Precedence correctness of binary expression trees, based on binding powers

joekreu

September 2021

This document is about parsing expressions that match the following simple scheme:

An alternating sequences of atomic operands and infix operators, where the operators can have arbitrary and independent left and right binding powers.

The notion of *precedence correctness*, which is independent of a specific parsing algorithm, is defined and examined.

Precedence correctness, as the term is used here, is a modification of this term from *Annika Aasa's* text *Precedences in specifications and implementations of programming languages* (1995).

Furthermore, some considerations for suitable parsing algorithms are included.

1. General assumptions

Expressions consisting of atoms and infix operators in the following form are studied:

```
(**)          A0 Op1 A1 Op2 ... Opn An
```

Here, `A0`, `A1`, ... are *atoms* (e.g., numbers or identifiers) and `Op1`, `Op2`, ... are infix operators.

Every infix operator `Opk` is supposed to have two numbers, a *left* and a *right binding power*, denoted by *lbp*`(Opk)`, *rbp*`(Opk)`, resp. Typically, binding powers are positive integers. Greater numbers mean stronger binding (higher precedence). In the definitions and algorithms to be considered here, binding powers are only *compared* (and minima are formed); there are no arithmetic operations on binding powers. Often the *lbp* and *rbp* values of a specific operator are equal or differ by one. However, *lbp* and *rbp* values of an operator may differ by *any number*. This makes the concept of binding powers more powerful, but also more complex, than the frequently used scheme based on *precedence* (an integer) and *associativity* (two possible values: *left* and *right*).

In simple terms, if the token sequence `... OpA P OpB ...` is part of an expression with operators `OpA` and `OpB` and operand `P`, then `OpA` will 'win' `P` as right operand if its *rbp* is greater than or equal to the *lbp* of `OpB`; otherwise `OpB` will 'win' `P` as left operand. However, what an operator 'wins' is not necessarily a single operand. It may be a more tightly connected part of the expression. In `x + u ^ v * y`, the operator `^` might have the greatest binding powers, therefore `+` and `*` 'compete' for the subexpression `u ^ v`, and usually `*` will 'win'.

> *Note:* The scheme `(**)` is the basis for parsing expressions with operators having arbitrary binding powers. Once it is clear how to parse (**), the transition to more complicated expressions containing unary operators, mixfix operators, function calls and other constructs is fairly straightforward. This should be studied elsewhere. Suffice it to say here that *unary* operators (*prefix* or *postfix*) can be treated as infix operators by inserting fake operands (to the left of a prefix and to the right of a postfix operator). The binding power towards the fake operand is set 'very high', i.e., higher than any 'normal' binding power.

A *parse tree for* `(**)` is an ordered binary tree with a root node (in the sense of graph theory) such that every node has zero or exactly two children, and the original left-to-right order is preserved. An operator node is an inner node; it has two children. A node without children is an *atom*; considered as node in the tree it is a *leaf*.

In this text, the term *parse tree for a specific expression* `(**)` always means: *A binary tree that preserves the original order*. This term by itself, without additional specification, does not consider binding powers. See example in section 2 and section 3.1.

Parse trees can be represented in several ways: By insertion of parentheses in `(**)` without changing the order, by S-expressions (i.e., fully parenthesized, operator first), or by two-dimensional diagrams. See the following example.

2. An example

The expression

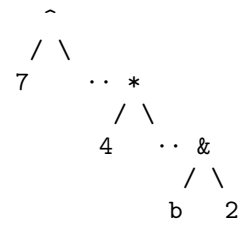
```
(1)    7 ^ 4 * b & 2
```

matches the scheme `(**)`. Here, `7`, `4`, `b`, `2` are atoms and `^`, `*`, `&` are infix operators. There are five parse trees for `(1)`. In the following, each of these trees is represented in three ways: To the left, first by simple parentheses insertion and below that by an *S-expression* (known from the *Lisp* language), to the right by a diagram.

Parse tree 1

7 ^ (4 * (b & 2))

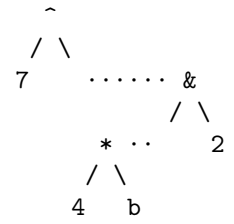
(^ 7 (* 4 (& b 2)))



Parse tree 2

7 ^ ((4 * b) & 2)

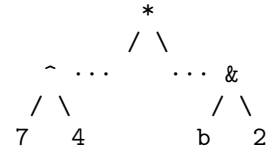
(^ 7 (& (* 4 b) 2))



Parse tree 3

(7 ^ 4) * (b & 2)

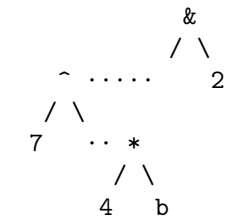
(* (^ 7 4) (& b 2))



Parse tree 4

(7 ^ (4 * b)) & 2

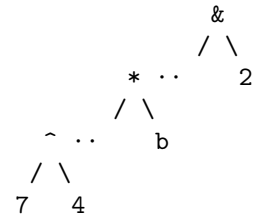
(& (^ 7 (* 4 b)) 2)



Parse tree 5

((7 ^ 4) * b) & 2

(& (* (^ 7 4) b) 2)



Note that the horizontal order of atoms is always the same: `7, 4, b, 2`.

In the simply parenthesized and the diagram representation, the original horizontal order of *all* tokens is preserved. Therefore, the original sequence `(**)` can easily be recovered from these representations.

Now binding powers come into play. Here is a set of binding powers for `(1)`:

```
       lbp(^) = 21,   lbp(*) = 14,   lbp(&) = 17
(2)
       rbp(^) = 20,   rbp(*) = 15,   rbp(&) = 18
```

The parse tree that takes into account the binding powers `(2)` is the third one: `(7 ^ 4) * (b & 2)`. This can easily be seen, because `*` has the smallest binding powers of the three operators, to both sides (left and right). So there is no reasonable alternative for a *precedence correct* parse result. The general situation is more complex.

3. More on parse trees and binding powers

3.1 Number of parse trees

In `(**)`, the case of `A0` (only one atom) is not excluded. There is only one possible parse tree, which just consists of one node.

Likewise, for the case `A0 Op1 A1` (one operator and two atoms), there is only one possible parse tree; as S-expression this is written as `(Op1 A0 A1)`.

For a sequence `(**)` with n operators there are C_n possible parse trees, where C_n is the n-th *Catalan number*. See https://en.wikipedia.org/wiki/Catalan_number. The first few Catalan numbers are

$C_0 = 1$; $C_1 = 1$; $C_2 = 2$; $C_3 = 5$; $C_4 = 14$; $C_5 = 42$; $C_6 = 132$

For the expression `(1)`, we have n = 3; so there are $C_3 = 5$ parse trees.

3.2 Subtrees

The root node of a parse tree is also called the *root operator*; it is the first operator in the S-expression representation; and the highest operator in the diagram. In many typical situations it is the operator with the smallest binding powers. However, *lbp* and *rbp* of an operator can be 'very' different. Moreover, more than one operator with the same '*smallest binding power*' can exist. Therefore, the general answer to the *root operator* question is more difficult.

Any operator in a parse tree may be considered as root node of a *subtree*, which by itself has the structure of a parse tree, and is indeed the parse tree of a corresponding subexpression.

A parse tree can be defined recursively:

- An atom is a parse tree
- If `LST` and `RST` are parse trees and `Op` is an operator, then there is a parse tree with `Op` as root operator, `LST` as left operand and `RST` as right operand.

This can be represented like this: A non-atomic parse tree is

```
                    Op
(3)               /    \
                LST      RST
```

where `LST` and `RST` are parse trees (atomic or not). They are called the left, resp. right *subtree* of the parse tree `(3)`.

Likewise, the recursive structure can be used to define properties of a parse tree:

- Define the property for atoms
- For a given tree or subtree of the form `(3)`, define the property on the basis of binding powers of `Op`, the property for `LST` and for `RST`, and possibly other conditions.

3.3 A definition of *precedence correctness*, based on binding powers

The following definitions are modelled after *Annika Aasa's* paper *Precedences in specifications and implementations of programming languages* (1995). See https://www.sciencedirect.com/science/article/pii/030439759590680J

A definition of *precedence correctness* should obviously imply that, given a parse tree `(3)` is precedence correct, `LST` and `RST` are also precedence correct. This condition is not sufficient for the precedence correctness of the whole tree (3).

To complete the definition of *precedence correctness*, first, two preparatory definitions are given ('`LST Op RS`' is short for the form `(3)`):

- The *left weight* (`LW`) of a parse tree is recursively defined:
  - `LW(A)` = ∞
  - `LW(LST Op RST) = min(LW(LST, lbp(Op))`
- The *right weight* (`RW`) is defined accordingly
  - `RW(A)` = ∞
  - `RW(LST Op RST) = min(RW(RST, rbp(Op))`

  In words: The left and the right weight of an atom is *infinity*. The left weight of a non-atomic tree (or subtree) of the form `(3)` is the minimum of the *lbp* of the operator `Op` and the left weight of the left subtree. The right weight of a non-atomic tree (or subtree) of the form `(3)` is the minimum of the *rbp* of the operator `Op` and right weight of the right subtree.

Now, the definition of *precedence correctness* is as follows:

- A tree consisting of only an atom is precedence correct.

- A tree of the form `(3)` is precedence correct if the following conditions are satisfied:
  - both subtrees `LST` and `RST` are precedence correct.
  - `RW(LSP) >= lbp(Op)`
  - `LW(RSP) > rbp(Op)`

  In words: An atom is precedence correct. A non-atomic tree is precedence correct if both subtrees are precedence correct, the right weight of the left subtree is greater than or equal to the *lbp* of the operator `Op`, and the left weight of the right subtree is greater than the *rbp* of the operator.

Caution: One might guess that the following property (called *weak precedence correctness* here) is equivalent to precedence correctness:

- A tree consisting of only an atom is *weakly precedence correct*.
- A tree of the form `(3)` is *weakly precedence correct* if the following conditions are satisfied:
  - both subtrees `LST` and `RST` are weakly precedence correct.
  - `LST` is atomic, or `rbp(OPL) >= lbp(Op)`
  - `RST` is atomic, or `lbp(OPR) > rbp(Op)`

where `OPL` is the root operator of `LSP` und `OPR` is the root operator of `RSP`.

By definition of `LW` and `RW`, we always have `RW(LST) <= rbp(OPL)` and `LW(RST) <= lbp(OPR)`. Therefore, a precedence correct tree is weakly precedence correct. The following example shows, that the reverse statement is not valid.

```
(4)    A0 4_3 A1 2_5 A2 4_5 A3
```

Here, `A0`, `A1`, `A2`, `A3` are atoms; `4_3`, `2_5`, `4_5` are operators. To make tracing easier, the operator symbols contain the binding powers, i.e., *lbp*(`4_3`) = 4, *rbp*(`4_3`) = 3, and so on.

The following parse tree for `(4)`

```
                          4_5
                         /   \
                   2_5···     A3
(5)               /   \
             4_3···     A2
            /   \
          A0     A1
```

is precedence correct (and weakly precedence correct). However, the tree

```
             4_3
            /   \
          A0     ··········4_5
(6)                         /   \
                      2_5···     A3
                     /   \
                   A1     A2
```

for `(4)` is weakly precedence correct, but not precedence correct. The subtrees of the root operator in `(6)` are even precedence correct (not only weakly).

Further, the example shows that the difference is not related to the asymmetry of the definition (see second note below). In `(4)`, all *lbp* values are even numbers, and all *rbp* values are odd numbers. Because minima are formed only of binding powers of one kind (*left* or *right*), and comparisons are done only between different kinds of binding powers, equal numbers are never compared for (4). So the results for this example will not change if `>` is replaced by `=>` or vice versa.

Further notes:

- *Infinity* (i.e., ∞, as used in the definition of left and right weight) compares greater than any number. A number that is greater than any binding power could be used instead (such as `1000`, provided all binding powers are less than `1000`). Since binding powers are only used for comparison and taking minima (not for arithmetics), the use of ∞ seems appropriate here. Besides, ∞ better expresses the intention of the definition.
- The slight asymmetry in the definition of precedence correctness (`>=` vs. `>`) is to ensure that there is exactly one precedence correct parse result, even if there are infix operators with equal *lbp* and *rbp*. If *lbp*(`+`) = *rbp*(`+`), then `(a + b) + c` will be the precedence correct parsing of the expression `a + b + c`; `a + (b + c)` will not be correct. With `>` at both places, both results would be incorrect; with `>=` at both places, both results would be correct. This provides some evidence for the usefulness of the definition of *precedence correctness*. It still remains to show the existence of exactly one precedence correct parse tree for every valid expression and valid definition of operators.
- *Differences to Aasa's definitions of left and right weight, and precedence correctness*: *Aasa* uses precedence (one number) and associativity (two possible values: *left* or *right*). Precedence values have reverse meaning: smaller numbers mean tighter binding in *Aasa*'s text. Therefore, in her definitions, atoms have zero left and zero right weight; maxima are taken instead of minima; comparisons are reversed. Unary operators (prefix and postfix) are considered explicitly by *Aasa*. Here, unary operators are implicitly included (see note on page 2).

The ultimate goals are

- to show that for every expression `(**)` and arbitrary valid binding powers, there is exactly one precedence correct parse tree.
- to show that the parsers in this repository always create precedence correct parse trees from valid input.
- to show that *precedence correctness*, as used here, is equivalent to *Aasa*'s use in the cases that can be described by *precedence* and *associativity*.

Another goal could be the construction of an equivalent unambiguous *context free grammar* for a system of infix operators with arbitrary left and right binding

powers. This grammar should parse expressions to precedence correct parse trees.

3.4 Ranges of operators. Top operators

To prove the elementary properties of *precedence correctness* (see '*goals*' at the end of the preceding subsection), some further considerations are needed.

The following definitions of *left range*, *right range* and *range* of an operator refer to an operator within an original expression `(**)`. *Ranges* are subsequences within the sequence `(**)`. They consider the binding powers of the operator and its neighboring operators. The definition does not refer to a parse tree.

The left range of an operator starts immediately to the left of the operator and extends further to the left. The right range of an operators starts immediately to the right of the operator and extends further to the right. The *range* of an operator is the union of left range, the operator itself and right range.

The ranges are meant to extend to the end of the operator's effect within the sequence `(**)`. They are expected to form the left vs. right subtree of the operator in a precedence correct parse tree. The formal definitions:
