How to evaluate integer expressions (part i : the simple case)

0. We consider the datatype T of (finite) binary trees, as defined recursively by:

$$\langle\rangle \quad \in T$$
$$\langle s,t\rangle \quad \in T \text{ , for all } s,t \text{ in } T \text{ .}$$

Such trees can be represented by lists in various ways, one of which is given by the function L, of type $T \to \mathcal{L}(\{0,1\})$, defined as follows:

$$L.\langle\rangle = [0]$$
$$L.\langle s,t\rangle = [1] +\!\!+ L.s +\!\!+ L.t$$

Function L maps a tree onto a list of 0's and 1's; what remains to be shown is that L maps different trees to different lists: otherwise the same list could represent different trees, which is not what we consider a proper representation. That is to say, L must have a left-inverse, which means that a function P must exist, of type $\mathcal{L}(\{0,1\}) \to T$, with the property that:

$$(\forall s : s \in T : P.(L.s) = s) \text{ .}$$

Before showing this, however, we first transform the definition of L into a more manageable form. We introduce a binary operator $\oplus$, of type $\mathcal{L}(\{0,1\}) \times T \to \mathcal{L}(\{0,1\})$ with the following definition:

$$(0) \qquad x \oplus s = x +\!\!+ L.s \text{ .}$$

Then we have:

$$L\cdot s \;=\; [] \oplus s \;,$$

so L can be defined in terms of $\oplus$. By using L's definition we can derive the following definition for $\oplus$, which implies (0):

$$\begin{aligned} x \oplus \langle\rangle &= x +\!\!+ [0] \\ x \oplus \langle s,t\rangle &= ((x +\!\!+ [1]) \oplus s) \oplus t \;. \end{aligned}$$

This definition is an instance of the pattern we discussed in rh208 (section 3); by application of the transformation associated with this pattern we obtain:

$$L\cdot s \;=\; [] \otimes [s] \;, \text{ where}$$

$$\begin{aligned} &(1a) && x \otimes [] &&= x \\ &(1b) && x \otimes ([\langle\rangle] +\!\!+ ss) &&= (x +\!\!+ [0]) \otimes ss \\ &(1c) && x \otimes ([\langle s,t\rangle] +\!\!+ ss) &&= (x +\!\!+ [1]) \otimes ([s,t] +\!\!+ ss) \;. \end{aligned}$$

Rules (1b) and (1c) have been derived from the following required property of $\otimes$ (which is just $\otimes$'s specification as the folded version of $\oplus$, together with (0)):

$$(1d) \quad x \otimes ([s] +\!\!+ ss) \;=\; (x +\!\!+ L\cdot s) \otimes ss \;.$$

That is, the validity of (1d) follows from (1b) and (1c) (and the properties of L) only and, therefore, (1d) holds for any binary operator $\otimes$ that satisfies (1b) and (1c).

This is nice because equations (1b) and (1c) can also be read from right to left, in which case the operator is defined recursively over x instead over ss. Calling the operator thus introduced $\odot$ we obtain:

$$(2a) \quad (x +\!\!+ [0]) \odot ss = x \odot ([\langle\rangle] +\!\!+ ss)$$
$$(2b) \quad (x +\!\!+ [1]) \odot ([s,t] +\!\!+ ss) = x \odot ([\langle s,t\rangle] +\!\!+ ss) .$$

(Notice that this admits of the operational interpretation of parsing list x.)
As stated above, we obtain for free that $\odot$ satisfies:

$$(2c) \quad (x +\!\!+ L{\cdot}s) \odot ss = x \odot ([s] +\!\!+ ss)$$

as well, a special instance of which is:

$$(3a) \quad L{\cdot}s \odot [] = [] \odot [s] ,$$

which is almost a solution to the problem of deriving a left-inverse of L: we are done if we can solve s from $[] \odot [s]$. Fortunately, (2a) and (2b) define $x \odot ss$ for nonempty x only, so we may still choose the value of $[] \odot ss$. By choosing:

$$(2d) \quad [] \odot ss = ss ,$$

formula (3a) becomes:

$$(3b) \quad L{\cdot}s \odot [] = [s] ,$$

which does the job. Removal of a pair $[\cdot]$ is trivial because $s = [s]{\cdot}0$, so our function P can be

defined by — repeating the definition of $\odot$ — :

$$P.x \;=\; (x \odot [\,]).0 \quad , \text{ where}$$

$$\begin{array}{lcl} [\,] \odot ss & = & ss \\ (x +\!\!+ [0]) \odot ss & = & x \odot ([\langle\rangle] +\!\!+ ss) \\ (x +\!\!+ [1]) \odot ([s,t] +\!\!+ ss) & = & x \odot ([\langle s,t\rangle] +\!\!+ ss) \end{array}$$

The above is an example, probably the simplest possible, of parser design by program inversion.

For any tree s the list $L.s$ is a representation of s in <u>prefix code</u>: the 0's and 1's are prefixed to the representations of the subtrees. The above definition of $\odot$ gives rise to a backward parsing of the list, that is, the list is parsed as <u>postfix code</u>. For example:

$$\begin{array}{ll} & P.(L.\langle s,t\rangle) \\ = & \quad \{ \text{ definition of } P \text{ and } L \} \\ & ((([1] +\!\!+ L.s +\!\!+ L.t) \odot [\,]).0 \\ = & \quad \{ \text{ (2c) } \} \\ & (([1] +\!\!+ L.s) \odot [t]).0 \\ = & \quad \{ \text{ (2c) } \} \\ & ([1] \odot [s,t]).0 \\ = & \quad \{ \text{ definition of } \odot \} \\ & ([\,] \odot [\langle s,t\rangle]).0 \\ = & \quad \{ \text{ definition of } \odot \} \\ & [\langle s,t\rangle].0 \\ = & \quad \{ \text{ definition of } [\cdot] \} \\ & \langle s,t\rangle \,. \end{array}$$

This example illustrates that lists representing subtrees are

parsed before the structure of the whole tree is known: the tree is constructed in a bottom-up fashion.

1. We now consider binary trees whose leaves and nodes are labelled with values of an as yet unspecified type. For the time being dummy b denotes such values. The datatype T is now defined recursively as follows:

$$\langle b \rangle \quad \in T \text{ , for all } b$$
$$\langle b,s,t \rangle \in T \text{ , for all } b \text{ and } s,t \in T .$$

We redefine function L as follows:

$$L\cdot\langle b \rangle = [\,\langle 0,b \rangle\,]$$
$$L\cdot\langle b,s,t \rangle = [\,\langle 1,b \rangle\,] +\!\!+ L\cdot s +\!\!+ L\cdot t$$

So, the elements of the list L·s are now pairs consisting of a 0 or a 1 and a label value.

This is a generalisation of the previous case; yet, we need not redo our work: in a way, the situation is still the same. All we must do to obtain a parser is to take into account that 0 and 1 have been replaced by $\langle 0,b \rangle$ and $\langle 1,b \rangle$ respectively, thus:

$$P\cdot x = (x \odot []) \cdot 0 \text{ , where}$$

$$[] \odot ss = ss$$
$$(x +\!\!+ [\langle 0,b \rangle]) \odot ss = x \odot ([\langle b \rangle] +\!\!+ ss)$$
$$(x +\!\!+ [\langle 1,b \rangle]) \odot ([s,t] +\!\!+ ss) = x \odot ([\langle b,s,t \rangle] +\!\!+ ss)$$

Because the only purpose of the 0's and 1's in the list representation of trees is to distinguish two cases —namely $\langle b \rangle$ and $\langle b,s,t \rangle$— , the 0's and 1's can be eliminated as soon as the labels of leafs can be distinguished from labels of nodes. This gives rise to the following variation, in which b denotes leaf labels and c denotes node labels, so we assume:

$$(\forall b,c :: b \neq c) \ .$$

The datatype T is now redefined as follows:

$$\begin{aligned} \langle b \rangle &\in T \ , \text{ for all } b \\ \langle c,s,t \rangle &\in T \ , \text{ for all } c \text{ and } s,t \in T \ . \end{aligned}$$

Then we obtain:

$$\begin{aligned} L.\langle b \rangle &= [b] \\ L.\langle c,s,t \rangle &= [c] +\!\!+ L.s +\!\!+ L.t \ , \end{aligned}$$

and:

$$P.x = (x \odot []).0 \ , \text{ where}$$

$$\begin{aligned} [] \odot ss &= ss \\ (x +\!\!+ [b]) \odot ss &= x \odot ([\langle b \rangle] +\!\!+ ss) \\ (x +\!\!+ [c]) \odot ([s,t] +\!\!+ ss) &= x \odot ([\langle c,s,t \rangle] +\!\!+ ss) \end{aligned}$$

If so desired, the 0's and 1's may be thought of as having been incorporated into the b's and c's. As a result, this definition is simpler than the previous one: all that matters is that $b \neq c$.

2. We now assume that b in ⟨b⟩ has type Int and c in ⟨c,s,t⟩ has type Int × Int → Int, that is, the trees are labelled with integer values in their leaves and with binary integer operators (like +, −, *, ⋯) in their nodes. Trees now represent integer values; the value of tree s is V·s where function V, of type T → Int, is defined by:

$$\begin{aligned} V\cdot\langle b\rangle &= b \\ V\cdot\langle c,s,t\rangle &= V\cdot s \ (c) \ V\cdot t \ , \end{aligned}$$

where we use (c) as infix notation for the operator c.

Function P from the previous section is a parser: it reconstruc a tree from its list representation. The function V∘P is an evaluator: it computes the value of a tree from its list representation. The definition of V is compositional: the value of a composite tree depends on the values of its constituents only (, not on other properties of its constituents). The definition of P is such that trees are constructed in bottom-up fashion. Because of these two properties, the definitions of P and V can be merged into a new definition for V∘P without much effort. (This transformation is nowadays called fusion.) Calling the new variation of the operator ⊙ from P's definition now $\bar{\odot}$ we obtain:

$$(V\circ P)\cdot x = (x \ \bar{\odot} \ [])\cdot 0 \ , \text{ where}$$

$$\begin{aligned} [] \ \bar{\odot} \ bs &= bs \\ (x +\!\!+ [b]) \ \bar{\odot} \ bs &= x \ \bar{\odot} \ ([b] +\!\!+ bs) \\ (x +\!\!+ [c]) \ \bar{\odot} \ ([b_0, b_1] +\!\!+ bs) &= x \ \bar{\odot} \ ([b_0 \ (c) \ b_1] +\!\!+ bs) \ . \end{aligned}$$

Aside: Whereas $\odot$ satisfies $(x +\!\!+ L.s) \odot ss = x \odot ([s] +\!\!+ ss)$
we now have $(x +\!\!+ L.s) \bar{\odot} bs = x \bar{\odot} ([V.s] +\!\!+ bs)$.

□

Operator $\bar{\odot}$ can be considered as an interpreter for integer expressions in postfix representation. In $x \bar{\odot} bs$ list x represents (the remainder of) the expression (still) to be evaluated and bs represents the "evaluation stack". List x can be considered as a machine code program in which an element b amounts to the instruction to "push value b onto the stack" whereas an element c is the arithmetic instruction to "replace the topmost two stack elements by their result". In this way, the above (iterative) definition of $\bar{\odot}$ is nothing but an (abstract) specification of a processor. (see rh210 for elaborations)

Eindhoven, 20 september 1994

Rob R. Hoogerwoord