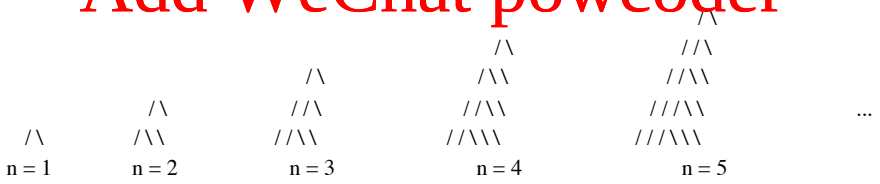

Figure 1: Mountain peaks: output as ASCII graphic.

Builds a function (: mountain-peaks (natural -> (list-of string))) that contains a mountain range of the given size n. The larger mountains lie behind the front smaller mountains (see Figure 1) .

Notes for the implementation:

- Note that the mountain range of size n is very systematically derived from the mountain range of size n - 1 can be constructed.
- The drawings of the mountains are composed of individual lines (e.g. the drawing consists of for n = 4 from four lines). Therefore, represents a drawing as a (list-of string) in which each List item represents one line.
- The functions unlines and print (see below) are already specified. The call of (print (mountain-peaks n)) then creates the desired image of the mountain in the REPL: print outputs a list of lines one below the other.
- The character \ (backslash) is noted in Racket by the string "\\".

```
(: 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))))
```

This task comes directly from a so-called “Code Golfing Challenge”, a variant of programming competitions, in which the participants try to undercut themselves in the brevity of their solutions:
http://codegolf.stackexchange.com/questions/98588/draw-some-mountain-peaks
(Program length is not a criterion in our task.)

Page 2

Exercise 2: [10 points]

In this task a function is to be implemented which for a given list contains all possible Arrangements (or: permutations) of their elements are generated.

(a) First write a function splits with the following signature:

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


The function receives a list of any elements, divides it into two lists at all possible places and returns all of these list pairs.

Example:

```
(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) Now write a function permutations that receives a list of arbitrary elements and is based then returns all permutations of this list. The function has the following signature:

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

Use the splits function implemented in subtask (a) for this . A list of n elements by the way has n! Permutations. The order of the permutations in the result list does not matter Role.

Examples:

```
(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"))

Exercise 3: [20 points]

The lecture demonstrated how arithmetic expressions are represented using binary trees leave (Chapter 11, Slide 19). That works, but it is quite inflexible: The binary tree structure is well suited for binary operations (+, -, ...) with two operands, but does it, for example not possible to represent unary operations (sign minus, square root,...).

A representation of arithmetic expressions that is much more powerful in practice is a special one Defined Abstract Syntax Tree, AST for short (see Figure 2) . 1 Instead of just one type of node, has Such an AST has a special type of node for each category of values or operations.

For the arithmetic AST term these are:

- binary operations (binop) with one operator (+, -, *, / or ^) and two operands,
- unary operations (unop) with an operator (-, /) and an operand,
- Variables (var) with a variable name and
- Numbers (num) with a value.

To give the referenced variables concrete values when evaluating our arithmetic ASTs
To be able to assign, we will use an environment:

1 Background: ASTs come from the world of programming languages and compilers. Every time you get into DrRacket If you click “Start”, your racket code will first be converted into an AST, which will then be evaluated internally. (This racket AST is of course much more complex than the arithmetic AST we are dealing with in this exercise.)

*
2 -
a

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

Figure 2: AST for the expression 2 ∗ (−a).

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

; An environment that assigns a = 7 and b = 42
(: env1 environment)
(define env1
    (list (make-tuple "a" 7)
          (make-tuple "b" 42)))
```

In the following subtasks you will primarily implement AST transformations, which together taken to enable an automated simplification of arithmetic expressions.

Important: The ast-arithmetic.rkt file contains:

- All definitions required for working with arithmetic ASTs term and the environment.
- A term-parse parser that translates an expression given as a string into an AST.
- A Pretty Printer term-prettyprint that outputs an AST in a human-readable form.

First of all, familiarize yourself with the definitions given in ast-arithmetic.rkt. There you can ignore the parser, but the Pretty Printer can be used as a template for the following Serving subtasks.

(a) In order for you to be able to evaluate arithmetic ASTs for a concrete numerical value, you must determine yourself let us know what values the variables contained therein have. So write a function

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

the number assigned to the transferred variable name in the transferred environment beats. If the variable name is not defined in the environment, #f should be returned - therefore this function returns a value of the signature (maybe-of number).

Examples:

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

(b) Formulate a function

(: eval (term environment -> number))

which evaluates an AST (signature term), i.e. the one behind the arithmetic expression Value "calculated". The values assigned to the variables should be in the transferred environment (Signature environment) can be looked up. Generate a violation if there is a variable in AST is referenced but not defined in the environment. Other errors that occurred in the calculation can occur (such as division by 0) do not need to be treated separately.

Note: a square root / a can be calculated using (sqrt a), a power a ^ b using (expt from).

Examples:

(eval term1 env1)	-14
(eval term1 (list (make-tuple "a" -21)))	42
(eval term1 empty)	(violation "variable not defined")
(eval (term-parse " / (a ^ 2 + b ^ 2) ") env1)	42,579
(eval (term-parse " / (7 ^ 2 + 42 ^ 2) ") empty)	42,579

(c) Define a predicate

(: constant? (term -> boolean))

that gives information about whether the passed AST (signature term) is constant. That is exactly then the case if no variables are referenced in it (see Figure 3) .

/

+ -

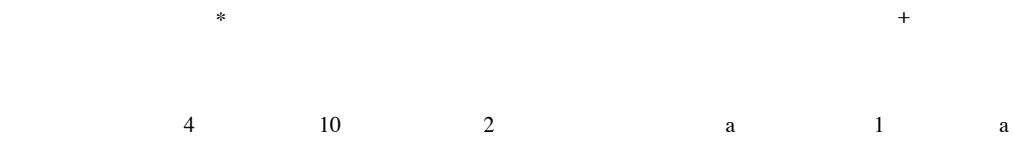

Figure 3: AST term2 for the expression ((4 ∗ 10) + 2) / (a - (1 + a)). The green dashed border part tree (term2-left) is constant; the subtree with a red dotted border (term2-right) is not, since the Variable a is referenced.

Examples:

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

(d) Write a function

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

the constant partial expressions / trees of an AST evaluates using eval and a num node replaced with the result value.[2] You can easily pass empty as the environment because the predicate constant? guarantees that no variables are referenced. The return is the resulting (usually simpler) AST.

Example:

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


(e) Formulate a function

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


which uses the following simple equivalences to calculate an arithmetic given as AST (signature term) to simplify mental expression:

$$a + 0 \equiv a \text{ and } 0 + a \equiv a \quad (1)$$

$$a - 0 \equiv a \quad (2)$$

$$a * 1 \equiv a \text{ and } 1 * a \equiv a \quad (3)$$

$$a * 0 \equiv 0 \text{ and } 0 * a \equiv 0 \quad (4)$$

$$a / 1 \equiv a \quad (5)$$

Implement all of the above and at least three other equivalences.

Examples:

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

2 Background: Constant folding is one of the most elementary optimizations that a compiler performs.

(f) Write a function

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

uses the variables defined in the environment (signature environment) in the AST (signature term), ie the references of the variables are replaced with the respective numerical values, the term otherwise but leaves unchanged.

Example:[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) Finally, formulate a function

(: simplify (term environment -> term))

which combine the AST transformations constant-folding, normalize and partial-application ned to a given AST (signature term) based on a given environment (signature environment) as much as possible and simplify it.

Make sure to use the three functions mentioned in the correct order so that you can work on the At the end you get an arithmetic expression that is as simple as possible.

Example:

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


3 Cool: partial-application is closely related to the curry function known from the lecture. Compare the example with:

```
(define add
  (lambda (from)
    (+ from)))
```

```
((curry add) 1)                (lambda (b) (+ 1 b)
```