A First Arithmetic Parser

The purpose of this study unit is to guide you through a first success in creating an arithmetic parsing computer – an operator-precedence parser that supports some minimal parsing features such as confix (grouping) operators, function application, and operator name disambiguation by context.

The first section covers mostly vocabulary and theory. Then we'll set about implementing in the operator precedence parsing algorithm a c++ implementation. This will be followed by a natural application: creating a graphical plot of a function using OpenGL. Finally we'll attempt an emulation – or imitation – of the old Apple II game, Algebra Arcade, which is essentially a curve fitting game.

Background and Theory

An operator is something (typically a symbol like '+' or '*') that denotes a mathematical operation. An operator does operations on operands. We can classify operators as coming before, after or between operands. Consider the two character expression, "–2." This is an example of a *prefix* (before the operand) negation operator. The sum "2+3" illustrates the *infix* (between operands) addition operator. The negation operator is an example of a *unary* operator. A unary operator takes only one operand. A *binary* operator takes two inputs; like the infix operator.

The *arity* of an expression is the number of operands it takes, typically unary or binary.

An *arithmetic expression* is a sequence of operators and operands where operators fall into one of the following categories:

Operator Type	**Arity**	**Placement**	**Examples**
prefix	unary	prior to operand	Unary minus (negation)
postfix	Unary	after operand	Factorial
infix	binary	between operands	Addition, multiplication, division & exponentiation
confix	Unary	surrounding operand	Parentheses, half-open ranges
function application	binary	after first operand and surrounding second operand	Elementary functions like ln(x), array indices such as a[5]

The `confix` and `function application` operators are parsed using an open symbol and a close symbol. The "open" symbol to the left-hand side and "close" symbol to the right.

Constructing the parser

A *stack* is a way of storing data in a pile. The parser we're going to construct uses two separate stacks: an `opr` stack to *push* operators onto and a `val` stack to push operands on. performs two main kinds of operations:

> **Shift**: put operators on top of the `opr` stack and operands on the `val` stack.
> **Reduce**: take an operator off the `opr` stack and one or more operands off the `val` stack and put the result of the operation on the `val` stack.

A look-ahead parser will wait to perform some operations until more of the expression is read. To keep things simple, we'll be using a shift/reduce parser with zero look-ahead. Any operand in the input stream is immediately shifted onto the operand stack; operators are immediately shifted onto the operator stack only if the operator stack is empty. Otherwise, the following table determines the action of the parser depending on the type of the operator on top of the operator stack and on the type of the current operator token.

Parsing table

		Current operator						
		Prefix	**Postfix**	**Infix**	**Confix Open**	**Confix/ Function Close**	**Function Open**	**End of Input**
Top of Stack	**Prefix**	shift	precedence	precedence	shift	reduce	precedence	reduce
	Postfix	-	reduce	reduce	-	reduce	reduce	reduce
	Infix	shift	precedence	precedence/ associativity	shift	reduce	precedence	reduce
	Confix Open	shift	shift	shift	shift	shift	shift	reduce
	Confix/ Function Close	reduce	reduce	reduce	reduce	reduce	reduce	reduce
	Function Open	shift	shift	shift	shift	shift	shift	reduce

Description of parsing actions

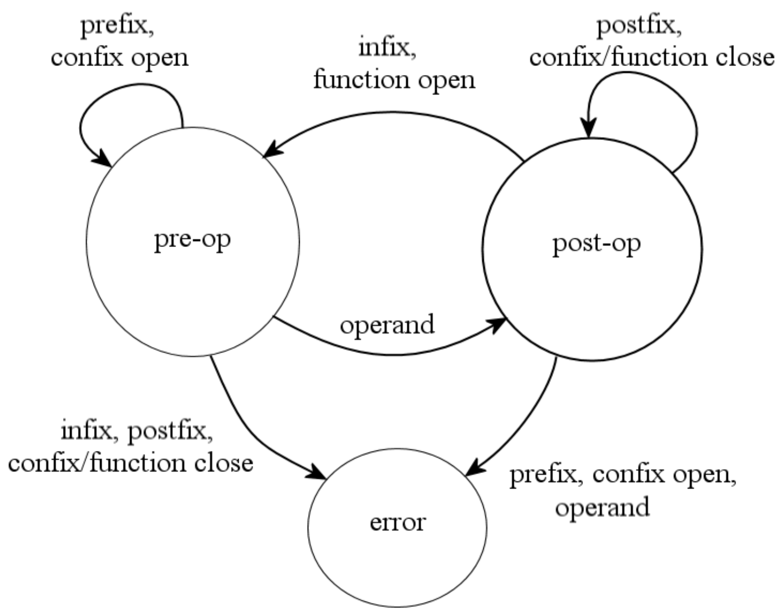
- A `shift` operation pushes the current operator token onto the operator stack (and maybe gets the next symbol too.)
- A `reduce` operation pops the operator token off the top of the operator stack, and then pops the appropriate number of operands from the operand stack: applying the operator to the operand(s) and pushing/replacing the result on the operand stack appropriately. Reduction of `confix` operators and of `function` application requires popping two operators (open and close) off the operator stack. The name of the operation may be another operand.
- A `precedence` operation (computed by parseTable in the instance presented here) determines the relative precedence of the operator on top of the operator stack (`tok`) and the

current operator (`pretok`).

  - If `pretok` has a lower precedence than `tok`, `shift`.
  - If `pretok` has a higher precedence than `tok`, `reduce`.
- A `precedence/associativity` operation first compares the precedence according to the `precedence` operation: if the precedence is equivalent, associativity is considered:
  - If top associates left of current, `reduce`.
  - If top associates right of current, `shift`.

Rejecting Invalid Expressions

Operator-precedence parsers are often avoided because they accept invalid strings. The shift-reduce parser as specified above will consider the expressions x + x, + x x, and x x + equivalent, even though only the first form is correct. This weakness is easily remedied with the use of the following state machine to track what type of operator or operand is expected at any given point in time.



The state machine has three states:

- The pre-op state is where confix open and prefix operators accumulate until, typically, an operand arrives, triggering the post-op state.

- The post-op state is where we can accumulate postfix operators to the operand or confix close operators and function close calls.

- The error state is entered if an invalid expression is detected by the state machine.

Disambiguation of Operator Names

The meaning of an operator's token may depend on context. For example, the unary negation and binary minus operators that use the same symbol `'-'`, the absolute-value confix operators use the same symbol '|' for both open and close. A good operator-precedence parser will support such common parsing requirements as function application, confix (grouping) operators, and operator name disambiguation.

An Example

Below is a step-by-step accounting of the operator precedence algorithm as it is used to parse the expression a*|b+c|+5^a^b using standard rules for precedence and associativity.

State	Operand Stack	Operator Stack	Token	Token type	Action
Pre			a	operand	shift
Post	a		*	infix operator	shift
Pre	a	tMul	|	confix open or confix close	disambiguate as confix open, shift
Pre	a	tMul tLAbs	b	operand	shift
Post	a b	tMul tLAbs	+	infix or prefix operator	disambiguate as infix, shift
Pre	a b	tMul tLAbs tAdd	c	operand	shift
Post	a b c	tMul tLAbs tAdd	|	confix open or confix close	disambiguate as close, reduce
Post	a (b+c)	tMul tLAbs	|	confix open or confix close	disambiguate as close, reduce
Post	a (|b+c|)	tMul	+	infix or prefix	disambiguate as infix, compare precedence, reduce
Post	(a * (|b+c|))		+	infix or prefix	disambiguate as infix, shift
Pre	(a * (|b+c|))	tAdd	-	infix or prefix	disambiguate as prefix, shift
Pre	(a * (|b+c|))	tAdd tUMin	3	operand	shift
Post	(a * (|b+c|)) 5	tAdd tUMin	^	infix	compare precedence, shift
Pre	(a * (|b+c|)) 5	tAdd tUMin tPow	a	operand	shift
Post	(a * (|b+c|)) 5 a	tAdd tUMin tPow	^	infix	compare precedence, compare associativity, shift
Pre	(a * (|b+c|)) 5 a	tAdd tUMin tPow tPow	b	operand	shift
Post	(a * (|b+c|)) 5 a b	tAdd tUMin tPow tPow	end	end	reduce
Post	(a * (|b+c|)) 5 (a^b)	tAdd tUMin tPow	end	end	reduce
Post	(a * (|b+c|)) (5^(a^b))	tAdd tUMin	end	end	reduce
Post	(a * (|b+c|)) (-(5^(a^b)))	tAdd	end	end	reduce
Post	((a * (|b+c|)) + (-(5^(a^b))))		end	end	accept

Exercises:

1. In the expression –3 * (a ^ 2 + 3) which symbols are operators and which symbols represent operands?
2. In the expression $\left(f\left(-2 * x+5!\right)\right)^\wedge 2$ there are 13 tokens.
   a. Which tokens represent operands?
   b. Which operators are unary. Which are binary?
   c. Which of the operators is infix?
   d. Which of the operators is confix?
   e. Which of the operators is prefix?
   f. Which of the operators is postfix?
3. If there is a postfix operator on top of the stack and an infix operator is read, what action takes place? Give an example of a simple expression in which this would occur.
4. If there is a postfix operator on top of the stack and a confix operator is read, what action takes place? Give an example of a simple expression in which this would occur.
5. If there is an infix operator on top of the stack and a prefix operator is read, what action takes place? Give an example of a simple expression in which this would occur.
6. If there is an infix operator on top of the stack and another infix operator is read, the result could be either to reduce or to shift. Why? Give several examples of simple expressions in which a reduction or a shift would occur.
7. Write a sentence or more to describe the purpose of the state machine and how it helps to parse mathematical expressions.
8. Complete a table like the open above, tabulating the state, the contents of the operand and operator stacks, the next token read and what type of token it is and what action is taken for each of the following expressions.
   a. 1 + 1 + 1
   b. 1 + (2 + 3)
   c. 2 * 3 / (2 + 3) ^ (4 – 6 / 3)

References:

http://www.boost.org/index.htm
http://epaperpress.com/oper/index.html
http://www.fredosaurus.com/notes-cpp/
http://souptonuts.sourceforge.net/code/desktop_calc.cc.html
http://ldp.rtin.bz/LDP/LGNET////106/chirico.html