ECOTE - preliminary project

Semester: 2022L

Author: Maciej Marcinkiewicz

Subject: Interpreter for arithmetic operations with time units

I. General overview and assumptions

The program is a simple tree-walk interpreter. Its purpose is to evaluate simple arithmetic expressions, i.e. addition, subtraction, multiplication and division, with values provided as different time units – hours, minutes and seconds. The interpreter is going to operate in two modes, common for the most interpreters – interactive in which user provides statements one by another and file mode in which user provides the file with several statements that will be evaluated.

The interpreter will be implemented in C# programming language, a part of .NET platform.

II. Functional requirements

1 Operating modes

1.1 Interpreter has to work in the interactive mode
1.2 Interpreter has to accept statements in the form of a text file

2 Variables

2.1 Value assignment to a variable in the memory should be possible
2.2 Values from the variables should be possible to be used in the expressions

3 Expressions

3.1 Expressions have to be evaluated as arithmetic operations
3.2 Each evaluated expression has to be printed in the output
3.3 Expressions have to be evaluated using the smallest time unit which is present in that expression

4 Error handling

4.1 Syntax errors has to be pointed out with a specific location of the error
4.2 Use of a non-initialised variable should be reported to the user

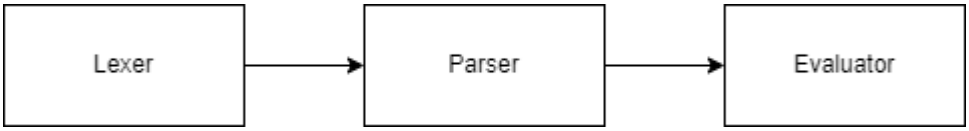
Syntax description

The syntax is rather simple – it allows only for arithmetic operations and for assignment of the values to the variables. Each statement can start with a name of a variable followed by '=' sign. The appearance of such structure indicates the process of assignment, however it is optional and user is allowed to evaluate expression without saving the result in the memory. Then any arithmetic expressions – compound or not – has to be provided. These are elementary operations such as addition, subtraction, multiplication and division which are indicated with '+', '-', '*' and '/',

respectively. Syntax allows also for a use of brackets – '(' and ')' – to indicate precedence of an operation enclosed in those brackets. Values are expressed either by numerical literals or by the names of variables. When numerical literals are used they have to be followed by a time unit expressed as the one of the three following options: 'h', 'min', 's'. Literals without those units are not permitted.

III. Implementation

General architecture



The interpreter consists of three modules: lexer responsible for lexical analysis (also known as a scanner), parser whose role is to parse given expressions with a syntax tree which is a form given to the last module – evaluator, which executes the parsed expression and provides the result.

Data structures

The first key data structure is a token. Tokens are produced by the lexer from the raw source code and consists of type description, lexeme (the actual set of consecutive characters) and location expressed as a line of code in order to use it for the error message display.

Set of token is then passed to the parser. Parser use a structure called abstract syntax tree. Such tree is a representation of the source code with respect to the rules of precedence and associativity. Compared to the parse tree this one omit some details and thus it does not represent every symbol such brackets as a single nodes – they may be combined and these combinations are based on the structure and content.

Module descriptions

Lexer

As mentioned before lexer splits the given code into several units called tokens. In order to do that lexemes have to be defined. They can be divided into:

- one-character lexemes: '+', '-', '*', '/', '(', ')'
- numerical literals with time units
- identifiers of variables
- end of file character 'EOF'.

Recognition of the first group is straightforward and it is a basic checking whether there is a match between character and defined symbol. Numerical literals are being checked from the appearance of a digit in the beginning of a new sequence. They are read until a letter is the next character. The unit is read by checking one of three defined possibilities. Identifiers of the variables are being read when a letter character is found. Then a sequence of alphanumerical characters is read until a different type of character appears as the next one. Lexer module is also responsible for checking for the syntax errors.

Parser

As mentioned parser module uses abstract syntax tree structure. The tree is created from a following grammar:

primary -> TIME | IDENTIFIER | '(' *expression* ')'
factor -> *factor* ('*' | '/') *primary* | *primary*
term -> *term* ('+' | '-') *factor* | *factor*
assignment -> ID '=' *assignment* | *term*
expression -> *assignment*
program -> (*expression* | *expression program*) 'EOF'

Evaluator

The final piece is the evaluation module responsible for the execution. Its methods return values for the provided expressions expressed as syntax trees and if necessary save results in the variables.

Input/output description

It is designed as a console program thus input is read from the standard input for the interactive mode and from file in the file mode. File mode is selected by providing an argument with a filename to the program. If it is not found, the interpreter runs in the interactive mode. Every operation which is done (in both modes) is printed to the standard output.

IV. Functional test cases

Simple arithmetic expressions

Example:

Input:
12 s + 2 min
Output:
132 s

Saving result of simple arithmetic expression to a variable and using the value

Example:

Input:
a = 20 h - 5 h
a * 3 h
Output:
a = 15 h
45 h

Compound arithmetic expressions

Example:

Input:
b = 5 min * (42 s - 7 s) + 1 h
Output:
b = 14100 s

Attempt of providing expression with wrong syntax

Example:

Input:

12 * 30 s

Output:

Syntax Error: line 1

Attempt of providing non-existing file