TDT4165 PROGRAMMING LANGUAGES

Exercise 2

Introduction to Language Theory

Fall 2019

This exercise is about grammar, lexing, tokenization, parsing, and interpretation. The contents of this exercise refer to lecture slides 2 and 3. The exercises will to an extent follow the lecture and reading plan.

It is acceptable to upload a single *.oz file containing the code to the exercises, in addition to answers to any theoretical questions as comments. You can also zip multiple files. It is permitted to answer the theory questions using the PDF format. You can answer in either English or Norwegian. The book should explain most if not all the topics needed, but you should also check the official documentation (most of the documentation for Mozart/Oz v1 also applies to v2).

For this exercise you may want to get acquainted with records (https://mozart.github.io/mozart-v1/doc-1.4.0/tutorial/node3.html).

You may also want to read up on RPN (Reverse-Polish Notation).
There's a short appendix containing the most useful features of Oz for this excercise.

Modelling

In this exercise you will create an interpreter for `mdc` (see lecture slide 2). You will also create a converter that converts postfix mathematical expressions that `mdc` understands into the more common infix notation.

Evaluation

You will get a score from 0 to 100 on this exercise. This score will later be converted into a final assignement score. Information on how this conversion will be done will be published on BlackBoard.

The points are awarded as follows, and you get either the full score or 0 for each criteria:
7 points: The code is indented in a reasonable and consistent way. One way to do this is to indent all blocks one level more than their parent block.
7 points: Variables/identifiers and functions/procedures have meaningful names.
7 points: Code runs without modifications, i.e. the code does not crash if run with the example inputs given in the task.
2 points per code subtask: The task is implemented correctly, i.e. given the example input the program output matches the example output.
2 points per theory subtask: The answer shows at least some knowledge on the topic.
2 points per code subtask: The workings of each function/procedure are described, for example with comments.
5 points: In task 2, give a high level description of how your `mdc` works, i.e. how it takes the postfix expression and computes the result.
6 points: In task 3, give a high level description of how you convert postfix notation to infix notation.

Task 1: Lists

The functions `{Length List}`, `{Take List Count}`, `{Drop List Count}`, `{Append List Count}`, `{Member List Count}`, and `{Position List Count}` need to be implemented. If you did exercise 1, then you can simply use those functions from the `List.oz` file like so (when you compile Oz):

```
functor
import
    System
define
    \insert List.oz
    % Your code here
end
```

If you have not done exercise 1, here are the tasks you need to finish:
a) Implement `{Length List}`, it returns the element count of List.

b) Implement `{Take List Count}`, it returns a list of the first *Count* elements. If *Count* is bigger than the amount of elements in the list, it instead returns the entire *List*.

c) Implement `{Drop List Count}`, it returns a list without the first *Count* values. If *Count* is greater than the length of the list, return *nil*.

d) Implement `{Append List1 List2}`, it returns a list of all elements in List1 followed by all elements in List2.

e) Implement `{Member List Element}`, it returns true if *Element* is present in *List*, false otherwise.

f) Implement `{Position List Element}`, it returns the position of *Element* in *List*, assume the element is present in the list.

Task 2: mdc

You will implement `mdc`: a program that interprets reverse-polish notation (postfix notation) and applies mathematical operations on numbers. See the second lecture slides for more information. The code from lecture 2 can supplement your understanding of `mdc`.

You also need to give a high level description of how your `mdc` works, i.e. how it takes the postfix expression and computes the result.

a) Implement `fun {Lex Input}` that takes a string as input and returns an array of lexemes as output. You may assume that only a single space separates the lexemes. (Hint: you may want to use `String.tokens` function)

```
{Lex "1 2 + 3 *"}
```

must return the list `["1" "2" "+" "3" "*"]`.
Depending on the print statement you use, you may have printed `[[49] [50] [43] [51] [42]]`. This is also valid, as these numbers merely represent the ASCII values of the characters in the strings.

b) Implement `fun {Tokenize Lexemes}` that puts each lexeme into a record, which we in this context call tokens. The function returns a list of records. The following records are to be used: `operator(type:plus)`, `operator(type:minus)`, `operator(type:multiply)`, `operator(type:divide)`, `number(N)` where `N` is any number.

```
{Tokenize ["1" "2" "+" "3" "*"]}
```

must return the list `[number(1) number(2) operator(type:plus) number(3) operator(type:multiply)]`.

c) Implement `fun {Interpret Tokens}` that interprets the list of records from **b)**. This means to execute the operators on the stack operands until no operators are left. The function returns the stack. Valid operators are +, -, *, and /, with / being floating point division.

```
{Interpret {Tokenize {Lex "1 2 3 +"}}}
```

Should return `[number(1) number(5)]`.
You can also write

```
{Interpret [number(1) number(2) number(3) operator(type:plus)]}
```

d) Add a matching rule for "p" which prints the stack. Use the record `command(print)`.

```
{Interpret {Tokenize {Lex "1 2 3 p +"}}}
```

must print `[number(1) number(2) number(3)]` and return `[number(1) number(5)]`.

e) Add a matching rule for "d" which duplicates the top element on the stack. Use the record `command(duplicatetop)`.

```
{Interpret {Tokenize {Lex "1 2 3 + d"}}}
```

must return `[number(1) number(5) number(5)]`.

f) Add a matching rule for "i" which flips the sign of the top of the stack. You may assume the number to be a float. You can use any record name.

g) Add a matching rule for "^" which takes the multiplicative inverse of the top of the stack. You may assume the number to be a float. You can use any record name.

Task 3: Convert postfix notation to infix notation

The postfix expressions that `mdc` consumes is unfamiliar to most, so in this task we will be converting postfix expressions to infix expressions.

You will implement `fun {Infix Tokens}`. `Tokens` is a list of tokens as defined in task 2. Your algorithm will recursively call itself whilst building an output stack of expressions. The output will in the end consist of one string that is the infix representation of the `mdc`-style input string. You scan the input tokens to `Infix` from left-to-right. The easiest way to create a non-ambigous postfix representation is by surrounding every expression in parantheses, e.g. "((1 + 2) + 3)". You can assume that the input represents a valid infix expression, "1 +" is an example of invalid input. The function does not have to handle "p" and "d".

In addition you need to give a high level description of how you convert postfix notation to infix notation.

a) Implement `fun {InfixInternal Tokens ExpressionStack}`: When you encounter a non-operator on the input stack, move it to the expression stack. When you encounter an operator on the input stack, remove expressions from the expression stack and construct the string representing this expression (e.g. "EX1 + EX2") and push the string onto the expression stack. Return the expression stack.

b) Implement `Infix` by calling `InfixInternal` with the appropriate arguments.

```
{Infix [number(3.0) number(10.0) number(9.0) operator(type:multiply) operator(type:minus)
    number(0.3) operator(type:plus)]}
% The same, but easier to read
{Infix {Tokenize {Lex "3.0 10.0 9.0 * - 0.3 +"}}}
```

must return something equivalent to (you can drop parantheses in places where they are not necessary)

```
"(3.0 - (10.0 * 9.0) + 0.3)"
```

Task 4: Theory

a) Formally describe the regular grammar of the lexemes in task 2.
b) Describe the grammar of the infix notation in task 3 using (E)BNF. Beware of operator precedence. Is the grammar ambiguous? Explain why it is or is not ambiguous?
c) What is the difference between a context-sensitive and a context-free grammar?
d) You may have gotten float-int errors in task 2. If you haven't, try running `1+1.0`. Why does this happen? Why is this a useful error?

Appendix

Records are data containers. These start with a non-capital letter. Variables, procedures, and functions start with a capital letter. A record doesn't need to be declared like a class in Python, MATLAB, or struct in C. They are simply created at the point where they are stated. The entry count of a record is not fixed, they can have any amount of entries. Key-Value types are written using `key:value`. Records can be nested. The following snippets will give you an introduction to records such that you can solve this exercise.

This exercise requires you to work with lists and recursive functions. Note that a string is a list of integers. Please see the `String` module documentation for conversion from strings to numbers. You will also need to read about procedures that 'return' variables by binding them to an input name like `String.toFloat`.

Differences for compiled and interpreted Oz: We use `System.showInfo` and/or `System.show` in the compiled version (because we don't want to focus on import shadowing). The difference is that `System.show` prints structures while `System.showInfo` is only really for printing strings. In interpreted Oz, please use `Show` and/or `Browse`. `functor import` and `define` is not required in interpreted mode. You can not `\insert` a file that contains `functor import`, because it will create the following:

```
functor
import
    System
define
    % from \insert SomeFile.oz, it simply pastes the code during compilation
    functor
    import
        System
    define
        % ...
    end
    % End of SomeFile.oz

    % Your code
end
```

If you use interpreted mode, you can just copy the code instead of doing `\insert`.

You can put all answers to the tasks in a single file.

How to create and pattern match on a record:

```
local X = myRecord(10) in
    case X of myRecord(N) then
        {System.showInfo N}
    end
end
```

Pattern matching simply strips common parts of both the variable and the pattern and binds anything StartingWithACapitalLetter in the pattern to the element in the variable.

```
local X = myRecord(value:10 2:nested("record")) in
    case X of myRecord(N) then
        {System.showInfo N}
    [] myRecord(value:N 2:S) then
        {System.showInfo N}
        {System.show S}
    end
end
```

Your patterns can be of arbitrary nestedness:

```
local X = myRecord(value:10 2:nested("record")) in
    case X of myRecord(N) then
        {System.showInfo N}
    [] myRecord(value:N 2:nested(Something)) then
        {System.showInfo N}
        {System.showInfo Something}
    end
end
```

Or you can assert that part of the pattern is already bound:

```
local X = myRecord(value:10 2:nested("record")) in
    case X of myRecord(N) then
        {System.showInfo "First Case"}
        {System.showInfo N}
    [] myRecord(value:N 2:nested("record")) then
        {System.showInfo "Second Case"}
        {System.showInfo N}
    end
end
```

Suppose we change ”record” in the pattern:

```
local X = myRecord(value:10 2:nested("record")) in
    case X of myRecord(N) then
        {System.showInfo "First Case"}
        {System.showInfo N}
    [] myRecord(value:N 2:nested("Something different")) then
        {System.showInfo "Second Case"}
        {System.showInfo N}
    else
        {System.showInfo "Nothing matched"}
    end
end
```

then only the else case will run because unification would fail on `"record" = "Something different"`.