Parser for expression with infix notation using precedence lists

Radovan Augustín
rado_augustin@yahoo.com

This document describes representation of parsed expression and algorithm for parsing expression with infix notation. Representation of parsed expression is stored in lists sorted by operator precedence and can be processed by recursive or linear non-recursive iteration to produce result of expression. Algorithm for parsing expression is single loop without using recursion or sorting.

Term dictionary

Operand value	Represents value in operand. It can be: - value itself, e.g. number or string - pointer to another expression in parentheses - variable - function call
CUO	Chain of unary operators with *Operand value*, stored in list, e.g. “+++1.0” or “*var_ptr”
Operand	Represents operand in expression. It can be: - *Operand value* - *CUO* - pointer to another list
Operator	Binary operator

Revisions:

October 2013	Initial release
May 2014	Added clarification of function `chain_of_unary_operators(token)`

Representation of parsed expression

Representation of parsed expression allows parsing of expression in single loop without recursion. It allows processing of parsed expression with recursive or non-recursive linear iteration. In case of recursive iteration, lists are processed from lowest precedence recursively to highest precedence. In case of linear iteration, lists are processed from highest precedence to lowest precedence.

Expression is parsed to one or more lists of *Operand* and *Operator*. One list contains chain of *Operand* and *Operator* with same precedence. Lists with same *Operator* precedence are stored in list and list is stored in precedence array with index equal to *Operator* precedence. If *Operand* is pointer to another list, it points to list with higher *Operator* precedence. Unary operators are stored with *Operand value* in *CUO* list. This simplifies representation of expression, also list of unary operators with *Operand value* can be evaluated with simple iteration. Operator associativity is not reflected in representation of parsed expression, because it can be solved by assigning dedicated precedence to *Operator* with different associativity or explicitly in evaluation function.

If expression contains parentheses, expression in parentheses is parsed separately creating another precedence array. Precedence arrays from expressions in parentheses are stored in list and evaluated before evaluating precedence array for main expression.

For expression of representation, examples has *Operand value*, pointer to *CUO* and pointer to list in place of *Operand*.

Example A: expression -1-+2
Precedence array index [precedence]: List of List of *Operand* and *Operator*
```
[0]:
[1]: LIST01{LIST_CUO_01, -, LIST_CUO_02}
[2]:
```

Chain of unary operators with *Operand value*:
```
LIST_CUO_01{-, 1.000}
LIST_CUO_02{+, 2.000}
```

Example B: expression 1+2*3-1/2
Precedence array index [precedence]: List of List of *Operand* and *Operator*
```
[0]:
[1]: LIST01{1.000, +, LIST02, -, LIST03}
[2]: LIST02{2.000, *, 3.000}, LIST03{1.000, /, 2.000}
```

Example C: expression 1+2*3-1>4/3-2
Precedence array index [precedence]: List of List of *Operand* and *Operator*
```
[0]: LIST03{LIST01, >, LIST05}
[1]: LIST01{1.000, +, LIST02, -, 1.000}, LIST05{LIST04, -, 2.000}
[2]: LIST02{2.000, *, 3.000}, LIST04{4.000, /, 3.000}
```

Example D: expression 1+2*3-1/2+(1+2)
Precedence array index [precedence]: List of List of *Operand* and *Operator*
```
[0]:
[1]: LIST01{1.000, +, LIST02, -, LIST03, +, PTR_PRECEDENCE_ARRAY_1}
[2]: LIST02{2.000, *, 3.000}, LIST03{1.000, /, 2.000}
```

List of expressions from parentheses:
```
LIST_PARENTHESES{PTR_PRECEDENCE_ARRAY_1}
```

Precedence array PTR_PRECEDENCE_ARRAY_1 index [precedence]: List of List of *Operand* and *Operator*
```
[0]:
[1]: LIST01{1.000, +, 2.000}
[2]:
```

Algorithm for parsing expression

Expression is processed as pairs of *Operand* and *Operator*, if expression contains only *Operand* or last item of expression is readed, then *Operator* is NULL. Thus only binary operators are expected at input, unary operators are stored with *Operand value* in *CUO*, which is type of *Operand*.

Algorithm compares precedence of previous and current *Operator* and saves *Operand* and *Operator* to list. Lists are maintained in precedence array cache and are reused for pairs with same precedence, until *Operator* with lower precedence causes to save and clear cache to precedence array. Operator is unary operator if it is first token in expression or first token after *Operator*.

Following code with comments describes algorithm:

```
Expression()
{
	while (1) {

		operand = get_operand();
		operator = get_operator();

		// If operand is only item in expression:
		// 1. Add operand to precedence array
		// 2. Parsing of expression is done
		// If operand is last item in expression:
		// 1. Add operand to current list
		// 2. Save and clear cache for all items in cache
		// 3. Parsing of expression is done
		if (operator == NULL) {
			if (list_current == NULL) {
				// Operand is only item in expression
				precedence_array[PRECEDENCE_0].push_back(new LIST(operand));
				break;
			}
			// Operand is last item in expression
			list_current->push_back(operand);
			save_and_clear_cache(-1);
			break;
		}

		precedence = get_operator_precedence(operator);

		// Init precedence_prev with first operator
		if (precedence_prev == PRECEDENCE_NONE)
			precedence_prev = precedence;

		// Init first list
		if (list_current == NULL) {
			list_current = new LIST();
			list_first = list_current;
			precedence_array_cache[precedence] = list_current;
		}

		// If operator has same precedence as previous operator:
		// 1. Add operand and operator to current list
		if (precedence_prev == precedence) {
		}

		// If operator has lower precedence than previous operator:
		// 1. Add operand to current list
		// 2. Allocate new operand containing pointer to list:
		//    - from cache get first list with higher precedence than current precedence
		//    - if list from cache is NULL, get first list in expression
		// 3. Save and clear cache with current precedence
		// 4. Get new current list:
		//    - list for current precedence from cache
		//    - if list from cache is NULL, allocate new list and store it in cache
		// 5. If current list was not allocated in step (4), set operand to NULL
		// 6. Add operand and operator to current list
		if (precedence_prev > precedence) {
			list_current->push_back(operand);

			operand = first_list_from_cache_with_higher_precedence(precedence);
			if (operand == NULL)
				operand = list_first;

			save_and_clear_cache(precedence);

			list_current = precedence_array_cache[precedence];
			if (list_current == NULL) {
				list_current = new LIST();
				precedence_array_cache[precedence] = list_current;
			} else
				operand = NULL;
		}

		// If operator has higher precedence than previous operator:
		// 1. Create new operand with NULL pointer to list as marker
		// 2. Save operand from step (1) to current list
		// 3. Get new current list:
		//    - list for current precedence from cache
		//    - if list from cache is NULL, allocate new list and store it in cache
		// 4. Add operand and operator to current list
		if (precedence_prev < precedence) {
			list_current->push_back(new OPERAND(NULL));

			list_current = precedence_array_cache[precedence];
			if (list_current == NULL) {
				list_current = new LIST();
				precedence_array_cache[precedence] = list_current;
			}
		}

		// Add operand and operator to current list
		if (operand != NULL)
			list_current->push_back(operand);
		list_current->push_back(operator);

		// Save precedence
		precedence_prev = precedence;

	} // while (1)
}
```

```
get_operand()
{
          token = token_get();
          if (token in ALLOWED_UNARY_OPERATORS) {
                    operand = chain_of_unary_operators(token);
          } else
          // if (token == '(') {
          //        // (expression)
          //        operand = parse_parentheses(token);
          //        // Add expression to list, thus it can be evaluated before evaluation of main expression
          //        list_parentheses.push_back(operand);
          // } else
          // if (token is VARIABLE_NAME) {
          //        operand = token;
          // } else
          // if (token is FUNCTION_NAME) {
          //        // function_name(expression, expression, ...)
          //        operand = parse_function(token);
          //        // If function call needs to be evaluated before evaluation of main expression,
          //        // it can be added to list
          //        list_functions.push_back(operand);
          // } else
                    operand = token;
          return operand;
}

get_operator()
{
          return token_get();
}

chain_of_unary_operators(token)
{
          while (1) {
                    operand.push_back(token);
                    if (token is OPERAND_VALUE)
                              break;
                    token = token_get();
          }
          return operand;
}

first_list_from_cache_with_higher_precedence(higher_than_precedence)
{
          for (precedence = higher_than_precedence + 1; precedence < PRECEDENCE_MAX; precedence ++) {
                    if (precedence_array_cache[precedence] != NULL)
                              return precedence_array_cache[precedence];
          }
          return NULL;
}

// Function iterates precedence_array_cache starting from higher_than_precedence, it does:
// 1. Check if last item in list is pointer to another list with value NULL, if yes, replace NULL
//    value with pointer to first list with higher precedence
// 2. Save list from cache to precedence_array and set pointer in precedence_array_cache to NULL
// If input parameter higher_than_precedence is < 0 then:
//     Step (1) and step (2) apply on all lists in cache
// else:
//     Step (1) apply on list in cache with precedence >= higher_than_precedence
//     Step (2) apply on list in cache with precedence > higher_than_precedence
save_and_clear_cache(higher_than_precedence)
{
          if (higher_than_precedence < 0) {
                    higher_than_precedence = 0;
                    all_items = true;
          } else
                    all_items = false;

          for (precedence = higher_than_precedence; precedence < PRECEDENCE_MAX; precedence ++) {
                    list = precedence_array_cache[precedence];
                    if (list == NULL)
                              continue;

                    // If latest item in list is NULL, then replace it with pointer to first list with higher precedence
                    list_item = list->back();
                    if ((list_item->type == LIST_PTR) && (list_item->list_ptr == NULL))
                              list_item->list_ptr = first_list_from_cache_with_higher_precedence(precedence);

                    if (!all_items && (precedence <= higher_than_precedence))
                              continue;

                    precedence_array[precedence].push_back(list);
                    precedence_array_cache[precedence] = NULL;
          }
}
```