# assignment1py

September 9, 2021

Programmer som Data - Assignment 1

PLC: 1.1, 1.2, 1.4, 2.1, 2.2, 2.3 (optionally also 2.6).

[ ]:
```
// needed to run F# Lexer/Parser
#r "nuget:FsLexYacc,10.0.0"

open System
#load "Intro/intro2.fs"
open Intro2

printfn "%i" e3v

let x = 42
```

```
1002
```

Exercise 1.1 (i) Extend the eval function to handle three additional operators: max, min, ==. using the Prim absstract syntax taking 2 argument expressions. equals should return 1 when true and 0 when false.

[ ]:
```
let rec eval e (env : (string * int) list) : int =
    match e with
    | CstI i              -> i
    | Var x               -> lookup env x
    | Prim("+", e1, e2)   -> eval e1 env + eval e2 env
    | Prim("*", e1, e2)   -> eval e1 env * eval e2 env
    | Prim("-", e1, e2)   -> eval e1 env - eval e2 env
    | Prim("max", e1, e2) ->
        let r1 = eval e1 env
        let r2 = eval e2 env
        if r1 > r2 then r1 else r2
    | Prim("min", e1, e2) ->
        let r1 = eval e1 env
        let r2 = eval e2 env
        if r1 < r2 then r1 else r2
    | Prim("==", e1, e2) ->
        let r1 = eval e1 env
        let r2 = eval e2 env
```

```
        if r1 = r2 then 1 else 0
    | Prim _            -> failwith "unknown primitive"
```

(ii) Write some example expressions in this extended expression language, using abstract syntax, and evaluate them using your new eval function.

[ ]:
```
// setting up easier printing..
let p q = printf "%O" q
let e (op:string) w en = display( sprintf "Evaluated using %s: %A" op (eval w␣
 ↪en))


let e4 = Prim("min", CstI 1, CstI 10)

p e4
e "min" e4 []

let e5 = Prim("max", CstI 1, CstI 10)

p e5
e "max" e5 []

let e6 = Prim("==", CstI 66, CstI 66)

p e6
e "==" e6 []

let e7 = Prim("==", CstI 66, CstI 22)

p e7
e "==" e7 []
```

```
Prim ("min", CstI 1, CstI 10)
```

```
Evaluated using min: 1
```

```
Prim ("max", CstI 1, CstI 10)
```

```
Evaluated using max: 10
```

```
Prim ("==", CstI 66, CstI 66)
```

```
Evaluated using ==: 1
```

```
Prim ("==", CstI 66, CstI 22)
```

```
Evaluated using ==: 0
```

(iii) Rewrite one of the eval functions to evaluate the arguments of a primitive before branching out on the operator in this style:

```
let rec eval e (env : (string * int) list) : int =
    match e with
```

```
| ...
| Prim(ope, e1, e2) ->
    let i1 = ...
    let i2 = ...
    match ope with
    | "+" -> i1 + i2
    | ...
```

[ ]:
```
let rec eval e (env : (string * int) list) : int =
    match e with
    | CstI i              -> i
    | Var x               -> lookup env x
    | Prim(ope, e1, e2)   ->
        let i1 = eval e1 env
        let i2 = eval e2 env
        match ope with
        | "+"   -> i1 + i2
        | "-"   -> i1 - i2
        | "*"   -> i1 * i2
        | "max" -> if i1 > i2 then i1 else i2
        | "min" -> if i1 < i2 then i1 else i2
        | "=="  -> if i1 = i2 then 1 else 0
        | _     -> failwith "unknown primitive"
```

(iv) Extend the expression language with conditional expressions If(e1, e2, e3) corresponding to java's e1 ? e2 : e3 or F#'s if e1 then e2 else e3

[ ]:
```
type expr =
  | CstI of int
  | Var of string
  | Prim of string * expr * expr
  | If of expr * expr * expr
```

(V) Extend the interpreter function eval correspondingly. It should evaluate e1, and if e1 is non-zero, then evaluate e2 else, evaluate e3. You should be able to evaluate the expression If(Var "a", CstI 11, CstI 22) in an enviroment that binds varible a.

[ ]:
```
let rec eval e (env : (string * int) list) : int =
    match e with
    | CstI i              -> i
    | Var x               -> lookup env x
    | Prim(ope, e1, e2)   ->
        let i1 = eval e1 env
        let i2 = eval e2 env
        match ope with
        | "+"   -> i1 + i2
        | "-"   -> i1 - i2
        | "*"   -> i1 * i2
```

```
        | "max" -> if i1 > i2 then i1 else i2
        | "min" -> if i1 < i2 then i1 else i2
        | "==" -> if i1 = i2 then 1 else 0
        | _    -> failwith "unknown primitive"
    | If(e1, e2, e3)    ->
        let con = eval e1 env
        if con > 0 then eval e2 env else eval e3 env

let e8 = If(Var "a" , CstI 11, CstI 22)

p e8

display( sprintf "a is %i" (lookup env "a"))

display( sprintf "The result is %A" (eval e8 env))
```

```
If (Var "a", CstI 11, CstI 22)

a is 3

The result is 11
```

Exercise: 1.2 (i) Declare analternative datatype ''aexpr" for a representaion of arithmetic expressions without let-bindings. The datatype should have constructors: CstI, Var, Add, Mul, Sub, for constants, variables, addition, multiplication, and subtraction. Then "x * (y + 3)" is represented as Mul(Var "x", Add(Var "y", CstI 3)), not as Prim("*", Var"x", Prim("+", Var "y", CstI 3)).

[ ]:
```
type aexpr =
| CstI of int
| Var of string
| Add of aexpr * aexpr
| Mul of aexpr * aexpr
| Sub of aexpr * aexpr
```

(ii) Write the representation of the expressions. "v - (w + z)" and "2 * (v - (w +z))" and "x + y + z + v".

```
Sub(Var "v", Add(Var "w", Var "z")),  Mul(CstI 2, Sub(Var "v", Add(Var "w", Var "z")))
and Add(Add(Add(Var "x", Var "y"), Var "z"),Var "v").
```

(iii) Write an F# function fmt : aexpr -> string to format expressions as strings. For instance, it may format Sub(Var "x", CstI 34) as the string (x-34). it has very much the same structure as the eval function but takes no enviroment argument (because the name of a varible is independent of its value).

[ ]:
```
let rec fmt ax : string =
    match ax with
    | CstI i -> string i
    | Var x -> x
    | Mul(x1,x2) -> "(" + string (fmt x1) + "*" + string (fmt x2) + ")"
    | Add(x1,x2) -> "(" + string (fmt x1) + "+" + string (fmt x2) + ")"
```

```
    | Sub(x1,x2) -> "(" + string (fmt x1) + "-" + string (fmt x2) + ")"
    //| _ -> "unknown type"

let t = Sub(Var "v", Add(Var "w", Var "z"))
let te = fmt t
//printf "%s" te
display( sprintf "The formatted string of %O is %A" (t) (te))

let t1 = Mul(CstI 2, Sub(Var "v", Add(Var "w", Var "z")))
let t1e = fmt t1
//printf "%s" t1e
display( sprintf "The formatted string of %O is %A" (t1) (t1e))

let t2 = Add(Add(Add(Var "x", Var "y"), Var "z"),Var "v")
let t2e = fmt t2
//printf "%s" t2e
display( sprintf "The formatted string of %O is %A" (t2) (t2e))

let t3 = Sub(Var "x", CstI 34)
let t3e = fmt t3
//printf "%s" t4e
display( sprintf "The formatted string of %O is %A" (t3) (t3e))
```

```
The formatted string of Sub (Var "v", Add (Var "w", Var "z")) is "(v-(w+z))"

The formatted string of Mul (CstI 2, Sub (Var "v", Add (Var "w", Var "z"))) is␣
↪"(2*(v-(w+z)))"

The formatted string of Add (Add (Add (Var "x", Var "y"), Var "z"), Var "v") is␣
↪"(((x+y)+z)+v)"

The formatted string of Sub (Var "x", CstI 34) is "(x-34)"
```

(iv) Write an F# function simplify : aexpr -> aexpr to perform expression simplification. For instance, it should simplify (x+0) to x and (1+0) to 1. The more ambitius student might want to simplify (1+0)*(x+0) to x. Hint: Pattern matching is your friend. Don't forget the case where you cannot simplify anything.

see. p. 10 in PLC for more.

[ ]:
```
let rec simplify ax : aexpr =
    match ax with
    |Add(x1,x2) ->
        if x1= CstI 0 || x2= CstI 0 then
            if x1= CstI 0 then simplify x2 else simplify x1
        else Add(simplify x1, simplify x2)
    |Sub(x1,x2) ->
        if x1 = x2 || x2 = CstI 0 then
            if x2 = CstI 0 then simplify x1 else CstI 0
        else Sub(simplify x1, simplify x2)
```

```
    |Mul(x1,x2) ->
        if x1 = CstI 1 || x2 = CstI 1 then
            if x1 = CstI 1 then simplify x2 else simplify x1
        else
            if x1 = CstI 0 || x2 = CstI 0 then CstI 0 else Mul(simplify x1,␣
 ↪simplify x2)
    |_ -> ax

// e + 0 -> e
let t5 = Add(CstI 0, CstI 1)
let t5s = simplify t5
//printf "%A" t5s
display( sprintf "The simplyfied string of %O is %A" (t5) (t5s))

// 0 + e -> e
let t6 = Add(CstI 1, CstI 0)
let t6s = simplify t6
//printf "%A" t6s
display( sprintf "The simplyfied string of %O is %A" (t6) (t6s))

// Should not change
let t1s = simplify t1
//printf "%A" t1s
display( sprintf "The simplyfied string of %O is %A" (t1) (t1s))

// e - 0 -> e
let t7 = Sub(CstI 1, CstI 0)
let t7s = simplify t7
//printf "%A" t7s
display( sprintf "The simplyfied string of %O is %A" (t7) (t7s))

// e - e -> 0
let t8 = Sub(CstI 5, CstI 5)
let t8s = simplify t8
//printf "%A" t8s
display( sprintf "The simplyfied string of %O is %A" (t8) (t8s))

// 1 * e -> e
let t9 = Mul(CstI 1, CstI 42)
let t9s = simplify t9
//printf "%A" t9s
display( sprintf "The simplyfied string of %O is %A" (t9) (t9s))

// e * 1 -> e
let t10 = Mul(CstI 42, CstI 1)
let t10s = simplify t10
//printf "%A" t10s
```

```
display( sprintf "The simplyfied string of %O is %A" (t10) (t10s))

// Since negatives is not handled in the simplify function it will reduce to␣
 ↪(0-1) + 1 AND not 0
let t11 = Mul(Add(Sub(CstI 0, CstI 1), Add(CstI 0, CstI 1)), CstI 1)
let t11f = fmt t11
//printf "%A" t11f
display( sprintf "The formatted string of %O is %A" (t11) (t11f))

let t11s = simplify t11
//printf "%A" t11s
display( sprintf "The simplyfied string of %O is %A" (t11) (t11s))

let t11sf = fmt t11s
//printf "%A" t11sf
display( sprintf "The formatted string of %O is %A" (t11s) (t11sf))
```

```
The simplyfied string of Add (CstI 0, CstI 1) is CstI 1

The simplyfied string of Add (CstI 1, CstI 0) is CstI 1

The simplyfied string of Mul (CstI 2, Sub (Var "v", Add (Var "w", Var "z"))) is␣
 ↪Mul (CstI 2, Sub (Var "v", Add (Var "w", Var "z")))

The simplyfied string of Sub (CstI 1, CstI 0) is CstI 1

The simplyfied string of Sub (CstI 5, CstI 5) is CstI 0

The simplyfied string of Mul (CstI 1, CstI 42) is CstI 42

The simplyfied string of Mul (CstI 42, CstI 1) is CstI 42

The formatted string of Mul (Add (Sub (CstI 0, CstI 1), Add (CstI 0, CstI 1)),␣
 ↪CstI 1) is "(((0-1)+(0+1))*1)"

The simplyfied string of Mul (Add (Sub (CstI 0, CstI 1), Add (CstI 0, CstI 1)),␣
 ↪CstI 1) is Add (Sub (CstI 0, CstI 1), CstI 1)

The formatted string of Add (Sub (CstI 0, CstI 1), CstI 1) is "((0-1)+1)"
```