TDT4165 — Programming Languages

Assignment 2: Introduction to Language Theory

Autumn 2025

This exercise is about grammar, lexing, tokenization, parsing, and interpretation. The contents of this exercise refer to slides from lectures 2 and 3. The exercises will to an extent follow the lecture and reading plan. Even if you upload a PDF with screenshot of the code, please always upload also a file with working Oz code in .oz format. In case you upload multiple files on Blackboard, please *do not* zip them, as Blackboard does not have a very friendly way to display archives.

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 v1 also applies to v2). For this exercise you may want to get acquainted with records (http://mozart2.org/mozart-v1/doc-1.4.0/tutorial/node3.html). You may also want to read up on Reverse-Polish Notation (RPN). This document has also an Appendix that summarizes records and other useful information about using Oz/Mozart.

Objective

In this exercise you will create an interpreter for **mdc** (see slides from lecture 2). You will also create a converter that converts postfix mathematical expression that **mdc** understands into their representation as an expression tree.

Evalutation

You will get a Approved/Not approved scoring for this assignment.

The requirements to get this exercise approved are as follows:

- Complete task 1
- Attempt task 2, meaning you have to try to implement the postfix-to-tree converter. You are required to include a high-level description of how the conversion works, even if you have not implemented the task.
- Complete task 3

Preliminaries

The \insert command

To complete this exercise, some of the functions the functions from Assignment 1 may turn out useful, i.e., `{Length List}`, `{Take List Count}`, `{Drop List Count}`, `{Append List Count}`, `{Member List Element}`, `{Position List Element}`, `{Push List Element}`, `{Peek List}`, and `{Pop List}`, If you did complete Assignment 1, then you can simply use those functions from the `List.oz` like so:

```
\insert 'List.oz'
% Your new code here
```

In the OPI, using `\insert` can be confusing, especially if you are using remote network locations; those are sometimes represented as a file path with two backslashes at the beginning. To get such paths to work, you should: 1) surround the entire path in single quotes, 2) escape any leading backslash, i.e., each backslash needs its own backslash beforehand to tell the compiler that the symbol is to be taken literally, and 3) change any Windows-style backslashes in the path to Unix-style forward slashes. An example of how a path would look like is below:

```
\insert \\network.location.ntnu.no\path\to\user\TDT4165\List.oz
%This is wrong

\insert '\\\\network.location.ntnu.no/path/to/user/TDT4165/List.oz'
%This is correct
```

If you want to use relative paths, you first need to know what is the current directory according to the OPI, i.e., where does the program think you are now. When using the 'Oz' buffer without saving it to a file, the current directory is set to the Oz installation folder, which might not be very convenient.

In general, it is more reliable to use absolute paths, as shown above, or to be sure to save the buffer to a file first. When working from a file saved in the filesystem, the current directory is set to the one where that file is stored. So, if you are working from a file that resides in a directory you know, by using `./` at the start of the file path, you tell the compiler to start searching from the location the current file is stored in.

Local vs. global scope

To avoid having to constantly use `local F in S end` or having to surround large pieces of code with the `local` statement, we can use `declare F in` which makes `F` of global scope, like this:

```
declare Length Take Drop Append Member Position in
```

By doing this in your `List.oz` file, your functions will be available for later use.

Review

If you have not yet completed the exercises from Assignment 1, go back there and finish those before continuing.

Task 1: mdc

You will implement `mdc`: a program that interprets reverse-polish notation and applies mathematical operations on numbers. See slides from the second lecture 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`}, which takes a string as input and returns an array of lexemes as output. You may assume that that lexemes are separated by a single whitespace character. (*Hint*: you may want to use `String.tokens` function). For example,

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

must return the list `["1" "2" "+" "3" "*"]`.

When printing the output, depending on the print statement you use, you may have the output printed as `[[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`}, which puts each lexeme into a record. The function returns a list of records. The following records have to be used: `operator(type:plus)`, `operator(type:minus)`, `operator(type:multiply)`, `operator(type:divide)`, `number(N)` where `N` is any number. For example,

```
    {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`}, which interprets the list of records from **b)**. This means to *interpret* the commands from the `mdc` language, modifying its internal stack until no operators are left. The function returns the stack. Valid operators are those corresponding to the record above, i.e., +, -, *, and /, with / being floating point division. For example,

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

Must return the corresponding stack (i.e., list) after processing all the symbols in the input, that is `[5 1]`. Note that the functions you are developing can be chained, that is, the above should be the same as writing:

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

which should also return `[5 1]`.

d) Add a matching rule for command “p”, which prints the current content of the stack. Use the record `command(print)`. For example,

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

must *print* `[2 1]` on the Emulator buffer or the terminal, and must also *return* `[5 1]`, as the new command does not alter the content of the stack.

e) Add a matching rule for “d”, which duplicates the top element on the stack. For example,

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

must return `[5 5 1]`.

f) Add a matching rule for “i”, which flips the sign of the number at the top of the stack. You can use any record name you find appropriate.

g) Add a matching rule for “c”, which clears the stack (i.e., makes it empty). You can use any record name you find appropriate.

Task 2: Convert postfix notation into an expression tree

You will implement `fun {ExpressionTree Tokens}`, where `Tokens` is a list of tokens as defined in Task 1. The function will return a record organized as a tree, representing the expression parsed from the postfix representation of the `mdc`-style input string.

For example, when receiving the input:

```
[number(2) number(3) operator(type:plus) number(5) operator(type:divide)]
```

the function should return the output:

```
divide(5 plus(3 2))
```

corresponding to the expression $5/(3 + 2)$.

Your algorithm will process the tokens in the input list from left to right, recursively calling itself whilst building an stack of expressions. You can assume that the input represents a valid postfix expression, "`1 +`" being an example of invalid input. You can also assume that the expression only contains numbers and the mathematical operators `+ - * /`. That is, you do not need to consider the "p" and "d" commands from Task 1.

In addition, you need to give a high level description of how you convert postfix notation to an expression tree.

a) Implement `fun {ExpressionTreeInternal Tokens ExpressionStack}`, as follows.

- When you encounter a non-operator token in the input list, push it to the expression stack.
- When you encounter an operator token in the input list, remove two expressions from the expression stack and use them as operands, in a newly built expression according to the received operator (e.g., `plus(EXP1 EXP2)`). Then, push this new expression to the expression stack.
- When all the input tokens have been processed, return the element at the top of the expression stack.

b) Implement `ExpressionTree` by calling `ExpressionTreeInternal` with the appropriate arguments.

```
{ExpressionTree [number(3) number(10) number(9) operator(type:multiply)
    operator(type:minus) number(7) operator(type:plus)]}

% The same, but easier to read
{ExpressionTree {Tokenize {Lex "3 10 9 * - 7 +"}}}
```

must return something equivalent to the record:

```
plus(7 minus(multiply(9 10) 3))
```

which in turns corresponds to the expression $(7 + ((9 * 10) - 3))$.

Task 3: Theory

a) Describe the grammar that defines the language whose strings are the *lexemes* in Task 1 using a formal notation. This means not using EBNF, but using the formal notation with sets and tuples. Refer to slide 53 in Lecture 2.

b) Describe the grammar that defines the language whose strings are the records returned by the `ExpressionTree` function in Task 2, using (E)BNF.

c) Which kind of grammar is the grammar you defined in step **a)**? Is it regular, context-free, context-sensitive, or unconstrained? What about the one from step **b)**?

Appendix

Records

Records are data containers. These start with a non-capital letter. Identifiers (i.e., names of values, which can be variables, procedures, and functions, among the others) start with a capital letter. A record doesn't need to be declared like a class in Python, or a struct in C. They are simply created at the point where they are stated.

A record can have any amount of entries, called 'components' and specified as key-value pairs using `key:value`. Once a record is created, components cannot be added or removed. However, the value of a component can be set to an unbound variable, which is then bound later. Records can be nested. The following snippets will give you an introduction to records such that you can solve this assignment.

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.

Producing output

There are different ways to print output in Oz, you can find details on the different print procedures in the documentation of the `System` module. We recommend using the command `System.show`, which is able to print structures, while `System.showInfo` is only really for printing strings. In interpreted Oz (i.e., in the OPI/Emacs or VSCode environment) you can also use `Browse` as a mean to debug your code.

Compiled Oz and functors

Mozart 1.4 used to have different ways to execute Oz code, including compilation to machine code and the possibility to define your own modules (called "functors"). See for example: http://mozart2.org/mozart-v1/doc-1.4.0/apptut/node3.html. You might want to try functors by using Mozart 1.4, but this is not required for the course. You can write the answers to all the tasks in a single file.

The `case` statement and pattern matching on records

The `case` statement in Oz works differently from similar constructs in other languages, for example the `switch...case` in Java. In Oz, the `case` statement can be used to perform *structural pattern matching* and "extract" values from data structures.

For example, the following code creates a record and then applies pattern matching on it.

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

The `case` statement tries to match the content of the provided identifier (i.e., `X` in the example above) to one of the patterns given in the various cases. When a match is found (i.e., the content of `X` and the pattern are compatible), then: 1) any unbound variables in the pattern are bound to the corresponding elements in `X`, 2) the statements corresponding to that case are executed, and 3) further cases are discarted.

```
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
```

Unbound variables can appear in your patterns at an arbitrary depth (recall that identifiers start with an uppercase letter).

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

Or, alternatively, you may want to prevcisely match part of the pattern, and only leave another part unbound.

```
    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
```

It is also possible to add a default case, in case none of the previous patterns are matched.

```
    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
```

In this case the “else” case will be executed, because unification would fail on `"record" = "Something different"`.