Ground-Up Computer Science

Chapter 5

(Dec 23, 2022)

Yin Wang

5 Calculator

A: How was the tree exercises?

B: They are just slightly different from the list exercises.

A: That is good. Their close relationship can help you understand the ideas.

B: In the trees of previous lesson, we can't have data members in the internal nodes. Can we have them in this lesson?

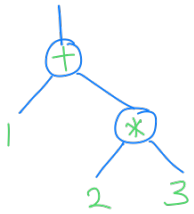
A: Yes, we will put some data members in the internal nodes. We will make something interesting out of this.

B: What is that?

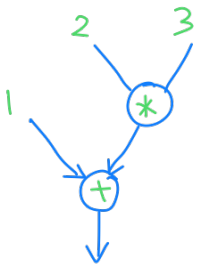
A: Actually, you have already seen trees with data members in the internal nodes before.

B: Really? I don't recall anything like that.

A: How about this one?



B: Oh, that looks like the *computation graph* we learned in the first lesson, except that it is upside-down.



A: It is the computation graph. Does the orientation matter at all?

B: No. We have drawn trees upside-down in the previous lesson. It doesn't matter in what direction we draw it. It is still the same thing.

A: This is a tree with two data members `"+"` and `"*"` in the internal nodes.

B: Are `"+"` and `"*"` strings?

A: Yes. The operators `"+"` and `"*"` are represented as strings here. It is not necessary to represent operators as strings, but for simplicity we just use strings here.

B: We just put operators in the internal nodes, and the tree becomes a computation graph. Baby steps can go a long way.

A: Yes. Can you see how we can construction this tree, the computation graph?

B: Previously we used pairs as internal nodes. If we use pairs, we have no place to store the data members, so I guess we need something larger.

A: Instead of simple pairs, we can use lists as internal nodes, then we can have data members in the internal nodes.

B: That makes sense.

A: The internal nodes can be as simple as `pair("*", pair(2, pair(3, null)))`.

B: Then the whole computation graph of `1 + 2 * 3` can be written as `pair("+", pair(1, pair(pair("*", pair(2, pair(3, null))), null)))`?

A: Right. But this is hard to read.

B: Yes, it will be even harder with complex expressions.

A: We can apply the idea of *abstract data type* again. We call this data type `binop`, which means "binary operation". A `binop` contains three members, an operator and two operands. The two operands can be `binop` themselves.

B: Why didn't we use abstract data type for the trees of last lesson?

A: That is a good question. Because our pairs happen to create trees, we didn't bother to create an abstract data type for those trees. But this time it is a bit different. We can no longer use pairs as internal nodes.

B: I see. Because `pair("+", pair(1, pair(pair("*", pair(2, pair(3, null))), null)))` is too complex and error-prone. We need to abstract those details out.

A: Right. Let's look at the abstract interfaces of `binop` one by one then. First, the constructor. Think about this, what should the constructor of `binop` create?

B: It should return a list containing three members, an operator and two operands. The operands can be `binop`'s themselves.

A: Correct. Our first attempt is something like this:

```
function binop(op, e1, e2)
{
  return pair(op, pair(e1, pair(e2, null))):
}
```

B: We just put the three members into a list.

A: But if we just write the constructor this way, then we would have trouble distinguishing `binop` from other kinds of lists.

B: Why do we need to distinguish them?

A: For example, when you write a recursive function on a computation graph, you will need to ask whether the input is an internal node (`binop`) or not.

B: How can we tell `binop` from other kinds of lists?

A: For this purpose, we can put a special string member `"binop"` into the list, like this.

```
function binop(op, e1, e2)
{
  return pair("binop", pair(op, pair(e1, pair(e2, null))));
}
```

Whenever the first member is the string `"binop"`, we think this list is a `binop`. This is like putting tags on things. We call this string a *type tag*.

B: What if there are people who happen to create a list with such a tag, but they don't mean the same thing?

A: This method is not for reliable prevention of accidents. It is just demonstrating a general idea how we distinguish data types. We just put some special tag in there, just like we put tags on products in a store.

B: That sounds like a good idea, inspired by everyday life.

A: Actually *type tags* are not necessarily strings. They can be any value as long as it is unlikely some other people come up with the same value.

B: I could think of a better type tag, such as `"binop$62A4#E91"`.

A: Good. Actually you may generate a random number which is very unlikely to be used by other people. That will be much more reliable. But for demo purposes we don't use complex schemes.

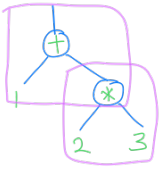
B: Okay. It is good to have ideas first, and refinements can come later.

A: So we have our constructor for `binop`. Can you come up with the *type predicate* and *visitors* yourself? You can call the type predicate `isBinOp` and the visitors `binopOp`, `binopE1` and `binopE2`.

B: *(Write your answers and send them to the teacher.)*

A: Well done. After having the `binop` node structure, we can use it to construct our previous examples for `2 * 3` and `1 + 2 * 3`. You may find how much clearer they are. If you are not sure, take a look at the following picture and see where are the `binop` structures.

B: *(Write your answers and send them to the teacher.)*



A: Good. Now we can think about what we can do with the computation graphs constructed with `binop`.

B: From the first lesson, we know that we can compute their values. That is, to *evaluate* them.

A: Evaluation of computation graphs is our final goal, but first let's do something simpler than that. We build some small utility functions which can make `binop` more convenient to use.

B: What are the utility functions?

A: First of all, computation graphs constructed by `binop` is still not easy to read. It is better if we can display them as usual math expressions such as `2 * 3` and `1 + 2 * 3`. We call these *infix notations* because the operator `*` and `+` are placed in the middle of the two operands.

B: That seems to be a nice thing to have.

A: We call this function `toInfix`. Using our abstract data type, you can proceed to Exercise Set 5 and work out `toInfix`. To avoid the problem of *operator precedence*, we put parentheses around all subexpressions, so we have `(2 * 3)` and `(1 + (2 * 3))`.

B: *(Write your answer to `toInfix` and send them to the teacher.)*

A: Now we have a function that can display the `binop` structure as an usual math expression. With this example, I hope you see the difference between *computation graphs* and *text expressions*.

B: We have talked about this in the first lesson. Now I can see that they are really different things. `binop("*", 2, 3)` constructs the *computation graph*, and `toInfix` function converts it into a *text expression* `"2 * 3"`.

A: The computation graph is *structural*, which has a clear structure. It is easy to extract its parts with visitor functions (`binopOp`, `binopE1` and `binopE2`). After it is transformed into text by `toInfix`, it is much harder to get the parts out in a meaningful way.

B: I can see that the string is just a series of characters with no clear division of structure. For the expression `(1 + (2 * 3))`, it is quite hard to extract the two operands `1` and `(2 * 3)` even with the parentheses. The computer only sees the sequence of characters `(`, `1`, `+`, `(`, `2`, `*`, `3`, `)`, and `)`.

A: We use `toInfix` only for displaying the computation graph to humans. Inside computer programs we almost never use the text format. So be sure not to use `toInfix` unless you need to display on the screen.

B: Got it.

A: There is a meaningful (but difficult) way to extract parts from text expressions. You can transform the string back into a graph by using a *parser*. A parser is a function which transforms a text expression into a computation graph.

B: So a parser is like the inverse function of `toInfix`? Can we write a parser?

A: Yes, the parser and `toInfix` are the mutual inverse functions. We don't write a parser now. Parsers are complex and difficult functions to write. It is better you finish all the exercises of this lesson before attempting to write a parser. I will give you a parser exercise if you do well in this lesson.

B: That is nice!

A: Next we write a function similar to `toInfix`, except that it produces *prefix notations*. If the infix notation is `(2 * 3)`, then the prefix notation is `(* 2 3)`. If the infix notation is `(1 + (2 * 3))`, then the prefix notation is `(+ 1 (* 2 3))`. You see what is going on?

B: Yes. The prefix notation just puts the operator before the two operands.

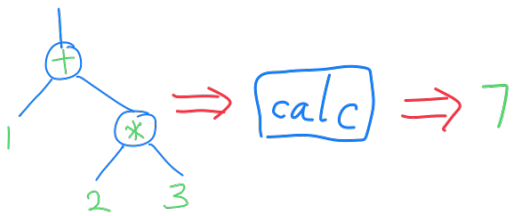
A: We call this function `toPrefix`. This is also in Exercise Set 5. You can do it now.

B: *(Write your answer to `toPrefix` and send them to the teacher.)*

A: Okay. Two warm-up exercises done, we can proceed to write the calculator.

B: Great!

A: The *calculator* (`calc`) is a function from computation graphs to values. It belongs to a general category of functions called *interpreters*. You can think of the calculator as an interpreter for a simple language with only arithmetic expressions, like `2 * 3` and `1 + 2 * 3`.



B: I have heard of *interpreters* before, such as the "JavaScript interpreter".

A: Interpreters are functions that execute your programs. A JavaScript interpreter can execute programs written in the JavaScript language. There are multiple implementations of JavaScript interpreter. For example, the JavaScript interpreter we use for this course is called *node.js*.

B: I see. So `calc` is also an interpreter, except that it runs programs written in a much smaller language, just arithmetic expressions.

A: Right. We will extend this small interpreter to something bigger and more interesting in the next few lessons.

B: Nice.

A: Now we start to write the `calc` function. It is quite easy. It is just a little different from a function you wrote in the last lesson.

B: Which function?

A: `treeSum`. The `calc` function is very much like `treeSum`.

B: How are they similar?

A: Think about this, the `treeSum` function just sums up all the numbers in a tree. This is as if you have a computation graph where all the internal nodes have the operator `"+"`. Compare the following two computation graphs.

B: I see, `treeSum` only do additions, so we don't need operators in the internal nodes. For general arithmetic expressions, every internal node may have a different operator, so we need to do different computations depending on the operator.

A: That is a crucial observation. I believe you can figure out what to do given this similarity.

B: *(Write your answer to `calc` and send them to the teacher.)*

(Exercise omitted for the sample.)