TDT4165

Assignment 2

Karoline Barstein

All Oz code is attached in the delivery in the .zip folder "Assignment 2".

High-level description of how the `mdc` works

The `mdc` consists of three parts - a lexemizer, a tokenizer and an interpreter - which together transform a single string of lexemes (in postfix notation) into a computed result (in token notation).

The lexemizer converts the single string of lexemes into a list of separate lexemes.

The tokenizer converts a list of separate lexemes into tokens. The valid operators and commands together with their corresponding names are defined. For each lexeme in the list, the tokenizer first checks if the lexeme is an operator. If so, it maps the lexeme into an operator token. If not, the tokenizer checks if the lexeme is a command, and if so, it maps the lexeme into a command token. If the lexeme does not match any of these, the tokenizer will try to convert the lexeme into a number. If the lexeme is not a number (i.e., the lexeme is neither a valid operator, a valid command nor a number), an exception will be raised. The mapped list of tokens is returned.

The interpreter interprets a list of tokens and executes the operators on the stack operands until no operators are left. First it defines the valid command and operator tokens. Then an inner interpreter will iterate through the tokens. If there are no tokens in the list, i.e., we have iterated through the whole list, the stack is reversed to get the correct order of the elements, then each element is converted into a string and tokenized, and at last returned. If the list is not empty, we check if the top token is a number. If so, the number is put on top of the stack, and the rest of the list is recursively iterated through. If the top token is not a number, we check if it is a command. If so, the corresponding command is executed, and the rest of the list is recursively iterated through. If the top token is not a command, it checks if it is an operator. If so, it will perform the operator on the two top elements of the stack, and then put the result back onto the stack before the rest of the list is recursively iterated through. If the token is not an operator, it is an invalid token. If so, the token is skipped, and the rest of the list is iterated through.

High-level description of postfix to infix conversion

In the converter, the valid operator and command tokens are defined. These are used to convert operator and command tokens into the corresponding character/lexeme. Then an inner converter is defined. It performs pattern matching on the given tokens. If there are no tokens in the list, i.e., we are at the end of the list, the top element of the expression stack (this element contains the fully converted expression, if the list of tokens represents a valid expression) is returned. If there are tokens left, we check if the top token is a number. If so, the number is put onto the expression stack and the rest of the list is recursively iterated through. If the token is not a number, we check if it is an operator. If so, an infix expression is constructed of the operator and the two top elements of the stack. The expression is put back onto the stack, then the rest of the list is recursively iterated through. If the token is not an operator,

we check whether it is a command. The valid commands are 'i', '^', 'd' and 'p', where the two first are one-operand operators. The two last ones are "functional" commands, so they are not to be a part of the resulting expression, and their corresponding character is therefore an empty string. The command is then put next to the top element in the stack and the expression is put back onto the stack. If the token is not a command, it is an invalid token, and the converter will skip this and iterate through the rest of the list.

Theory

a) **Formally describe the regular grammar of the lexemes in task 2.**
Regular grammar:
`<lexeme> ::= number | operator | command`
`<operator> ::= '+' | '-' | '*' | '/'`
`<command> ::= 'p' | 'd' | 'i' | '^'`

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?**
In my implementation of the postfix to infix notation I incorporated paranthesis around all multiplicative or additive expressions, mainly to increase readability and shortness of the code. Using (E)BNF:
`<primary> ::= '(' expression ')' | number | 'i'primary | '^'primary`
`<multiplicative-expression> ::= '(' primary ( ( '*' | '/' ) primary ) * ')'`
`<expression> ::= '(' multiplicative-expression ( ( '+' | '-' ) multiplicative-expression ) * ')'`
To show that a grammar is ambiguous, one only needs to prove that *one* sentence derived from the grammar have two or more parse trees. To show that the grammar is *un*ambiguous, one needs to prove that for all sentences derived from the grammar, no sentence has more tan one parse tree. I believe this grammar is unambiguous, mainly because of the incorporation of the parenthesis that separates the smaller/inner expressions in an expression and the construction of the multiplicative-expression, which takes care of operator precedence. But I am not able to prove this statement, except from deriving some example statements. All my attempts gave only one parse tree.

c) **What is the difference between a context-sensitive and a context-free grammar?**
Let ν be a non-terminal and α, β, γ be terminals or non-terminals.
In a *context-sensitive* grammar all production rules are of the form $\alpha\nu\beta ::= \alpha\gamma\beta$, i.e., if ν is in a given context (given by α and β), then you can replace it by γ.
In a *context-free grammar*, all rules are of the form $\nu ::= \gamma$, i.e., all production rules are simple replacements.

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?**
In Oz, there is no implicit conversion from float to int or vice versa, so if one tries to execute an arithmetic procedure where one (or more) input argument(s) are float(s) and one (or more) input argument(s) are int(s), this will raise a type (float-int) error. This error is useful because it raises type safety. This prevents, e.g., illegal operations, and makes the operations more predictable (e.g., you know that you get the correct type from a calling function to a caller function). This is especially useful since Oz is a dynamically typed language (you do not specify the type of a variable when declaring it).