Opgave 1 (25%)

Simple udtryk med `+`, `-` og `*` over heltal og en enkelt variabel `x` kan repræsenteres som værdier af følgende type:

Type Expression = **Sum**(const: Int, x: Unit, plus, minus, times: Arguments)
Type Arguments = **Prod**(left, right: Expression)

> **a)** Skriv udtrykket `(2+x)*(5-x*x)+18` som en konstant af typen Expression.

Et sådant udtryk kan som bekendt *differentieres* efter følgende regler:

```
diff(k) = 0
diff(x) = 1
diff(A+B) = diff(A)+diff(B)
diff(A-B) = diff(A)-diff(B)
diff(A*B) = diff(A)*B+A*diff(B)
```

hvor `k` er en heltalskonstant. Fx har vi, at:

```
diff((2+x)*(5-x*x)+18) = (5-x*x)-(2+x)(2*x)
```

(efter lidt simplifikation).

> **b)** Skriv en værdiprocedure:
>
> **Proc** diff[e: Expression] → (Expression)
>
> der returnerer den differentierede af argumentet (der skal ikke foretages simplifikation af resultatet). Der lægges vægt på, at besvarelsen er letlæselig, detaljeret og korrekt.

Et udtryk er (som bekendt?) *lineært* hvis den differentierede er en konstant funktion, dvs. at den ikke afhænger af `x`. Betragt følgende værdiprocedurer:

Proc nox[e: Expression] → (Bool)
    if is(e, const) → **return** true
    & **is**(e, x) → **return** false
    & **is**(e, plus) → **return** nox(e.plus.left) ∧ nox(e.plus.right)
    & **is**(e, minus) → **return** nox(e.minus.left) ∧ nox(e.minus.right)
    & **is**(e, times) → **return** nox(e.times.left) ∧ nox(e.times.right)
    fi
end nox

Proc linearOrWhat[e: Expression] → (Bool)
    (+ **Var** d: Expression
        d := diff[e]
        return nox[d]
    +)
end linearOrWhat

c) Er det rigtigt, at værdiproceduren linearOrWhat afgør, om argumentet er et lineært udtryk? Argumentér for dit svar.

Opgave 2 (30%)

Betragt følgende algoritme:

Algoritme: Naiv multiplikation
Stimulans: a,b: $(a \geq 0) \wedge (b \geq 0)$
Respons: r: r=a*b
Metode: r, t := 0, 0
do $\{ (r = a*t) \wedge (t \leq b) \}$
t<b → r, t := r+a, t+1
od

a) Bevis, at algoritmen er gyldig og korrekt.

b) Angiv algoritmens udførelsestid, hvis additioner antages at tage konstant tid.

Antag nu, at vi kender b's decimale repræsentation og at listen B indeholder b's cifre, det vil sige:

$$b = \sum_{j=0}^{|B|-1} B.(j) * 10^j$$

Betragt nu følgende algoritme (hvor r↑i betyder $r*10^i$):

Algoritme: Bedre multiplikation
Stimulans: a,b: $(a \geq 0) \wedge (b \geq 0)$
Respons: r: r=a*b
Metode: s, i := 0, 0
do $\{ (s = a* \sum_{j=0}^{i-1} B.(j)*10^j) \wedge (i \leq |B|) \}$
i< | B | →
r, t := 0, 0
do t<B.(i) → r, t := r+a, t+1 **od**
s := s+(r↑i)
i := i+1
od

c) Argumentér for, at denne algoritme også er korrekt (der kræves ikke nødvendigvis noget formelt bevis herfor).

d) Hvad er algoritmens udførelsestid, hvis additioner igen antages at kunne udføres i konstant tid?

e) Hvad er algoritmens udførelsestid, hvis omkostningerne ved en addition er proportional med antallet af cifre i addenderne.

Opgave 3 (25%)

Betragt følgende box, der implementerer hvad vi kunne kalde en *monoton prioritetsmængde*:

```
Box PM
    Type R = Prod(B: List(Bool), size: Int)

    Proc Init[r: R](N: Int)
        r := Prod(List(true | N), N)
    end Init

    Proc Empty[r: R] → (Bool)
        return r.size = 0
    end Empty

    Proc Delete[r: R](x: Int)
        if r.B.(x) → r.B.(x), r.size := false, r.size-1 fi
    end Delete

    Proc DeleteMin[r: R, x: Int]
        if r.size>0 →
              x := 0
              do ¬r.B.(x) → x := x+1 od
              Delete[r](x)
        & true → abort
        fi
    end DeleteMin
end PM
```

a) Angiv en specifikation af proceduren DeleteMin.

b) Angiv udførelsestiderne for samtlige procedurer.

c) Modificer boxen PM, så Init, Empty og Delete bevarer deres udførelsestider, men så DeleteMin får amortiseret konstant udførelsestid. Angiv en brugbar potentialfunktion.

d) Kan du modificere boxen, så både Empty, Delete og DeleteMin får konstante worst-case udførelsestider?

Opgave 4 (20%)

Vi betragter strenge over bogstaverne `a` til `z`, cifrene `0` til `9`, samt mellemrum. Et *simpelt mønster* er en sådan streng, der desuden må indeholde bogstavet `*`, der står for en vilkårlig streng (inklusive den tomme). Et mønster m *matcher* en streng s, hvis man kan vælge en streng for hvert `*` i mønsteret, således af den resulterende streng er lig med s. Fx matcher mønstret til venstre i alle disse tilfælde strengen til højre:

```
"hej * dig"          "hej med dig"
"data* er sj*vt"     "datalogi er sjovt"
"*111*222*"          "111122223333444"
"***"                "hvadsomhelst"
```

hvorimod ingen af disse matcher:

```
"bare * jul"         "gid det snart var jul"
"*13*"               "karakter"
```

Følgende algoritme kan bruges til at afgøre, om et mønster matcher en streng:

```
Proc Match[m, s: Text] (i, j: Int) → (Bool)
    if i ≥ | m | → return j ≥ | s |
    & m.(i) = '*' →
        if i+1 = | m | → return true fi
        (+ Var k: Int
            k := j
            do k < | s | →
                if Match[m, s](i+1, k) → return true fi
                k := k+1
            od
        +)
        return false
    & j = | s | → return false
    & m.(i) = s.(j) → return Match[m, s](i+1, j+1)
    & true → return false
    fi
end Match
```

Bemærk, at kaldet `Match[m, s](i, j)` afgør, om `m(i..|m|)` matcher `s(j..|s|)`.

a) Forklar, hvordan proceduren Match virker.

I det følgende spørgsmål kan det uden bevis benyttes at udførelsestiden for proceduren Match tager tid $\Omega(2^k)$, hvor k er længden af mønsteret.

b) Vis, hvordan man kan benytte dynamisk programmering til at at gøre proceduren mere effektiv. Hvad bliver den forbedrede tidskompleksitet?