Kontopoulos Panagiotis
ECE student in NTUA
21/9/2020

# Parsing and Abstract Syntax Trees

## Introduction:

The goal of this mini project is to create a program that accepts as input text files and parses them searching for expressions that are then analysed into binary Abstract Syntax Trees (AST). Accepted expressions are described by the following grammar:

```
assignment ::= expression .
expression ::= term | term ("+" | "-") term
term ::= factor | factor ("*" | "/") factor
factor ::= ["-"] (identifier | integerConstant)
```

## Implementation:

The code for this project consists of one c source code file containing all the necessary functions as well as the definition of the AST. The parsing is done in one pass and the file is read in tokens, which can be either factors or operators, as described by the grammar. Most of the job is done by the three functions *make_expression(), make_term() and factor()* which are tasked with creating the corresponding node.

The first two of the above functions accept as argument a pointer to a node which will become the left child of the new node, read and store the next token and corresponding function is called in order to create the right child.

To read an expression *make_expression(make_term(factor()))* must be called, which returns a pointer to the root of an AST if a valid expression has been parsed or a NULL pointer if there is an error. Due to the nature of the provided grammar, we know that factors will be connected with term operators (‘*’, ‘/’), turning them into terms which then connect with expression operators (‘+’, ‘-’). Generalising from the above, when the first factor of a term is followed by an exp.

op. instead of a term op. , then no new node is created and a pointer to the factor is returned to be connected to the tree.
A key role is the function *read_token()* which is used to read the file and cut it into tokens, one at a time, which are stored in a global variable. It's job is also to determine the type of the token (exp op, constant, identifier, etc) and set the appropriate flags.
The resulting tree will have operators as intermediate nodes and factors as leaves, and it can be printed to the console by the *printtree()* function.

## Usage:

The program must be called with a .txt file as parameter. Multiple files can be processed with one call. The file can contain any number of sentences. Each sentence must end with the '.' character. Identifiers and constants have a maximum length of 50 characters and can have the negative sign '-' attached to them. Spaces between tokens do not matter, with the exception of '-' which when it is followed by space is exp. op. and when followed by a number or a word it denotes it as negative.
The tree that has been created is printed sideways, with the root on the left, and the leaves of the right.