Abacus
Interpreter for mathematical expressions in SML

Vågbratt, Tommy Loberg, Micael Jin, Wenting

March 6, 2014

Contents

1 Introduction

A tool that takes mathematical expressions in text form may sound nothing special, but one may have encountered difficulties to find the right symbol on a typing-in calculator to express trigonometric related functions such as "sin", "arctan" etc. However, this program (Abacus) performs as an interpreter and functions just as a "normal" calculator, but can evaluate a whole mathematical expression that is just based on text! No more time wasting looking up symbols, with ability to declare variables, to do more advanced calculation. Interesting? Well, it's just a "calculator".

2 Abacus - Calculator in SML

2.1 Design and Structure

There are three major parts of the system: Input, Compiler and Evaluation which are combined into a REPL[1]. All these parts are strictly sequential between each other. All steps inside a part are to build preparation work for next part, and all three parts form a successive execution. Executions can be achieved as many times as user need.

2.1.1 Structure Overview

Input
Handles input text, takes the input and passes it to the compiler.

Compiler

- Tokenize: Takes an expression represented as a string and split it into Tokens.
- Validate: Performs grammatical validation on tokens, but careless of priority of the functions/operators
- Translate: Convert an expression represented by Tokens from infix notation to postfix notation with priority considered

Evaluation
Compiled expression gets evaluated. A stack based "virtual Machine" evaluates the expression using the following rules:

- Numbers are pushed onto the stack.
- Variables are looked up and get pushed onto the stack.
- Functions and operators take item(s) off from the stack and push the evaluated result back onto the stack

[1]Read-Evaluate-Print Loop, more about REPL, please refer to `http://en.wikipedia.org/wiki/REPL`

2.2 Algorithms and Alternatives

Data-types used in this program: Stack, Token, Environment.

Stack is a data-structure that only has three operations, pushing (adding) data to the top and the stack, and popping (removing) data from the top of the stack, and reading the top element of the stack without modifying the stack.

Token is the data-structure used in the program to represent numbers, functions, variables, parentheses and operators.

Enviroment is a list of variables and their values.

2.2.1 Tokenize

An expression can consist of number, assignment, identifiers of variable, operator and function, open and close parenthesis and whitespace. Tokenize takes expression represented as a string and splits them into token characterized as Number, Variable, Assignment, Function, Operator, Open and Close parenthesis

- Number (*ex: 1.2 , 0.9 , 34.5 , 0*):
  1. *Start*: take in digit 0-9.
  2. if 0 encountered, either is a 0, *or* followed by a decimal point with 0-9 combinations
  3. if 1-9 encountered, either *1)*:followed by 0-9 combinations *or 2)*: 0-9 combinations followed by a dot(.) with 0-9 combinations *or 3)*: followed by decimal point with 0-9 combinations
- Identifier (*ex: a1 , Ea9d , d*):
  1. *Start*: take in alphabet a-z, A-Z until first digit 0-9 encountered
  2. check if exist in function or operator list
     - if exist recognized as an operator or a function, terminate and back to *Start*
     - if not exist in list, take in the digit encountered and keep on reading a-z, A-Z, 0-9
- Symbolic Operator: take in symbolic operator (such as. *,/,+,-, etc)
- Parentheses
  - take in left parenthesis (, terminate and back to *Start*
  - take in right parenthesis), terminate and back to *Start*
- Whitespace: whitespace " " encountered, terminate and back to *Start*

2.2.2 Validate

Validate is achieved with recursive descent parser.
Validate is implemented using a recursive descent parser algorithm. It is made from a number of mutually recursive states, the states being:

A has two constructs, either Variable = A, or E

E is an expression and has two constructs, N op E, and N

N has two constructs, either its T, or –T

T has four contructs: Number — Variable — Function on N — (E)

- $A \Rightarrow Var = A|E$
- $E \Rightarrow NopE|N$
- $N \Rightarrow T| - T$
- $T \Rightarrow Number|Variable|FunctionN|(E)$

2.2.3 Translate

The method for converting an expression in infix notation to postfix notation makes use of two lists, the first one holding the input and the second one holding the output, and a stack that holds the operators, functions and parentheses. This algorithm is called the *Shunting-yard algorithm* and was invented by *Edsger Wybe Dijkstra*[1].

2.2.4 Evaluate

Evaluate is implemented with an algorithm that makes use of an Enviroment, a stack and a list containing the input. It takes a list of Tokens as input, a stack and an enviroment that holds all the variables. After the evaluation is done the result will be added to a Variable called "ans" that is inserted into the Enviroment

2.2.5 Alternative algorithms

The main reason for chosing to first convert it from postfix to infix notation before evaluating the expression is that it would be easier for us to implement it compaired to building an abstract syntax tree. Another reason for our choice of algorithms is that the method of converting an expression from infix notation to postfix notation and then evaluating it was briefly mentioned during one of the lectures which sparked an interest for this project.

In the end both methods would give the same result, and for our purpuse our method is fast enought.There are other ways of evaluating mathematical expressions. Instead of first converting the expression from infix notation to postfix notation using the *shunting-yard algorithm* and then evaluating it, an abstract syntax tree could be built while reading the input. That could be achieved with an algorithm similar to the one used in this program to validate the expression. That is, using an *top down recursive descent algoithm*[2] If an abstract syntax tree was used the expression could then be evaluated by simple traversing the tree and evaluate the operators and functions as they are encountered.

The main reason for chosing to first convert it from postfix to infix notation before evaluating the expression is that it would be easier for us to implement it compaired to building an abstract syntax tree. Another reason for our choice of algorithms is that the method of converting an

expression from infix notation to postfix notation and then evaluating it was briefly mentioned during one of the lectures which sparked an interest for this project.

In the end both methods would give the same result, and for our purpuse our method is fast enought.

2.3 Implementation

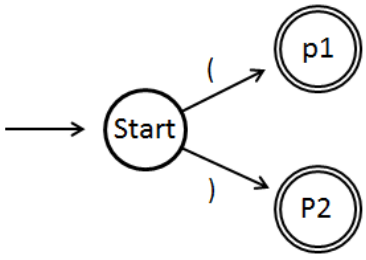
In this section the functions Tokenize, Validate, Translate and Evaluate will be presented more in detail separately as followed.

2.3.1 Tokenize implemitation

The first step in order to evaluate an input expression is to break up into its fundamental parts, called tokens. A token generally consists of two parts, a type or category and the string match when it was created. The categories of tokens are the following:

- Number
- Variable
- Function
- Operator
- Assignment
- Negate
- Open (parentheses)
- Close (parentheses)

In order to recognises a token and decide which category it belongs to a, tokenize combines a series of transition diagrams for recognizing each token into a larger more complex diagram. The concept is quite simple, it consist of a states (which are drawn as circles) and transitions (which are drawn as arrows) between them. A state is called accepting (which is marked with double circles) if terminating in it means a valid token was found. Transitions have a condition



check that must be satisfied to move to state it points to.

In figure(Open,Closed) above show the transition diagram for parentheses. The initial state is labelled start. Start is non-accepting and has two to transitions. The top one goes to p1 if and only if the next character is "(". The bottom one goes to p2 if and only if the next character is ")". Neither p1 or p2 has any transitions, but both states are accepting and terminating in

them will generate the corresponding token.

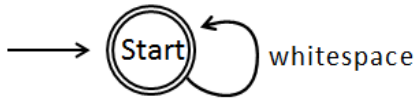
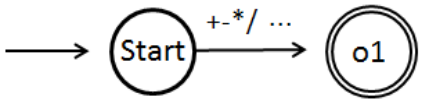
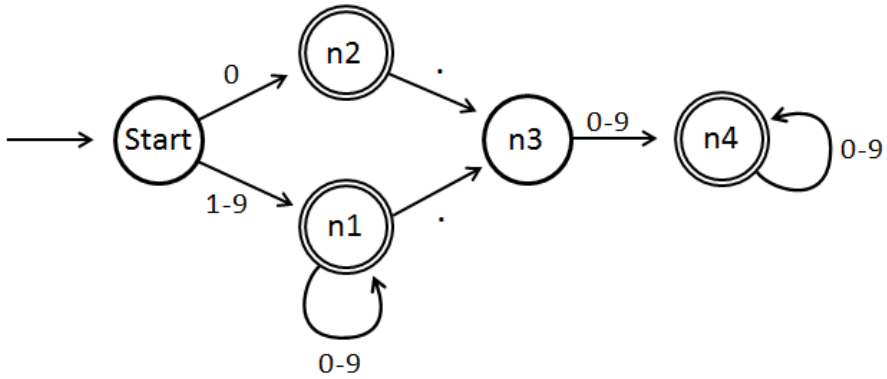


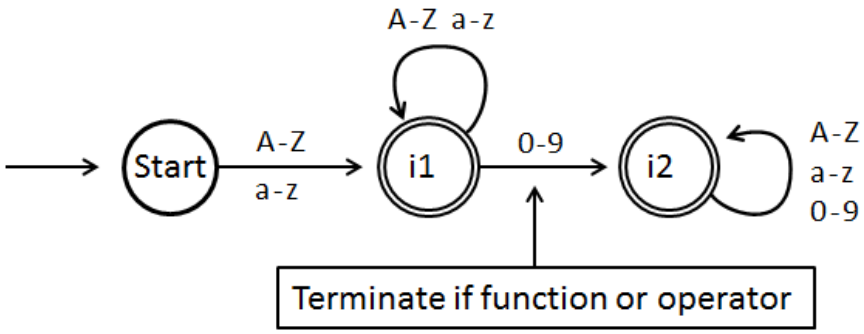
Figure (whitespace) show the transition diagram for whitespace. Whitespace is important in order to separate tokens, but no token is generated for whitespace because once an expression is split into tokens whitespace is obsolete.



Figure(symbolic operator) show the transition diagram for symbolic operator. In order to transition from start to the accepting state o1 next character have to match a symbolic operator such as "+", "-", etc.



Figure(number) show the more complex transition diagram for matching numbers. From the initial state Start has two transitions, "1-9" to n1 and "0" to n2. The accepting state n1 has two transitions, "0-9" to itself(n1) and "." To n3. The state n2 is also accepting, but only have one transition, "." to n3. The state n3 only have one transition, "0-9" to n4. Notice that n3 is non-accepting. The last state n4 is accepting and transitions back to itself on "0-9".



Figure(identifier) show the transition diagram for functions, variables and non-symbolic operator(div, mod, etc). The initial state Start has the transition, on "A-Z" or "a-z" to i1. The state i1 is an accepting state and it has two transitions, "A-Z" or "a-z" to itself(i1) and "0-9" to i2. The transition "0-9" to i2 is special because before the character "0-9" is added to the characters read before, a lookup to see if the characters read before match a non-symbolic operator or function. If a match is found the corresponding token is generated. The accepting state i2 only have one transition, "A-Z" or "a-z" or "0-9" to itself (i2). Terminating inside of i1 or i2 will trigger a lookup to see if the characters found match a non-symbolic operator or function. If a match is found the corresponding token is generated, otherwise a variable token

is generated. This means that non-symbolic operator and function have higher priority than variables. Also Non-symbolic operator has higher priority than function.

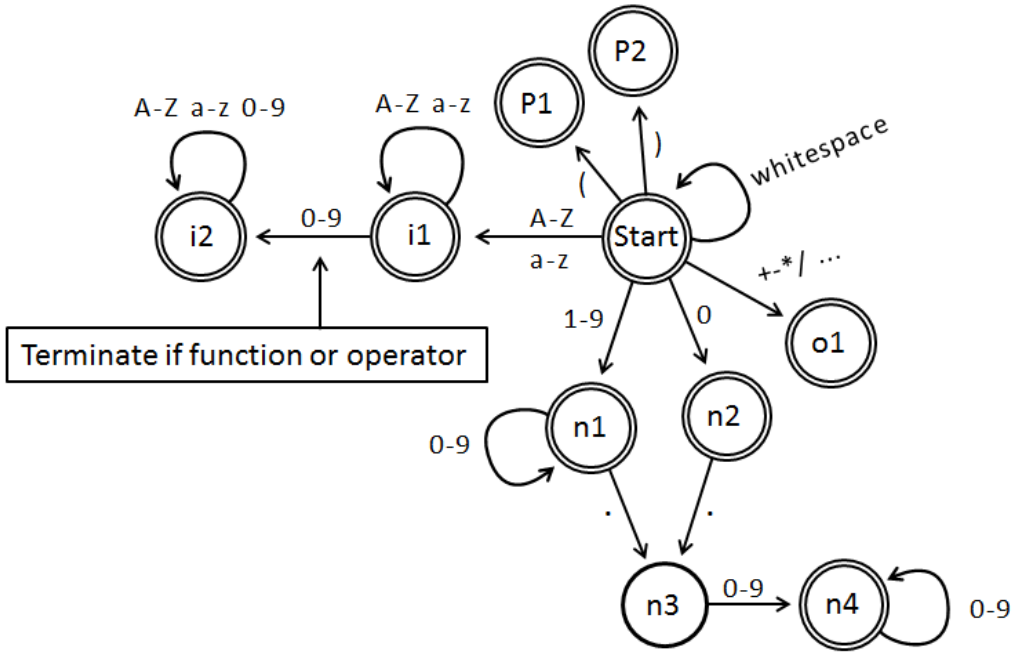


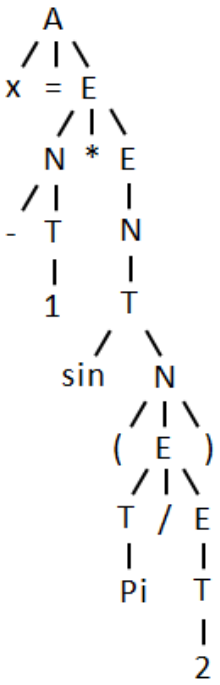
Figure (transition diagram) show how the diagrams above are combined. Tokenize works recursively when any exception case is reached. Tokenize generate tokens by recursively pulling tokens of the input.

2.3.2 Validate implemetation

- $A => Variable = E|E$
- $E => NoperatorE|N$
- $N => T| - T$
- $T => Number|Variable|FunctionN|(E)$

Expressions are built up in a recursive way, for example "1+2+3" Validate is implemented using a recursive-descent algorithm. The first token from input list starts from A which has two constructs and matches to either Variable or E, then gets into respective matching cases. Such match is constructed with other three parts: E, N and T. As shown above, all these parts can be a partial construct to the other, they are mutually recursive. Once a token gets validated through such recursive process, and not all the input has been consumed, the next token will start from A and get validated just as what happened to the first token. Successive validation happens recursively, and invalid validation returns immediately.

Expressions are built up in a recursive way, for example "1+2+3" Validate is implemented using a recursive-descent algorithm. The first token from input list starts from A which has two constructs and matches to either Variable or E, then gets into respective matching cases. Such match is constructed with other three parts: E, N and T. As shown above, all these parts can be a partial construct to the other, they are mutually recursive. Once a token gets validated through such recursive process, and not all the input has been consumed, the next token will start from A and get validated just as what happened to the first token. Successive validation



happens recursively, and invalid validation returns immediately.

2.3.3 Translate implemetation

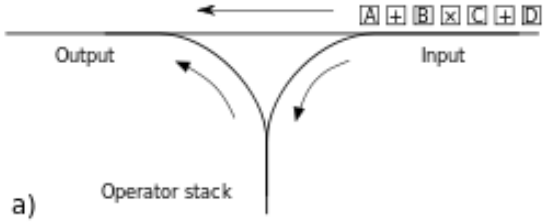
The algorithm is implemented like this in pseudo-code:

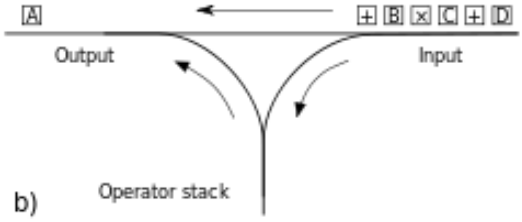
- While input is not empty, read a Token
  - If Token is a number or a variable, add it to the output list
  - If Token is a left parenthesis, add it to the operator stack
  - If Token is a function or an operator, check the priority of the top element of the operator stack.
    - if the operator/function on the top of the stack has higher or equal priority to Token, push it to the output list, repeat until Token has lower priority than the top element of the stack.
    - if priority of Token is higher, put it on the output list
  - If Token is a right parenthesis, pop elements off the stack to the output list until a left parenthesis is found, then discard both the left and right parenthesis.
  - If the input list is empty:
    - While the stack is not empty:
      - Pop elements off the stack to the output list.
  - if both the input list and the stack are empty:
    Reverse the output list.

The expression has now been converted from infix notation to postfix notation and is now ready to be evaluated.

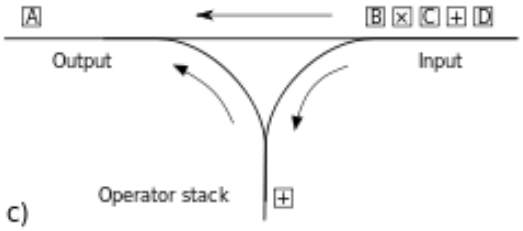
Example: $A + B * C + D$

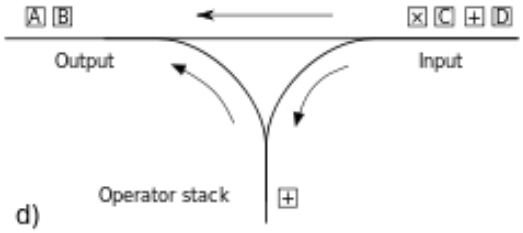
A is a number, so move it to the output list.



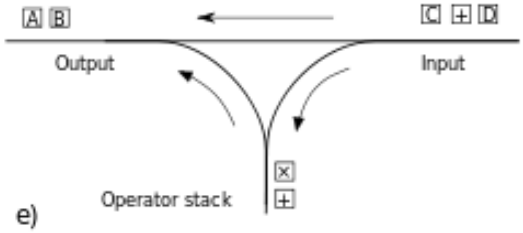


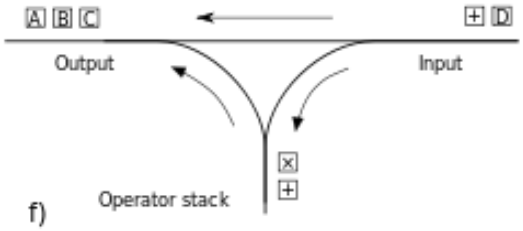
+ is an operator, push it to the stack.
B is a number, put it on the output list.



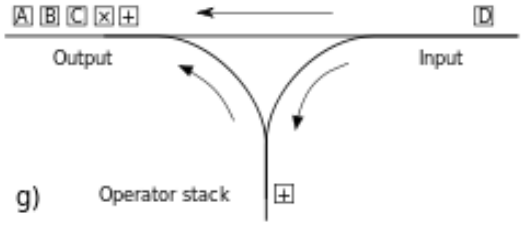


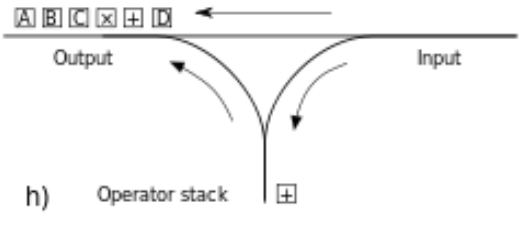
** is an operator and has higher priority than +, push it to the stack.*
C is a number, put it on the output list.



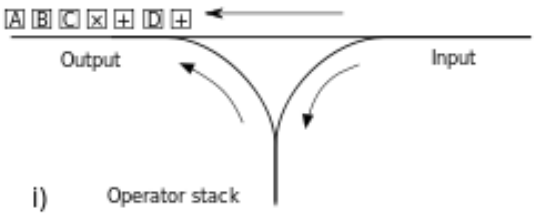


*+ is an operator with lower priority than *, pop * off the stack to the output list.*
+ is an operator with equal priority to +, pop + off the stack to the output list.
D is a number, put it on the output list.





Input list is empty, so pop everything off the stack to the output list.



[2]

[2]Images from wikipedia. `http://en.wikipedia.org/wiki/File:Shunting_yard.svg`

2.3.4 Evaluate implemetation

The algorithm is implemented like this in pseudo-code:

- While input is not empty, read a Token
  - If Token is a number then push the value of the number to the stack.
  - If Token is a variable
    - Get the value for the variable from the Enviroment and push it to the stack.
    - If the variable can't be found in the Enviroment it is undefiened, raise an exception.
  - if Token is a function or operator, pop off as many arguments as the function needs from the stack, evaluate the function and push the result back to the stack.
- If input is the empty list, then everything on the stack has been evaluated and only has one element, this element is the result.

After the expression has been evaluated, the result is put in to the environment as the variable "ans".

2.4 General Analysis

Abacus is a tool with feature of being handy and flexible, though user might find it clueless when error message happens. When expressions become long and more advanced, it can be more time-consuming to get result.
Logotype is presented and there is a user manual built into the program, by typing "help", user can get user guide on how to use the program.

Test cases are included for every source code file.

It is easy to add more support for functions and operators because priority is not part of the validation but is handled in the translation part.

2.4.1 Furture Development

Even though the project started out as an idea to make a simple calculator it has become quite powerful. The user is able to to use most functions available on expensive calculators and store variables for later use with a user friendly interface. The program can very easily be expanded upon since the way functions and operators are stored is very flexible. Allowing the user to define functions could be implemented without much effort, unfourtunitly this is a feature that had to be skipped due to time constraints, but it would be implemented in a similar fashion to variable assignment.

2.4.2 Sustainability

One major flaw in the program is that error messages aren't always very descriptive. The reason for this is that handling exceptions in Standard ML can be a bit tricky. Another side effect of it being hard to handle exceptions is that in order to keep the main loop running when exceptions are raised a new instance of the main function are run on top

of it. This could in theory make the program crash as the available memory would be depleted, though this would take a very long time.

3 Conclusion

In the end we are very satisfied with the result, it is a very functional program that has almost all of the features we had planned for. We picked this project knowing that we would be in for a challange. Before we could start writing the code we had to read about different ways of solving the problems we expected to encounter. Writing the parser that is used in the Validate part was especially problamatic but by reading more about top down recursive algorithms we managed to solve it.

In the end we are very satisfied with the result, it is a very functional program that has almost all of the features we had planned for.

4 Simple Guide for Simple Calculator

Typing "help" in the prompt will show a brief overview of the available functions, operators and default variables (such as Pi, e). Typing "logo" will display the logo for Abacus. Typing "credits" will display information about the creators of the program. Upon starting the application the user will be presented with a command line interface. The user simply types in the expression that is to be evaluated.

Typing "help" in the prompt will show a brief overview of the available functions, operators and default variables (such as Pi, e). Typing "logo" will display the logo for Abacus. Typing "credits" will display information about the creators of the program.

4.1 Examples

To get the value of the previous evaluated expression typ "ans". To evaluate the expression $3 + (3 \cdot 4)$ simply input the expression in the prompt. To use the sin function, simply type *sin expression* where *expression* can be any expression. To assign a variable to the value of the same expression, type $x = 3 + (3 \cdot 4)$, where x can be any identifier.

To get the value of the previous evaluated expression typ "ans".

References

[1] MR 34/61 Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60 1961

[2] Alfred Aho, Monica Lam, Ravi Sethi, Jeffry Ullman
Compilers: Principles, techniques and tools, second edition. Chapter 5, page 338
Pearson internation edition 2006
ISBN: 978-0-321-49169-5