CS3901 - Introduction to Data Structures

How to Parse Arithmetic Expressions

Lt Col Joel Young

One of the common task required in implementing programming languages, calculators, simulation systems, and spreadsheets is parsing and evaluating arithmetic expressions stored in strings. Arithmetic expressions are composed of binary operators, unary operators, numbers, and parenthesized sub-expressions.

Parsing a fully parenthesized expression is straightforward as each sub-expression is bracketed by open and close parentheses so order-of-operations does not need to be considered. Typical expressions (as typed by humans) rely on the standard order of operations to resolve ambiguity. For example, "`-2^3^4x6`" would parenthesize out to "`(-((2^(3^4)) x 6))`".

A typical system would perform a right-to-left pass over the input expression inserting parenthesis for exponentiation (highest-precedence) and then a (or several) left-to-right pass(es) adding parenthesis for the other operators by their precedence.

In this note we discuss a strategy for directly parsing the string from left to right handling exponentiation, addition, subtraction, multiplication, division, and negation. The result of a successful parse is an expression tree which could be evaluated or used to convert the expression to post-fix or pre-fix or fully parenthesized in-fix notation.

To help us, we are going to assume that we have a tokenizer that removes whitespace and separates the operators, parentheses, and numbers in an expression into separate strings called tokens. Most languages have such a capability either built in or in a library. For example, C++ has strtok or boost's tokenizer library. An example tokenization for `6 x 6 - (--3)` would be `Tokens: [(] [6] [x] [6] [-] [(] [-] [-] [3] [)] [)]` Notice that there is an extra set of parenthesis in the above. For simplification during the parse, we are going to force all expressions to have leading and closing parenthesis. You can think of this as similar to using sentinels in parsing a linked list.

The first thing we'll need is a means of expressing the precedence of operations. We'll use a mapping of operators to integers (could be implemented with an associative memory structure or hard-coded in a function). Our base precedence table is:

operator	priority
^	5
x	2
/	2
+	1
-	1

Notice that "x" and "/" have the same precedence. This is because they are tied for precedence and simply ordered from left to right. likewise for "-" and "+". In addition, we have some internal operators that are used for bookkeeping:

operator	priority
exponent	4
unary	3
open	0

Their use will be explained during the discussion below. Using the above table, we can compare the priority of two operators: `priority(``^'')` $\leq$ `priority(``x'')` which would be false. This will be useful for determining order of operations.

We will store our state in five data-stores. We will have two push-downs (stacks) and three booleans:

- `operators`: stack holding operators as listed in the tables above. It contains all of the operators that we haven't yet found operands for. The top operator will be the last one we've received a token for.
- `expressions`: stack holding expressions that have been partially parsed. The top expression on the stack is the last one we have processed.
- `next_is_unary` boolean that is true when the next operator to the right must be a unary operator and false otherwise. This is used to determine if the token "-" is a unary or binary operator and also to recognize ill-formed expressions like "`6 + x 7`".

- `last_was_expression` boolean that is true when the previous token was a number and false otherwise. This is used to identify ill-formed expressions like "`(6 2 + 3)`".
- `last_was_parenthesis` boolean that is true when the previous token was an open paren and false otherwise. This is used to identify ill-formed expressions like "`()5+6`".

In the end we will find that only one of the booleans are really required, but the reasoning is easier with them.

Let's return to our tokenized representation of the expression. There are four kinds of tokens which we will need to respond differently two:

1. open parenthesis
2. close parenthesis
3. operators
4. numbers

We can imagine that if we looped through our tokens from left-to-right we could use a big compound if - else statement to select which kind of token we are processing each time. We'll assume that our tokenizer loop gives us the current token in a variable `token` and the following token in a variable `next_token`.

In this context, let's consider our boolean `last_was_expression`. This will need to be set to true any time we finish handling a number token or any other expression and false at all the other times. Also, if we are handling a number token or an open parenthesis, this variable better not be true! So adding this to the above list we get:

1. open parenthesis
   (a) (*) if `last_was_expression` is true, throw an error.
   (b) (*) `last_was_expression` is false
2. close parenthesis
   (a) (*) `last_was_expression` is true
3. operators
   (a) (*) `last_was_expression` is false
4. numbers
   (a) (*) if `last_was_expression` is true, throw an error.
   (b) (*) `last_was_expression` is true

Next let's think about `next_is_unary`. Consider the close parenthesis ")". Are there any circumstances when the next token could be a unary operator after parsing this token? How about after immediately after a number? In an expression like "(6 + 6 - ...)" is the "-" ever going to be a unary operation? In both cases the answer is no. Unary operators occur only after open parenthesis and after operators. Binary operators occur in exactly the opposite circumstances. Let's annotate our branches with this idea:

1. open parenthesis
   (a) if `last_was_expression` is true, throw an error.
   (b) (*) `next_is_unary` is true
   (c) `last_was_expression` is false
2. close parenthesis
   (a) (*) `next_is_unary` is false
   (b) `last_was_expression` is true
3. operators
   (a) (*) if `next_is_unary` is true then we know that `token` is a unary operator, otherwise it is binary:
      i. true: unary operator
      ii. false: binary operator
   (b) (*) `next_is_unary` is true
   (c) `last_was_expression` is false
4. numbers
   (a) if `last_was_expression` is true, throw an error.
   (b) (*) `next_is_unary` is false
   (c) `last_was_expression` is true

Next let's handle our third boolean `last_was_parenthesis`. We don't want expressions like "`()`" to be valid. Any time we have an empty parenthesis pair, we want to catch it. The simplest way is to set our boolean to true when we've handled an open parenthesis:

1. open parenthesis
   (a) if `last_was_expression` is true, throw an error.
   (b) `next_is_unary` is true

   (c) `last_was_expression` is false
   (d) (*) `last_was_parenthesis` is true
2. close parenthesis
   (a) (*) if `last_was_parenthesis` is true, throw an error.
   (b) `next_is_unary` is false
   (c) `last_was_expression` is true
   (d) (*) `last_was_parenthesis` is false
3. operators
   (a) if `next_is_unary` is true then we know that `token` is a unary operator, otherwise it is binary:
      i. true: unary operator
      ii. false: binary operator
   (b) `next_is_unary` is true
   (c) `last_was_expression` is false
   (d) (*) `last_was_parenthesis` is false
4. numbers
   (a) if `last_was_expression` is true, throw an error.
   (b) `next_is_unary` is false
   (c) `last_was_expression` is true
   (d) (*) `last_was_parenthesis` is false

Now that we have some basic machinery in place, let's think about the simplest sort of expression possible: The number. Consider the expression "`?  6 ?`". "?" are unknown operators. Until their relative priority is known, we can't decide whether the number belongs to the left operator or the right operator. For example "`+ 6 x`" would result in the number belonging to the right side and "`x 6 +`" would result in the number belonging to the left side. Since at this point we haven't yet processed the next token, let's just hold off on deciding what to do with it. So, if we receive a number token, let's just wrap it in an expression and push it on our expression stack:

1. open parenthesis
   (a) if `last_was_expression` is true, throw an error.
   (b) `next_is_unary` is true
   (c) `last_was_expression` is false
   (d) `last_was_parenthesis` is true
2. close parenthesis
   (a) if `last_was_parenthesis` is true, throw an error.
   (b) `next_is_unary` is false
   (c) `last_was_expression` is true
   (d) `last_was_parenthesis` is false
3. operators
   (a) if `next_is_unary` is true then we know that `token` is a unary operator, otherwise it is binary:
      i. true: unary operator
      ii. false: binary operator
   (b) `next_is_unary` is true
   (c) `last_was_expression` is false
   (d) `last_was_parenthesis` is false
4. numbers
   (a) if `last_was_expression` is true, throw an error.
   (b) (*) push number in `token` onto `expressions`
   (c) `next_is_unary` is false
   (d) `last_was_expression` is true
   (e) `last_was_parenthesis` is false

What is the next thing we can readily handle? What if our current token is an open parenthesis? What do we know? Well, at this point, almost all we know is that the next token better not be a close parenthesis (which we covered above) and our previous token shouldn't have been either a number or a close parenthesis. We also know one more thing: Everything that follows to the next close parenthesis (which had better be there) will be bundled into an expression. At this time, we don't know how far away the next close parens is nor do we know what kind of expression will follow (number, binary, unary, or compound), so let's just push a marker onto our operations stack to help us remember where the expression started when we do get to the close parenthesis:

1. open parenthesis
   (a) if `last_was_expression` is true, throw an error.
   (b) (*) push "open" onto `operations`
   (c) `next_is_unary` is true
   (d) `last_was_expression` is false
   (e) `last_was_parenthesis` is true
2. close parenthesis
   (a) if `last_was_parenthesis` is true, throw an error.
   (b) `next_is_unary` is false
   (c) `last_was_expression` is true
   (d) `last_was_parenthesis` is false
3. operators
   (a) if `next_is_unary` is true then we know that `token` is a unary operator, otherwise it is binary:
      i. true: unary operator
      ii. false: binary operator
   (b) `next_is_unary` is true
   (c) `last_was_expression` is false
   (d) `last_was_parenthesis` is false
4. numbers
   (a) if `last_was_expression` is true, throw an error.
   (b) push number in `token` onto `expressions`
   (c) `next_is_unary` is false
   (d) `last_was_expression` is true
   (e) `last_was_parenthesis` is false

Let's consider now how we would handle unary expressions. We already know when we should expect to see one: It will be when `next_is_unary` is true and our token is an operator. What should we do with it? The first thing to remember is that a unary operator (at least the only one we have to deal with) has only one operand (aka "child" expression) which follows it to the right. If the operator is a "-" we will push a unary expression of the form "(- ?)" onto our expressions stack. The question mark is a place holder that we will fill in down the road once we've fully parsed the operand. We'll also push the operation onto our operations stack. Because unary "-" is exactly the same character as binary "-" we'll also push a "unary" flag onto the stack to emphasize that the operation is "unary". Check the priority of "unary" in the priority table. It is higher than everything other than exponentiation. Note that if the operator is anything else, there is an error in the input and we can stop. Here are our changes:

1. open parenthesis
   (a) if `last_was_expression` is true, throw an error.
   (b) push "open" onto `operations`
   (c) `next_is_unary` is true
   (d) `last_was_expression` is false
   (e) `last_was_parenthesis` is true
2. close parenthesis
   (a) if `last_was_parenthesis` is true, throw an error.
   (b) `next_is_unary` is false
   (c) `last_was_expression` is true
   (d) `last_was_parenthesis` is false
3. operators
   (a) if `next_is_unary` is true then we know that `token` is a unary operator, otherwise it is binary:
      i. true: unary operator
         A. (*) if `token` is "-" then
            - (*) push "- ?" onto expressions
            - (*) push "-" onto operations
            - (*) push "unary" onto operations
         B. (*) otherwise throw an error
      ii. false: binary operator
   (b) `next_is_unary` is true
   (c) `last_was_expression` is false

   (d) `last_was_parenthesis` is false
4. numbers
   (a) if `last_was_expression` is true, throw an error.
   (b) push number in `token` onto `expressions`
   (c) `next_is_unary` is false
   (d) `last_was_expression` is true
   (e) `last_was_parenthesis` is false

Now let's consider handling the binary operators. The interesting thing with binary operators is determining how the bind to their operands because of the order of operations. Exponentials are the real kicker. They have higher priority than any other operator, and furthermore they bind right-to-left, rather than left to right. This means that an expression like "`-2^3^4`" needs to bind as "`(-(2^(3^4)))`" and "`3-2^3^4x6`" is "`(3-((2^(3^4))x6))`". Look back at the priority table. This is why "^" has higher priority than "unary" and also why we have the "exponent" operator with lower priority than "^".

When we get the binary operator, we first build our new binary expression with the form "? token ?" where like above, the question mark is a place holder for the real expression. Before we push this new expression onto the expressions stack, we've got to check the priority of this operator against the one on the top of the operations stack with an expression like: `priority(operations.top)` $\geq$ `priority(token)`. If the operator on the stack is higher priority, we need to handle that one first. Note that the operator could be unary or binary so we'll need to check if the top operator is "unary" to decide if we are going to assemble a complete unary expression or a binary expression. We'll delay the discussion for how to actually assemble the expressions for a moment and, for now, just delegate it to functions.

Finally note that there may be several higher priority operators on the stack so we need to do the above check in a loop instead of in a single if. This would happen with an expression such as "`2^3^4`$\times$ `5`" when processing the "$\times$" token.

1. open parenthesis
   (a) if `last_was_expression` is true, throw an error.
   (b) push "open" onto `operations`
   (c) `next_is_unary` is true
   (d) `last_was_expression` is false
   (e) `last_was_parenthesis` is true
2. close parenthesis
   (a) if `last_was_parenthesis` is true, throw an error.
   (b) `next_is_unary` is false
   (c) `last_was_expression` is true
   (d) `last_was_parenthesis` is false
3. operators
   (a) if `next_is_unary` is true then we know that `token` is a unary operator, otherwise it is binary:
      i. true: unary operator
         A. if `token` is "-" then
            - push "- ?" onto `expressions`
            - push "-" onto `operations`
            - push "unary" onto `operations`
         B. otherwise throw an error
      ii. false: binary operator
         A. (*) while priority(`operations.top`) $\geq$ priority(`token`)
            - (*) if `operations.top` is "unary" then:
              – (*) assemble_unary_expression
            - (*) otherwise:
              – (*) assemble_binary_expression
         B. (*) push "? `token` ?" onto `expressions`
         C. (*) if `token` is "^" then
            - (*) push "exponent" onto `operations`
         D. (*) otherwise:
            - (*) push `token` onto `operations`
   (b) `next_is_unary` is true
   (c) `last_was_expression` is false
   (d) `last_was_parenthesis` is false

4. numbers
   (a) if `last_was_expression` is true, throw an error.
   (b) push number in `token` onto `expressions`
   (c) `next_is_unary` is false
   (d) `last_was_expression` is true
   (e) `last_was_parenthesis` is false

Well, now that we've gotten to the point where we've got expressions on our `expressions` stack ready to be assembled, how should we do it?

Let's talk about `assemble_unary_expression` first. There are several things we need. First is to pop a couple operations off of the `operations` stack. At this point there will be a "unary" operator on top followed by a "-" operator. Next we need to take our operand off the top of the `expressions` stack. Then we need to take our placeholder expression "- ?" off the stack. Next, we need to replace the question mark with our operand. Finally, we need to push our completed placeholder expression back on the stack.

There are a couple of things that can go wrong here. First, there might be less than two expressions left on the stack and secondly, the expression we popped with the place holder might not actually be a unary expression! In either case, it means that the user entered an expression with a missing operand. Can you figure out why?

Here is the pseudo-code for our `assemble_unary_expression` routine:

```
1  pop operations twice
2  pop expressions into operand
3  pop expressions into placeholder
4  replace the question mark in placeholder with operand
```

Note that the safety checks to identify missing operands are missing from the above.

Next we need to do `assemble_binary_expression`. It is very similar to our code for unary expressions except it doesn't work in a loop since there isn't enough information to determine if we can safely assemble the next expression. Here is the pseudo code:

```
1  pop operations
2  pop expressions into right operand
3  pop expressions into placeholder
4  pop expressions into left operand
5  replace the right question mark in placeholder with the right operand
6  replace the left  question mark in placeholder with the left  operand
```

As before, the safety checks for missing operands are not specified.

The only thing left that we need to do is finish handling our close parenthesis tokens. When we we reach a closed parenthesis, we know that we can assemble any pending operations on the stack until we reach out "open" operator that we pushed on the stack when we got an the matching open parens. We need to remember to pop our "open" operator too!

What's going to happen if we have an expression with an extra ")"? When we reach it we'll spin back and digest all of the expressions that come before and consume our sentinel "open" that came from the open and close parenthesis tokens we wrapped the user's expression with. At this point, our operations stack will be empty. The only time that is good is when we are at the end, in other words if `next_token` is end.

Let's see the results:

1. open parenthesis
   (a) if `last_was_expression` is true, throw an error.
   (b) push "open" onto `operations`
   (c) `next_is_unary` is true
   (d) `last_was_expression` is false
   (e) `last_was_parenthesis` is true
2. close parenthesis
   (a) if `last_was_parenthesis` is true, throw an error.
      i. (*) while `operations.top` $\neq$ "open"
         - (*) if `operations.top` is "unary" then:
           – (*) assemble_unary_expression

- (*) otherwise:
  - (*) assemble_binary_expression

ii. (*) pop `operations`

iii. (*) if `operations` is empty and `next_token` is not at the end
- (*) throw an unbalanced parenthesis error

(b) `next_is_unary` is false

(c) `last_was_expression` is true

(d) `last_was_parenthesis` is false

3. operators

(a) if `next_is_unary` is true then we know that `token` is a unary operator, otherwise it is binary:

i. true: unary operator

A. if `token` is "-" then
- push "- ?" onto `expressions`
- push "-" onto `operations`
- push "unary" onto `operations`

B. otherwise throw an error

ii. false: binary operator

A. while priority(`operations.top`) $\geq$ priority(`token`)
- if `operations.top` is "unary" then:
  - assemble_unary_expression
- otherwise:
  - assemble_binary_expression

B. push "? `token` ?" onto `expressions`

C. if `token` is "^" then
- push "exponent" onto `operations`

D. otherwise:
- push `token` onto `operations`

(b) `next_is_unary` is true

(c) `last_was_expression` is false

(d) `last_was_parenthesis` is false

4. numbers

(a) if `last_was_expression` is true, throw an error.

(b) push number in `token` onto `expressions`

(c) `next_is_unary` is false

(d) `last_was_expression` is true

(e) `last_was_parenthesis` is false

Now finally we have handled all the cases! If somehow we get to the end of our tokens and we still have operations on the `operations` stack or not exactly one expression on the `expressions` stack, we know the user entered a bad expression say with extra open parenthesis. Assuming that a good string was entered, you now have the final expression sitting as the only item in the `expressions` stack.

Now let's consider our state. We have three boolean variables which should give us eight different possible states. But there are only four different transitions which means that only a maximum of four of those states are reachable. If we map out the transitions, we quickly see that `last_was_expression` is always inverted from `next_is_unary` so we can replace all references to it with "`next_is_unary` is false" in our algorithm. Further more we can notice that `last_was_parenthesis` is false whenever `next_is_unary` is false. The difference is that `last_was_parenthesis` is false after receiving a operation token and `next_is_unary` is true in that case. Well when is `last_was_parenthesis` used? Only when handling a closed parenthesis. What would happen if we got a closed parenthesis after getting an operator? It means we have a missing operand which is the same thing we were catching before. This means we can replace `last_was_parenthesis` with `next_is_unary`. Now we are down to only one state variable! By the way, if your implementation is carefully coded, you shouldn't need the safety checks in your `assemble` routines.

Remember, we're going to loop through our tokens from left-to-right. Each token will be handled according as either a operator, parenthesis (open or closed), or as a number. When we finish looping, the `operations` stack should be empty and the `expressions` stack should have exactly one item, otherwise, we had an unbalanced parenthesis.

Here are the final algorithms:

`handle_token`:

1. open parenthesis
   (a) (*) if `next_is_unary` is false, throw an error.
   (b) push "open" onto `operations`
   (c) `next_is_unary` is true
2. close parenthesis
   (a) (*) if `next_is_unary` is true, throw an error.
      i. while `operations.top` $\neq$ "open"
         - if `operations.top` is "unary" then:
           – assemble_unary_expression
         - otherwise:
           – assemble_binary_expression
      ii. pop `operations`
      iii. if `operations` is empty and `next_token` is not at the end
         - throw an unbalanced parenthesis error
   (b) `next_is_unary` is false
3. operators
   (a) if `next_is_unary` is true then we know that `token` is a unary operator, otherwise it is binary:
      i. true: unary operator
         A. if `token` is "-" then
            - push "- ?" onto `expressions`
            - push "-" onto `operations`
            - push "unary" onto `operations`
         B. otherwise throw an error
      ii. false: binary operator
         A. while priority(`operations.top`) $\geq$ priority(`token`)
            - if `operations.top` is "unary" then:
              – assemble_unary_expression
            - otherwise:
              – assemble_binary_expression
         B. push "? `token` ?" onto `expressions`
         C. if `token` is "^" then
            - push "exponent" onto `operations`
         D. otherwise:
            - push `token` onto `operations`
   (b) `next_is_unary` is true
4. numbers
   (a) (*) if `next_is_unary` is false, throw an error.
   (b) push number in `token` onto `expressions`
   (c) `next_is_unary` is false

`assemble_unary_expression`:

```
pop operations twice
pop expressions into operand
pop expressions into placeholder
replace the question mark in placeholder with operand
```

`assemble_binary_expression`:

```
pop operations
pop expressions into right operand
pop expressions into placeholder
pop expressions into left operand
replace the right question mark in placeholder with the right operand
replace the left  question mark in placeholder with the left  operand
```