COMP3000: Tips for assignment 1

General

While there are a small amount of marks for programming in the functional style (minimal vars, minimal mutable data structures, etc.), unless you are aiming for a distinction, I suggest you forget about this.

Some things require mutable data structures anyway, e.g. for holding variable names and their values.

Remember also that common thing you want to do to some data element probably already exists as a method on the relevant class, e.g.

- check if an element is in a list (see List class)
- check if a character is a letter (see Char class)
- reduce a vector with a function, e.g. for +/4 3 7 (see Vector class and/or Scala collections or the Scala Intro notes in week 2)

Review the week 2 Scala Intro notes as most things you need are in there.

Read the spec and this document carefully. Make sure you keep an eye on announcements and the forum. Check for any code updates on the unit page.

Regular expressions

The week 4 class should help with regular expressions so that you can fill in the value for variable pat.

You can easily test pat by opening up Scala in a cmd window and trying:

```
val pat = “TO DO”.r
pat.findAllIn(“joe + 34 + mary”).toList
```

And keep modifying by using the up arrow to go back to either line and change the string.

Other than in pat, you don’t need any other regular expressions in your code.

strToObj

This function receives a string token and produces the corresponding AObject. For example:

- “+” becomes AOperator(“+”) because “+” in the operators list
- “mul” becomes AOperator(“mul”) because “mul” in the operators list
- “joe” becomes ASymbol(“joe”) because “joe” consists of letters and is not an operator
- “34” becomes ANumber(34.0) because “34” consists of digits
- “_34” becomes ANumber(-34.0) because “_34” starts with “_” and is followed by digits
- “(“ becomes LRBrac   (note: brackets will always balance)
- “)“ becomes RRBrac
- “<-“ becomes Assign

(Note: the points above look a lot like the function’s if/else if structure. Don’t try to use match.)

Once you get past the operators, the first character of the token is mostly sufficient to determine what to do.

There are links at the top of Interpret.scala including:

https://www.scala-lang.org/api/2.12.8/scala/collection/immutable/StringLike.html
https://www.scala-lang.org/api/2.12.8/scala/Char.html

This shows that there are a couple of ways to get the first character of a string s:

s.head or s(0) [no square brackets]

tokensToAObjs

This function looks like it just needs to iterate over a list of string tokens and convert each using strToObj (which sounds like a case for map, but …). This is mostly what needs to be done. For example:

- List() produces None
- List("joe", "+", "34") produces Some(List(ASymbol("joe"), AOperator("+"), ANumber(34.0)))

But numbers next to each other need to be treated as a vector, e.g. in 3 + 6 9 7.

List("3", "+", "6", "9", "7")

produces:

Some(List(ANumber(3.0), AOperator("+"), AVector(Array(6.0, 9.0, 7.0))))

So, the function needs to be on the lookout for groups of numbers, gather them into an array and produce an AVector.

The function lineToAObjs can be used in the console to test the above.

Note the function sameTokensToAObjs in the tests file. This is needed because == won't work on two AVectors.

(If you are not comfortable with Option, view it in the week 2 Scala Intro notes.)

Evaluation: exec, eval, eval1, evalWithRightArg

These functions oversee the evaluation of a line of APL. There is a loop in Main.scala that keeps prompting for a line of APL (until an empty line) and executing it:

- it converts the input line to a list of string tokens using matchPat
- that list of string tokens is converted to a list of AObjects using tokensToAObjs
- providing the list of AObjects exists (Some vs None) then the list is passed to exec
- the result of exec is displayed

Function exec is already written; all it does is reverse the list and pass it to function eval. The reversing is done because execution in APL works from right to left along the input and it is easier programmatically to peel things off the front of a list rather than the end.

You can see that eval is quite simple (if you are not going to do the distinction task of handling brackets). eval just splits the (reversed) line into the first item and the rest of the line.

Example:

```
input line:         3
reversed line given to eval:        List(ANumber(3.0))
```

eval sees that the list has only one item, so it uses eval1 to evaluate it, giving ANumber(3.0).

Example:

```
input line:         3 - 2
reversed line given to eval:        List(ANumber(2.0), AOperator(“-“), ANumber(3.0))
```

eval sees that the list has more than one item, so it uses eval1 to evaluate the first item, giving ANumber(2.0). It now calls:

```
evalWithRightArg(  ANumber(2.0),     List(AOperator(“-“), ANumber(3.0))
```

Note: the Right in evalWithRightArg refers to the argument's position in the original input line before the list is reversed; in the reversed list, the item at the start of the list is the right argument.

If the second argument is an empty list then evalWithRightArg just returns its first argument; but that is not the case here.

If the second argument is a list of length one, then it needs to contain a monadic function which would be applied to the first argument (e.g. if the input line was “- 3”); but that is not the case here.

Tip: make a list of all the monadic functions and another list for all the dyadic functions.

If the second argument is a list of length at least two then we could be looking at:

- an assignment; e.g. input: x <- 4 [credit]
- an outer product; e.g. input: x out . + y [distinction]
- an inner product; e.g. input: a + . mul b [distinction]
- operator /; e.g. input: + / 3 4 7 [distinction]
- operator rdc; [distinction]
- a dyadic function followed by a value; e.g. input: 3 – 2 [pass]
- a monadic function followed by something that isn't a value; e.g. input: 4 + - 3; the “-“ must be treated as monadic because it is followed by “+” which is not a value [pass]

Our example is the second-last case. “-“ is a dyadic operator. It is also a monadic operator but because “-“ is followed by something that has a value (“2”) then this must be dyadic usage. The function “-“ is applied to ANumber(3.0) and ANumber(2.0), producing ANumber(1.0).

The function isValue will determine whether an item from the list is a value.

eval example

This is a longer example using the input line:

```
+/4 5 7 + 2 mul - iota 2 + 1
```

This gets converted to a list containing things like ANumber(1.0) but for brevity, we will use [ ] to indicate a list and we will write 1 instead of ANumber(1.0), etc.

eval is given the reversed list of tokens:

```
eval([1, +, 2, iota, -, mul, 2, +, 4 5 7, /, +])
```

which calls (after peeling off the head of the list):

```
evalWithRightArg(eval1(1), [+, 2, iota, -, mul, 2, +, 4 5 7, /, +])

i.e.    evalWithRightArg(1, [+, 2, iota, -, mul, 2, +, 4 5 7, /, +])
```

Expect (APL) function or operator and find: +

Determine that + can be monadic or dyadic.

Is the next element in the list (2) a value (isValue)? yes, so dyadic usage

eval1(2) gives 2; need to use eval1 because the item might be a variable name.

```
1 + 2 gives 3
```

Continue with a recursive call with that result of 3:

```
evalWithRightArg(3, [iota, -, mul, 2, +, 4 5 7, /, +])
```

Expect (APL) function or operator and find: iota

Determine that iota can be monadic or dyadic.

Is the next element in the list (-) a value (isValue)? no, so monadic usage

```
iota 3 gives 1 2 3
```

So:

```
evalWithRightArg(1 2 3, [-, mul, 2, +, 4 5 7, /, +])
```

Expect (APL) function or operator and find: -

Determine that - can be monadic or dyadic.

Is the next element in the list (mul) a value (isValue)? no, so monadic usage

```
- 1 2 3 gives -1 -2 -3
```

So:

```
evalWithRightArg(-1 -2 -3, [mul, 2, +, 4 5 7, /, +])
```

Expect (APL) function or operator and find: mul

Determine that mul can be monadic or dyadic.

Is the next element in the list (2) a value (isValue)? yes, so dyadic usage

    2 mul -1 -2 -3 gives -2 -4 -6

So:

    evalWithRightArg(-2 -4 -6, [+, 4 5 7, /, +])

Expect (APL) function or operator and find: +

Determine that + can be monadic or dyadic.

Is the next element in the list (4 5 7) a value (isValue)? yes, so dyadic usage

    4 5 7 + -2 -4 -6 gives 2 1 1

So:

    evalWithRightArg(2 1 1, [/, +])

Expect (APL) function or operator and find: /

Determine that / can be dyadic function (selection) or operator (reduction).

Is the next element in the list (+) a value (isValue)? no, so not selection.
Is it a reduction function (+ - mul div flr ceil)? yes, so it is the reduction operator

    2 + 1 + 1 gives 4

So:

    evalWithRightArg(4, [])

The list is empty, so evalWithRightArg returns 4

eval example with brackets

Input line: 2 + (3 mul 4) – 5

    eval([5, -,), 4, mul, 3, (, +, 2])

    evalWithRightArg(eval1(5), [-,), 4, mul, 3, (, +, 2])

    evalWithRightArg(5, [-,), 4, mul, 3, (, +, 2])

Expect (APL) function or operator and find: -

Determine that - can be monadic or dyadic.

Is the next element in the list (")") a value (isValue)? yes, so dyadic usage. ")" signals a value because what is in the bracketed expression will produce a value.

Now evalWithRightArg wants to get that (bracketed) value. It notices that the value starts with a right bracket (if it didn't then it could just use eval1), so it has to:

1. find the matching left bracket
2. extract the list of tokens between the two brackets: [4, mul, 3]
3. get the rest of the list that is beyond the brackets: [+, 2]
4. use eval to evaluate this list of tokens: eval([4, mul, 3])
5. eval([4, mul, 3]) gives 12
6. 12 – 5 gives 7
7. now use evalWithRightArg with this value and the rest of the list:
   evalWithRightArg(7, [+, 2])
8. as before, + could be monadic or dyadic but it is followed by a value (2), so must be dyadic
9. 7 + 2 gives 9

Note that eval and evalWithRightArg are being used recursively.

Monadic functions

How is a monadic function like "-" applied, e.g. in "- 3"? It depends on what it is being applied to; so, inevitably, a match is required. If "-" is being applied to b of type AObject then:

```
b match
{
case ANumber(n) => ANumber( - n)
case AVector(v) => … // need to apply "-" to each element of v,
                     // creating a new vector and AVector
case AMatrix(m) => …
case _          => err("bad argument")
}
```

For the AVector case, try with map (e.g. Vector(4, 5, 6) map ?????).

You don't want to have to write all the above for each monadic function. Instead you want a general version of the above that is given a Scala function that applies the monadic function to a Double and returns a Double; e.g. def minus(d:Double):Double = - d.

Dyadic functions

This is similar to the monadic except now there are two parameters and the match needs to look like:

```
(a, b) match
{
case (ANumber(m), ANumber(n)) => …
case (ANumber(m), AVector(v)) => …
case (AVector(u), AVector(v)) if u.size == v.size => …
// etc.
case _                        => err("bad arguments")
}
```

Remember to allow for the case where one parameter is a vector of length one and it should be treated like a scalar.

Variables

There are only two ways that variables are used:

- “x <- 3”; assigning a value to a variable; x may be an existing variable or a new variable – it makes no difference
- “4 + x”; using the value of a variable

There are two functions corresponding to these to usages: setValue and getValue. These two functions will need a global variable to hold the association between a variable name (e.g. “x”) and its value (e.g. ANumber(3.0)). I think there is a fairly obvious Scala data structure you can use for this.

Error handling

All the test cases will use valid input lines. Your error handling won’t be tested.

23 Aug 2022