Assignment 3 Due at 2024-02-20 08:00

Assignment description

Assignment 3

For this assignment you will work on a .NET project template (download here)that includes two files

- `Types.fs` which is a shared file between your submission and the test files
- `Assignment3.fs` which is the standard template that you are used to, but which uses `Types.fs`. You should **only** hand in `Assignment3.fs`.

In this assignment we will start work on a domain-specific language to write the functions for squares that we have been working on in Assignment 1 and Assignment 2. Ultimately, this language will also be used to describe the boards we play on.

This assignment also focusses on using algebraic datatypes, which are staples of functional programming. We will start simple and gradually build towards something more complex.

Important: The first three assignments work by extending an algebraic datatype for arithmetic expressions. You are given a template to work off of where the entire thing is defined from the start. For exercise 3.1 and 3.2 you can simply ignore the non-relevant cases, either by having a non-exhaustive pattern match, throwing an exception, or return something non sensical as the tests wont test for those cases.

Green exercises

Evaluating arithmetic expressions

We start with a simple calculator that supports addition, subtraction and multiplication. A grammar for these expressions is usually written as follows

The way to read this is that an arithmetic expression is either an integer, or an addition of two arithmetic expressions, or a subtraction of two arithmetic expressions, or a multiplication of two arithmetic expressions. We currently do not support division as we then would have to handle division by zero in a clean way. We will revisit this in Assignment 6.

These types of grammars can be mimicked closely in functional languages (one of the reasons that they are so good for writing compilers) using algebraic datatypes.

```
type aExp =
   | N of int            // Integer value
   | Add of aExp * aExp  // Addition
   | Sub of aExp * aExp  // Subtraction
   | Mul of aExp * aExp  // Multiplication

let (.+.) a b = Add (a, b)
let (.-.) a b = Sub (a, b)
let (.*.) a b = Mul (a, b)
```

From this, we can (for instance) write the following expressions.

```
> let a1 = N 42;;
- val a1 : aExp = N 42

> let a2 = N 4 .+. (N 5 .-. N 6);;
- val a2 : aExp = Add (N 4,Sub (N 5,N 6))

> let a3 = N 4 .*. N 2 .+. N 34;;
- val a3 : aExp = Add (Mul (N 4,N 2),N 34)

> let a4 = (N 4 .+. N 2) .*. N 34;;
- val a4 : aExp = Mul (Add (N 4,N 2),N 34)

> let a5 = N 4 .+. (N 2 .*. N 34);;
- val a5 : aExp = Add (N 4,Mul (N 2,N 34))
```

Note that our infix expressions have a notion, as demonstrated by `a4` and `a5`. Our grammar does not contain parentheses but the order of computation is enforced by the structure of the term and we piggy-back on the parentheses of the F# programming language to get the desired result. In this setting there is no concept of the `*` operator binding tighter than the `+` operator for instance.

Assignment 3.1

Create a function `arithEvalSimple : aExp -> int` that given an arithmetic expression `a` calculates its integer value.

Examples:

```
> arithEvalSimple a1;;
- val it : int = 42

> arithEvalSimple a2;;
- val it : int = 3

> arithEvalSimple a3;;
- val it : int = 42

> arithEvalSimple a4;;
- val it : int = 204

> arithEvalSimple a5;;
- val it : int = 72
```

Adding state

Imperative languages like C# and Java support variables that can change as the program executes. These languages have a rich typing system that dictate what type the values of these variables can have. Our language will also have variables, but we only allow integer types. This simplifies our lives significantly.

Variables are stored in maps of type `Map<string, int>` where we map from variable names to integer values. As the program executes this map changes as variables are added or updated. Arithmetic expressions, however, do not update the state but can access the variables. We update our `aExp` datatype to:

```
type aExp =
   | N of int            // Integer value
   | V of string         // Variable
   | Add of aExp * aExp  // Addition
   | Sub of aExp * aExp  // Subtraction
   | Mul of aExp * aExp  // Multiplication
```

Examples of these new expressions are:

```
> let a6 = V "x";;
- val a6 : aExp = V "x"

> let a7 = N 4 .+. (V "y" .-. V "z");;
- val a7 : aExp = Add (N 4,Sub (V "y",V "z"))
```

Evaluating these arithmetic expressions requires this variable map so that the evaluation function knows exactly what value each variable currently evaluates to. These variable maps are often referred to as ***state***.

Assignment 3.2

Create a function `arithEvalState : aExp -> Map<string, int> -> int` that given an arithmetic expression `a` and a state `s` returns the integer that `a` evaluates to where variable values are retrieved from `s`. If a variable does not exist in the state use `0` for its value.

Examples:

```
> arithEvalState a6 (Map.ofList [("x", 5)]);;
- val it : int = 5

> arithEvalState a6 (Map.ofList [("y", 5)]);;
- val it : int = 0

> arithEvalState a7 (Map.ofList [("x", 4); ("y", 5)]);;
- val it : int = 9

> arithEvalState a7 (Map.ofList [("y", 4); ("z", 5)]);;
- val it : int = 3
```

Adding word lookups

In Assignment 2 we had the following type

`type word = (char * int) list`

to model words as a list of characters and their point values. Square functions had the type

`type squareFun = word -> int -> int -> int`

that given a word `w`, the position in the word containing the letter that is placed on this particular square `pos`, and an accumulator `acc` containing the number of points calculated so far, returned a point value for this square function.

Since we want to use our DSL to replace these square functions we need a way to get `w`, `pos`, and `acc` to the functions. The integer values `pos` and `acc` are simple enough - we use the state from Assignment 3.2 to store them as variables with names `"_pos_"` and `"_acc_"` respectively. The word, however, is not an integer and can hence not be stored with the state. We will solve this by passing the word to the evaluation function (just as we do with the state) and create custom constructors to access it. For all intents and purposes the word is a constant that can be accessed, but not modified, by our programs. Our final definition of arithmetic expressions (for this assignment) is the following:

```
type aExp =
| N of int            // Integer value
| V of string         // Variable
| WL                  // Length of the word
| PV of aExp          // Point value of character at specific word index
| Add of aExp * aExp  // Addition
| Sub of aExp * aExp  // Subtraction
| Mul of aExp * aExp  // Multiplication
```

We have added the constructors `WL` that returns the length of the word, and `PV x` that returns the point value of the character at position `x` in the word.

From this definition we can create arithmetic expressions for the square functions from last week.

```
let arithSingleLetterScore = PV (V "_pos_") .+. (V "_acc_");;
let arithDoubleLetterScore = ((N 2) .*. PV (V "_pos_")) .+. (V "_acc_");;
let arithTripleLetterScore = ((N 3) .*. PV (V "_pos_")) .+. (V "_acc_");;

let arithDoubleWordScore = N 2 .*. V "_acc_";;
let arithTripleWordScore = N 3 .*. V "_acc_";;
```

Assignment 3.3

Recall from previous week that `word` is the type `(char * int) list`.

Create a function `arithEval : aExp -> word -> Map<string, int> -> int` that given an arithmetic expression `a`, a word `w`, and a state `s`, evaluates `a` with respect to `w` and `s`.

For these examples, use your definition for the word HELLO `hello` from Assignment 2.13.

Examples:

```
> arithEval WL [] Map.empty;;
- val it : int = 0

> arithEval WL hello Map.empty;;
- val it : int = 5

> arithEval (PV (N 0)) hello Map.empty;;
- val it : int = 4

> arithEval arithSingleLetterScore hello (Map.ofList [("_pos_", 4); ("_acc_", 0)]);;
- val it : int = 1

> arithEval arithSingleLetterScore hello (Map.ofList [("_pos_", 4); ("_acc_", 42)]);;
- val it : int = 43

> arithEval arithDoubleLetterScore hello (Map.ofList [("_pos_", 4); ("_acc_", 0)]);;
- val it : int = 2

> arithEval arithDoubleLetterScore hello (Map.ofList [("_pos_", 4); ("_acc_", 42)]);;
- val it : int = 44

> arithEval arithTripleLetterScore hello (Map.ofList [("_pos_", 4); ("_acc_", 0)]);;
- val it : int = 3

> arithEval arithTripleLetterScore hello (Map.ofList [("_pos_", 4); ("_acc_", 42)]);;
- val it : int = 45
```

A small imperative language

The arithmetic expressions we have so far get us a long way, but are not sufficient for all types of squares we want to be able to model. The function `containsNumbers` from Assignment 2.14, for instance, can not be modelled using arithmetic expressions as we have no conditional statements and no way to iterate over the word we are given. To solve this problem we will expand on the language we have so far and add

1. character expressions
2. boolean expressions
3. Variable assignment (all variables will still be integers)
4. conditional statements (if-then-else)
5. Sequential composition of statement (similar to the `;`-operator from Java or C#)
6. While loops

Assignment 3.4

Our language supports character constants, looking up the character from the word parameter, and casting upper case characters to lower case equivalents and vice versa.

```
type cExp =
| C  of char      (* Character value *)
| ToUpper of cExp (* Converts lower case to upper case character, 
non-letters are unchanged *)
| ToLower of cExp (* Converts upper case to lower case character, 
non-letters are unchanged *)
| CV of aExp      (* Character lookup at word index *)
```

Using the `arithEval` function from the last exercise, create a function `charEval : cExp -> word -> Map<string, int> -> char` that given an character expression `c`, a word `w`, and a state `s`, evaluates `c` with respect to `w` and `s`.

Hint: The library functions `System.Char.ToLower` and `System.Char.ToUpper` will come in handy.

Examples:

```
> charEval (C 'H') [] Map.empty;;
- val it : char = 'H'

> charEval (ToLower (CV (N 0))) hello Map.empty;;
- val it : char = 'h'

> charEval (ToUpper (C 'h')) [] Map.empty;;
- val it : char = 'H'

> charEval (ToLower (C '*')) [] Map.empty;;
- val it : char = '*'

> charEval (CV (V "x" .-. N 1)) hello (Map.ofList [("x", 5)]);;
- val it : char = 'O'
```

Assignment 3.5

Boolean expressions are defined using the following type:

```
type bExp =             
| TT                   (* true *)
| FF                   (* false *)

| AEq of aExp * aExp   (* numeric equality *)
| ALt of aExp * aExp   (* numeric less than *)

| Not of bExp          (* boolean not *)
| Conj of bExp * bExp  (* boolean conjunction *)

| IsDigit of cExp      (* check for digit *)
| IsLetter of cExp     (* check for letter *)
| IsVowel of cExp      (* check for vowel *) 
...
let (~~) b = Not b
let (.&&.) b1 b2 = Conj (b1, b2)
let (.||.) b1 b2 = ~~(~~b1 .&&. ~~b2)       (* boolean disjunction *)
...
let (.=.) a b = AEq (a, b)   
let (.<.) a b = ALt (a, b)   
let (.<>.) a b = ~~(a .=. b)                (* numeric inequality *)
let (.<=.) a b = a .<. b .||. ~~(a .<>. b)  (* numeric less than or equal to *)
let (.>=.) a b = ~~(a .<. b)                (* numeric greater than or equal to *)
let (.>.) a b = ~~(a .=. b) .&&. (a .>=. b) (* numeric greater than *)
```

Using the `arithEval` and `charEval` functions, create a function `boolEval : bExp -> word -> Map<string, int> -> bool` that given an boolean expression `b`, a word `w`, and a state `s`, evaluates `b` with respect to `w` and `s`.

Hint: You can appeal to the F# primitives for boolean connectives for all but the last two cases.

Hint: For the `isDigit` and `isLetter` you can use `System.Char.isLetter` and `System.Char.isDigit` respectively.

Hint: You will need to create an `isVowel` function yourself that works with both lower- and upper case characters, but there is `System.Char.toUpper` or `System.Char.toLower` to ensure that you only have to actively reason about either upper case or lower case characters respectively.

Examples:

```
> boolEval TT [] Map.empty;;
- val it : bool = true

> boolEval FF [] Map.empty;;
- val it : bool = false

> boolEval ((V "x" .+. V "y") .=. (V "y" .+. V "x")) 
          [] (Map.ofList [("x", 5); ("y", 7)]);;
- val it : bool = true

> boolEval ((V "x" .+. V "y") .=. (V "y" .-. V "x")) 
          [] (Map.ofList [("x", 5); ("y", 7)]);;
- val it : bool = false

> boolEval (IsLetter (CV (V "x"))) hello (Map.ofList [("x", 4)]);;
- val it : bool = true

> boolEval (IsLetter (CV (V "x"))) (('1', 0)::hello) (Map.ofList [("x", 0)]);;
- val it : bool = false

> boolEval (IsDigit (CV (V "x"))) hello (Map.ofList [("x", 4)]);;
- val it : bool = false

> boolEval (IsDigit (CV (V "x"))) (('1', 0)::hello) (Map.ofList [("x", 0)]);;
- val it : bool = true
```

Yellow exercises

Assignment 3.6

Create a function `isConsonant : cExp -> bExp` that given a character expression `c` returns a `bExp` such that `boolEval c w st` evaluates to `true` if `c` represents a consonant, and `false` otherwise.

Hint: You do *not* have to do any pattern matching on `c` here. Your function can be built purely by using the logical connectives from `bExp`.

```
> boolEval (isConsonant (C 'H')) [] Map.empty
- val it : bool = true

> boolEval (isConsonant (C 'h')) [] Map.empty
- val it : bool = true

> boolEval (isConsonant (C 'A')) [] Map.empty
- val it : bool = false

> boolEval (isConsonant (CV (V "x"))) hello (Map.ofList [("x", 0)]);;
- val it : bool = true

> boolEval (isConsonant (CV (V "x"))) hello (Map.ofList [("x", 1)]);;
- val it : bool = false
```

Evaluating statements

Neither arithmetic expressions, character expressions, or boolean expressions update the state - they use the state to calculate integer, character, and boolean values respectively. In order to update variables and actually change the state we need statements.

```
type stmnt =
| Skip                        (* does nothing *)
| Ass of string * aExp        (* variable assignment *)
| Seq of stmnt * stmnt        (* sequential composition *)
| ITE of bExp * stmnt * stmnt (* if-then-else statement *)    
| While of bExp * stmnt       (* while statement *)
```

This is a small language that supports doing nothing (`Skip`), variable assignment (`Ass`), sequential composition (`Seq`), conditional statements (`ITE`) and while loops (`While`).

The statement that actually changes the state is variable assignment that given a variable name `x` and an arithmetic expression `a` evaluates the arithmetic expression in the current state and updates the state by mapping the variable `x` to the result of the evaluation.

For instance, running the command `Ass ("z", Add (V "x", V "y"))` in the state `map [("x", 5); ("y", 7)]` will result in the map `map [("x", 5); ("y", 7); ("z", 12)]`. Our evaluation function for character, boolean, and arithmetic expressions also take a word as an argument, but this word is constant and cannot be changed by the program. The state is the only thing that changes.

More precisely, given a word `w` and a state `s`, the command

- `Skip` returns `s`
- `Ass (x, a)`
  - let `v` be the result of evaluating `a` with respect to `w` and `s`.
  - return `s` updated so that `x` maps to `v`
- `Seq (stm1, stm2)`
  - let `s'` be the result of evaluating `stm1` with respect to `w` and `s`
  - return the result of evaluting `stm2` with respect to `w` and `s'`
- `ITE (guard, stm1, stm2)`
  - let `b` be the result of evaluating the boolean expression `guard` with respect to `w` and `s`
  - if `b` is `true` return the result of evaluating `stm1` with respect to `w` and `s`
  - if `b` is `false` return the result of evaluating `stm2` with respect to `w` and `s`
- `While (guard, stm)`
  - let `b` be the result of evaluating the boolean expression `guard` with respect to `w` and `s`
  - if `b` is `true`
    - let `s'` be the result of evaluating `stm` with respect to `w` and `s`
    - return the result of evaluting `While (guard, stm)` with respect to `w` and `s'`
  - if `b` is `false`
    - return `s`

Assignment 3.7

Create a function `evalStmnt : stmnt -> word -> Map<string, int> -> Map<string, int>` that given a statement `stm`, a word `w`, and a state `s` returns the state resulting in executing `stm` with regards to `w` and `s`.

Examples:

```
> evalStmnt Skip [] Map.empty;;
- val it : Map<string,int> = map []

> evalStmnt (Ass ("x", N 5)) [] Map.empty;;
- val it : Map<string,int> = map [("x", 5)]

> evalStmnt (Seq (Ass ("x", WL), Ass ("y", N 7))) hello Map.empty;;
- val it : Map<string,int> = map [("x", 5); ("y", 7)]

> evalStmnt (ITE (WL .>=. N 5, Ass ("x", N 1), Ass ("x", N 2))) hello Map.empty;;
- val it : Map<string,int> = map [("x", 1)]

> evalStmnt (ITE (WL .<. N 5, Ass ("x", N 1), Ass ("x", N 2))) hello Map.empty;;
- val it : Map<string,int> = map [("x", 2)]

> evalStmnt (While (V "x" .<=. WL, 
                    Seq (Ass ("y", V "y" .+. V "x"), 
                         Ass ("x", V "x" .+. N 1)))) 
            hello Map.empty;;
- val it : Map<string,int> = map [("x", 6); ("y", 15)]

> evalStmnt (While (V "x" .<=. WL, 
                    Seq (Ass ("y", V "y" .+. V "x"), 
                         Ass ("x", V "x" .+. N 1)))) 
            hello (Map.ofList [("x", 3); ("y", 100)]);;
- val it : Map<string,int> = map [("x", 6); ("y", 112)]
```

Modelling squares as programs

We will use the DSL that we have created to model programs, and as described above we will pass our word parameter explicitly to the evaluation function and have integer arguments stored in the initial state that we pass to the program. The result of the program will be stored in a variable called `"_result_"`.

Assignment 3.8

Recall from Assignment 2.12 that we had the type `squareFun` where

```
type squareFun = word -> int -> int -> int
```

and the arguments are a word, a position, and an accumulator respectively.

Create a function `stmntToSquareFun : stmnt -> squareFun` that given a statemnt `stm` returns a function that given a word `w`, a position `pos`, and an accumultor `acc` evaluates `stm` with respect to `w` and the inital state `map [("_pos_", pos); ("_acc_", acc)]` and returns the value of the variable `"_result_"` after `stm` has been evaluated. You may assume that `_result_` will always be set by the program and you do not have to handle the case when it is not.

Using your function, we can create the following square functions.

```
let·singleLetterScore·=·stmntToSquareFun·(Ass·("_result_",·arithSingleLetterScore))
let·doubleLetterScore·=·stmntToSquareFun·(Ass·("_result_",·arithDoubleLetterScore))
let·tripleLetterScore·=·stmntToSquareFun·(Ass·("_result_",·arithTripleLetterScore))

let·doubleWordScore·=·stmntToSquareFun·(Ass·("_result_",·arithDoubleWordScore))
let·tripleWordScore·=·stmntToSquareFun·(Ass·("_result_",·arithTripleWordScore))

let·containsNumbers·=·
→   stmntToSquareFun·
→   →   (Seq·(Ass·("_result_",·V·"_acc_"),
→   →   ······While·(V·"i"·.<.·WL,
→   →   ·············ITE·(IsDigit·(CV·(V·"i")),
→   →   ··················Seq·(
→   →   ·······················Ass·("_result_",·V·"_result_"·.*.·N·-1),
→   →   ·······················Ass·("i",·WL)),
······················Ass·("i",·V·"i"·.+.·N·1)))))
```

Examples:

```
>·singleLetterScore·hello·0·0;;
-·val·it·:·int·=·4

>·doubleLetterScore·hello·0·0;;
-·val·it·:·int·=·8

>·tripleLetterScore·hello·0·0;;
-·val·it·:·int·=·12

>·singleLetterScore·hello·0·42;;
-·val·it·:·int·=·46

>·doubleLetterScore·hello·0·42;;
-·val·it·:·int·=·50

>·tripleLetterScore·hello·0·42;;
-·val·it·:·int·=·54

>·containsNumbers·hello·5·50;;
-·val·it·:·int·=·50

>·containsNumbers·(('0',·100)::hello)·5·50;;
-·val·it·:·int·=·-50

>·containsNumbers·(hello·@·[('0',·100)])·5·50;;
-·val·it·:·int·=·-50
```

Red exercises

Assignment 3.9

Create a statement `oddConsonants` such that the function generated from `stmntToSquareFun oddConsonants` behaves exactly the same as the `oddConsonants` function from Assignment 2.14.

For this exercise, the letter `Y` is [considered to be a consonant](#).

Hint: To check if a character is a consonant you will need to construct a boolean expression. You will have to manually match against specific characters, but there are constructs in the DSL that can make this easier. However you choose to do this, constructing this boolean expression from a helper function is recommended.

Assignment 3.10

In Assignment 2 we had the type square defined as

```
type square = (int * squareFun) list
```

Our new square type `square2` will be defined as

```
type square2 = (int * stmnt) list
```

The programs for the standard scrabble board squares can then be defined as follows, using the same priority rules as we had in Assignment 2.15.

```
let SLS = [(0, Ass ("_result_", arithSingleLetterScore))]
let DLS = [(0, Ass ("_result_", arithDoubleLetterScore))]
let TLS = [(0, Ass ("_result_", arithTripleLetterScore))]

let DWS = [(1, Ass ("_result_", arithDoubleWordScore))] @ SLS
let TWS = [(1, Ass ("_result_", arithTripleWordScore))] @ SLS
```

Create a function `calculatePoints2 : square2 list -> word -> int` that behaves exactly like `calculatePoints` from 2.14 except that this one operates on `square2` rather than `square` types.

Hint: This is a one-line function if you use `map` and `stmntToSquareFun` and then appeal to your old `calculatePoints` function.

Examples:

```
calculatePoints2 [DLS; SLS; TLS; SLS; DWS] hello;;
 val it : int = 28

calculatePoints2 [DLS; DWS; TLS; TWS; DWS] hello;;
 val it : int = 168
```