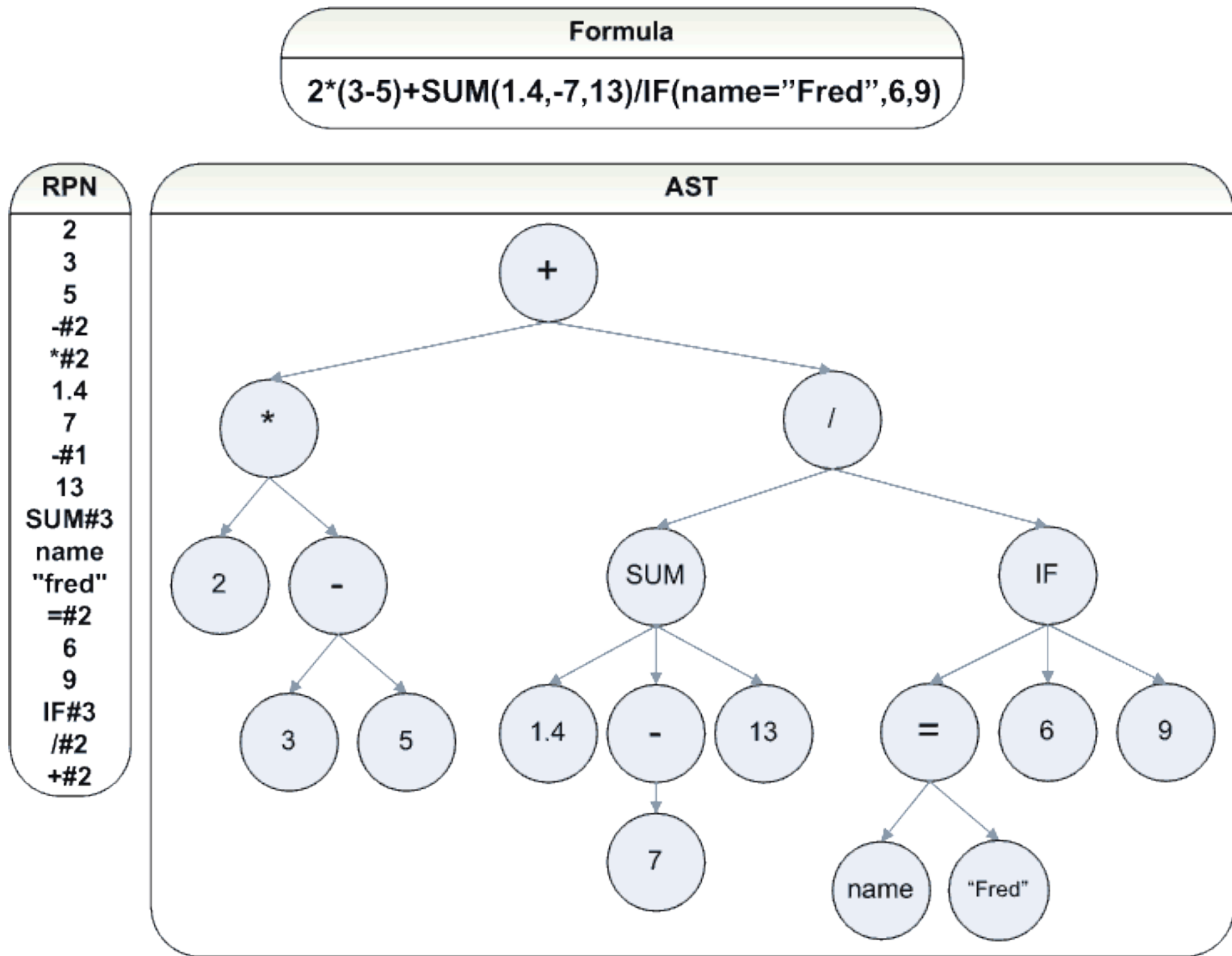
EVALUATING MATH FORMULAS

Transforming Excel-Like Text into Code



Robert Stehwien

CPSC 5115 with Dr. Bosworth

TABLE OF CONTENTS

Introduction 3
Translation and Execution 3
Data Structures - Tokens, Stack, and Queue 4
    Reverse Polish Notation (RPN) or Postfix 5
Grammar 5
    Syntax 5
    Tokens 6
    Primitives 6
    Operator Precedence 7
Lexer 8
Parser - Shunting Yard Algorithm 9
Interpreter - RPN 10
Summary 11
Bibliography 12

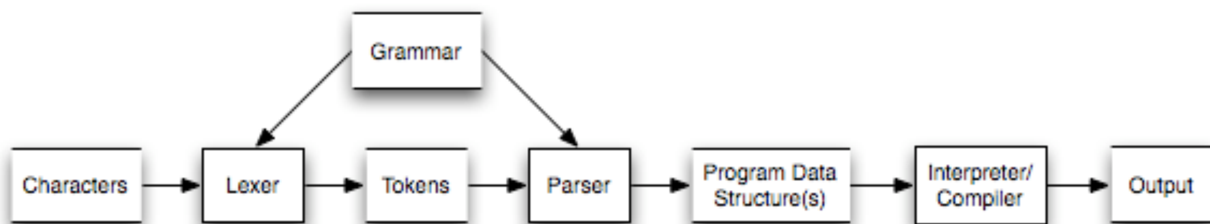
Introduction

Transforming text into code is a well known problem and is executed by every compiler and interpreter. The following paper discusses algorithms used to evaluate mathematical or Excel-like formulas such as '2*(3-5)+SUM(1.4,-7,13)/IF(name="Fred",6,9)' into an executable form. While this is but a small subset of the problem faced by compilers/interpreters for more complex languages like C++, Java, Ruby, etc. , this paper does show the pieces involved in more complicated problems.

The algorithms analyzed are the "shunting yard" algorithm by Edsger Dijkstra (around 1960) and "reverse polish notation" (RPN) by Charles Hamblin (mid-1950's). These algorithms were chosen for their relative simplicity and applicability to the problem at hand. Finally the paper will briefly discuss algorithms commonly used for processing more complicated languages.

Translation and Execution

The following diagram shows the typical process for transforming a stream of text into action:



1. A stream of characters is read from a file or string and fed into a *lexer*.

2. The *lexer* transforms the stream of text into an array of *tokens* based on the *grammar* of the language. Tokens are a collection of characters with a specific meaning and are typically defined in the grammar and recognized in the character stream by using regular expressions. The grammar describes the tokens of the language and how they relate to the language syntax.

3. The *tokens* are read by a *parser* which transforms those tokens into a *data structure* representing the language based on the syntax defined in the grammar. The data structures are typically in the form of parse trees or abstract syntax trees; possibly with additional data annotating the structure. For our purposes the program data structure will be a RPN stack.

4. The *program data structure* is read by the *compiler* or *interpreter* to either create machine code that will create output when executed in the case of a compiler or directly interpreted to create output. Sometimes an intermediate transformation will be called to optimize code. For example the operations "I + 1 - I) could be simplified to "1" since "I" and "-I" cancel each other out.

Data Structures - Tokens, Stack, and Queue

The data structures for the "shunting yard" and RPN algorithms are quite simple. A token object or structure is needed to represent each token. Tokens will be placed in a stack or queue.

- **Token** - Represents each token parsed from the character stream. A token has the following attributes.
  - **Text** - The text of the token as a string.
  - **Location** - Where in the character stream the token began. Used for error reporting.
  - **Type** - The type of token (see grammar below).
  - **Operands** have the following attribute
    - **Data** - The actual data being stored as either a Number, Boolean, or Text. Variables return the data by doing a lookup on their current data value.
  - **Operators** have the following method and attribute (a function can be considered to be a special type of operator).
    - **Execute** - Performs the operation with a given stack of Operands.
    - **Precedence** - The order of precedence for each operator
    - **NumParams** - The number of parameters needed by the operator
- **Stack** - A last in first out (LIFO) abstract data type with the following methods:
  - **Push** - Put an item on the top of the stack.
  - **Pop** - Take an item off the top of the stack and return it.
  - **Peek** - Look at the item on the top of the stack.
  - **Length** - Length of the stack.

- **Queue** = A first in first out (FIFO) abstract data type with the following methods:
  - **Enqueue** - Add a new item to the end of the queue.
  - **DeQueue** - Take the first item off the top of the queue and return it.
  - **Length** - Length of the queue.

Reverse Polish Notation (RPN) or Postfix

Standard mathematical operations are written in infix notation (ex: '3+5*7'). In infix notation the operators are written infix-style between the operands. The RPN notation writes the operator after all of its operands. For example the RPN equivalent of '3+5*7' is '357*+', which means that you multiply 5*7 then the result of that is added to 3. The process will become more clear in the interpreter algorithm discussion.

Grammar

Language grammar can be written in several forms. Below is the grammar for the Excel-like language written in a modified version of Backus–Naur form (BNF).

- Each row in the table represents "SYMBOL → <expression with symbols>"
- [optional] - Atoms enclosed between brackets are optional.
- () - Atoms are enclosed between parenthesis for grouping.
- '' - Atoms enclosed between single-quotes are literal characters.
- * - Indicates zero or more of the previous atom.
- + - Indicates one or more of the previous atom.
- | - A choice of two or more atoms.

Syntax

Below is the grammar representation of any given Excel-like expression. Think of the syntax as describing the structure of a sentence in the english language. The syntax assembled by the parser from the tokens.

EXPR	→	OPERAND | FUNEXPR | ([OPEN] EXPR OPERATOR EXPR [CLOSE])+
FUNEXPR	→	FUNCTION OPEN PARAM CLOSE

PARAM	→	EXPR[SEPARATOR EXPR]*

Tokens

While the syntax represents a sentence, the tokens represent the words of that sentence and are extracted from a stream of characters by the lexer.

OPEN	→	‘(‘
CLOSE	→	‘)’
SEPARATOR	→	‘,’
START	→	‘:=’
OPERATOR	→	MATHOP | STRINGOP | BOOLEANOP
FUNCTION	→	‘SUM’ | ‘AVERAGE’ |’IF’ | ‘MID’
OPERAND	→	NUMBER | BOOLEAN | TEXT | VARIABLE
WS	→	(‘ ‘ | ‘\t’ | ‘\n’)+
SPECIALOP	→	OPEN | CLOSE | SEPARATOR | START

Primitives

Tokens are formed up of several primitive atoms (which can in turn be formed of more primitive atoms).

DIGIT	→	(‘0’ .. ‘9’)
ALPHA	→	(‘a’ .. ‘z’) | (‘A’ .. ‘Z’)
VARCHAR	→	ALPHA | ’.’ | ‘_’
TEXTCHAR	→	all printable characters except “
ESCQUOTE	→	\”’
INT	→	DIGIT+
FLOAT	→	[DIGIT+]’.’ DIGIT+
POS	→	‘+’
NEG	→	‘-’
MATHOP	→	POS | NEG | ‘*’ | ‘/’ | ‘+’ | ‘-’ | ‘^’ | ‘%’

STRINGOP	→	‘&’
BOOLEANOP	→	‘AND’ | ‘OR’ | ‘NOT’ | ‘=’ | ‘>’ | ‘<‘ | ‘>=’ | ‘<=’ | ‘<>’
NUMBER	→	INT | FLOAT
TEXT	→	‘“’(TEXTCHAR | ESCQUOTE)*’”’
BOOLEAN	→	‘true’ | ‘false’
VARIABLE	→	ALPHA (VARCHAR)* | ‘@’VARCHAR+’@’

Operator Precedence

Grammar would not truly be complete without noting which operators take precedence over one another. An operator with a lower precedence is processed before operators with higher precedence. So in the expression “3+5*7” the operator * has lower precedence than the operator +, so * is processed first.

Operator	Description	Precedence in Calculations
	Unary Operators	
-	Negation (operates on value to its right)	1
+	Positive (operates on value to its right)	1
%	Percent (operates on value to its left)	2
	Mathematical Operators	
^	Exponentiation	3
*	Multiplication	4
/	Division	4
+	Addition	5
-	Subtraction	5
	Text Operator	
&	Concatenation	6
	Logical Operators	
=	Equal to	7
<	Less than	7
>	Greater than	7

<=	Less than or equal to	7
>=	Greater than or equal to	7
<>	Not equal to	7
AND	Logical AND	7
OR	Logical OR	7
NOT	Logical NOT	7

Lexer

The lexer's duty is to break the stream of characters into tokens that are the input to the parser. This is done with string matching algorithms such as Boyer-Moore-Horspool or the Ken Thompson regular expression algorithm. Many languages have regular expressions as part of the language or available as a library.

Since regular expressions are so readily available and the modified BNF notation can quickly render regular expressions for each token, I will not discuss tokenization at length. For my purposes, I used regular expressions that were part of ActionScript (ECMAScript), Ruby, or Perl. Since the shunting-yard algorithm below doesn't need look ahead (only needs to look behind one token), the lexer was made part of the Parser.

Regular expressions for an input string of size n can be tested against a regular expression of size m can have poor performance to good performance $O(n+2^m)$ or O(nm) depending on implementation. All of the regular expressions used for the Lexer are not looking for the expression anywhere in the expression; instead there is a list of regular expressions, each of which expects to find the expression starting at the first character. This should reduce the regular expression search to O(m) (linear at any rate).

In addition to the above optimization, the tokens are ordered from most to least specific to ensure that the exact match of 'IF' is looked for before the more general variable match of 'ifVariableB'. One additional optimization is added to add the search for the OPEN token after finding a FUNCTION.

The lexer algorithm tokenizes the text from left to right in a single pass and looks like this:

```
Tokens = [
  [/^+/, POS],
  [/^-/, NEG],
```

```
  ...
  [/^IF\w+\(/, FUNCTION]
  ...
]
Token getNextToken(string, start)
  Tokens.foreach {|i|
    type = Tokens[i].match(string, startChar)
    if type is valid
      return createToken(type, string, startChar)
  }
  return null
```

The time for finding the next token will be O(n) where n is the number of tokens times the time to find each token O(m).

Parser - Shunting Yard Algorithm

The shunting yard algorithm was invented by Edsger Dijkstra (around 1960) and transforms infix notation into postfix notation and was thus named for the similarity to how a railroad shunting yard processes trains from one track to another. The algorithm parses the tokens from left to right in a single pass and has O(n) time complexity where n is the number of tokens. There is no need to look ahead, and other than the stacks used by the algorithm, only the last token needs to be kept for syntax comparisons.

When an OPEN operator is found, the algorithm recursively calls the parser to process up until the CLOSE operator. This handles keeping the proper order of nested () and functions. To keep each recursive call behaving in the same manner the first call initializes the lastToken to the special START token.

```
parse(string, curChar = 0, lastToken = START)
  queueTokens.empty
  stackOperators.empty
  firstToken = lastToken
  numParams = 0
  while (token = getNextToken(string, startChar)) != null and
      curChar != string.lastChar)
    curChar += token.Text.length
    if (token.Type == WS)
      continue // skip whitespace
    if (lastToken.Type == FUNCTION and token.Type != OPEN)
      throw exception "function must be followed by OPEN"          lastToken = token
    if numParams = 0 and token.Type != SEPARATOR
      numParams = 1
    switch token.Type
      case OPERAND
```

```
        queueTokens.queue(token)
      case CLOSE
        if firstToken.Type != OPEN
          throw exception "mismatched parens"
      case OPEN
        newParser.parse(string, curChar, lastToken)
        queueTokens.queue(newParser.queueTokens)
        if (lastToken.Type == FUNCTION)
          lastToken.numParams = newParser.numParams
      case SEPARATOR
        numParams += 1
        until stackOperators.peek.Type == OPEN
          queueTokens.queue(stackOperators.pop)
      case FUNCITON
        stackOperators.push(token)
      case OPERATOR
        // handle case of POS and NEG since they are the same as plus and minus
        while stackOperators.peek.Precedence < token.Precendence
          queueTokens.queue(stackOperators.pop)
        stackOperators.push(token)
  if (token == null and curChar != string.lastChar)
    throw exception "invalid token found"
  while stackOperators.peek.Type != START or OPEN
    queueTokens.queue(stackOperators.pop)
  if lastToken.Type == CLOSE and queueTokens.peek.Type != OPEN
    throw exception "mismatched parens"
```

Interpreter - RPN

The RPN interpreter makes one pass through all the tokens queued by the parser executing the following algorithm:

```
interpret(queueTokens)
  stackOperands.empty
  while queueTokens.length > 0
    token = queueTokens.dequeue
    if token.Type = OPERAND
      stackOperands.push(token)
    else // token is a FUNCTION or OPERATOR
      if stackOperands.length < token.NumParams
        throw exception "too few parameters"
      arguments = stackOperands.pop(token.NumParams)
      stackOperands.push(token.execute(arguments))
  if stackOperands.length != 1
    throw exception "too few or too many values"
  return stackOperands.pop
```

The time complexity of the Parser is O(n) where n is the number of tokens plus the time it takes to execute any operators. This is the quickest step of the whole algorithm. The lexer and parser need to be performed each time an expression changes, but only when the expression changes. Unless there are variables in the tokens, the expression doesn't need to be interpreted unless the expression changes.

Summary

There are other ways an Excel-like language could be interpreted. Tokens could be stored as more compact bytecodes and instead of using stacks and queues and abstract syntax tree (AST) or parse tree could be used. Even the shunting yard algorithm could use an AST and post-order traversal to walk the tree in the interpreter. The parser too had many choices to choose from including top-down parsers and bottom-up parsers.

For the given problem domain of Excel-like formula parsing, the combination of a shunting yard parser and RPN interpreter was found to be cleaner. The data structure was simpler and easier to interpret. While not many people are familiar with RPN, it isn't difficult to understand.

Bibliography

Parr, Terence. “The Definitive ANTLR Reference: Building Domain-Specific Languages” The Pragmatic Bookshelf, 2007

“Shunting yard Algorithm” http://en.wikipedia.org/wiki/Shunting_yard_algorithm, July 2007

“Reverse Polish Notation” http://en.wikipedia.org/wiki/Reverse_Polish_notation, October 2007

Cox, Russ. "Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)". http://swtch.com/~rsc/regexp/regexp1.html. January 2007

“Regular Expression” http://en.wikipedia.org/wiki/Regular_expression. October 2007

“Backus - Naur Form”. http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form October 2007.

Norvell, Theodore. "Parsing Expressions by Recursive Descent" http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm 1999 - 2001