****

A VERY PERSONAL GUIDE FOR SCANNING AND PARSING:

Everything you need to know to build a scanner and a parser

that can evaluate basic arithmetic expressions

KARL LEE

University of Rochester

****

PRELUDE

Towards thee I roll, thou all-destroying but unconquering whale; to the last I grapple with thee; from hell's heart I stab at thee; for hate's sake I spit my last breath at thee. Sink all coffins and all hearses to one common pool! and since neither can be mine let me then tow to pieces, while still chasing thee, though tied to thee, thou damned whale! Thus, I give up the spear!

- Captain Ahab, *Moby Dick* -

Beware; if you are learning about the definite finite automata (DFA) and context-free grammar (CFG) in class, and if you are supposed to contruct a scanner and a parser in some computer language as a school project, the most perilous venture in your computer science career may be at hand. Of course, if you are one of those I-can-figure-out-everything-by-myself kiddos, you will flourish; you need to read no further.

Unfortunately, I was not one of them when I had to do this project. And it became my first incomplete project because I could not complete it – until now. That is, I re-did the school project which I had failed to complete. Now, everything is quite clear (I see why I could not complete it). Though I am certainly not an expert in this field, I figured there are so many aspects in this work that can be, well, enhanced. For instance, many of the obstacles that seemed gargantuan when I was tackling them actually turn out to be a bunch of simple things. Therefore, the purpose of this writing is for you to avoid having to go through the same painful process that I did, and focus on the central learning of the project.

GET THE OVERVIEW

First, you need to thoroughly understand what you are doing. Read the following illustrations closely. Loosely, a **language** is a set of strings with some rules. For example, English is a language. When you exclaim the sentence “I love you!”, you are using some words according to the grammar of the language.

Subect	Action	Object
I	love	you

As the English grammar states, the Subject is doing the Action onto the Object. One way of “understanding” the English sentence is thus to ask oneself

1. Are the words used in the sentence valid (no spelling error)?
2. If so, do they form the sentence according to the agreed-upon grammatical rules?
3. If so, how do they translate into the *meaning* that is supposed to be conveyed?

For the above example, the answers might look

1. *I* check, *l*+*o*+*v*+*e* check, *y*+*o*+*u* check, no spelling error.
2. Subject + Action + Object, check, standard grammar.
3. So it must mean *I* is doing the action of *love* onto *you*.

The step(1) corresponds to **scanning**, the step(2) corresponds to **parsing**, and the step(3) corresponds to **evaluation**. We scan because we want to make sure the strings are correct. We parse because we want to make sure the rules are obeyed.

Instead of the English language, we are dealing with arithmetic expressions. However, the general gist is the same. For instance, we might have the following expression

2+1*3

and answer ourselves the three quetstions:

1. 2 check, + check, 1 check, * check, 3 check, no spelling error.
2. Operators between numbers check, no repeated numbers… , valid.
3. * has higher precedence than +, so it must "mean" 1*3 → 3 → 2+3 → 5.

It is *crucial* to understand the connection between the step(2) and step(3) in our case, because we have to build a parse tree from step(2) for step(3). Evaluating an arithmetic expression by using the result from parsing is *not* the only way to accomplish the task (look up infix/postfix calculator). Personally, I hated having to shove in the rather out-of-place step(3) into step(2), because the process involves a very messy business behind the scence (as you shall observe).

AVOID PITFALLS

The following are the steps you need to take in order to complete the project:

1)	2)	3)	4)	5)
Prepare the systems. Specifically, make sure you can run the code, make sure your language compiler works, make sure you understand each part of the program and its purpose, make sure you can focus on one at a time.	Design a DFA and implement it by using (generally) switch statements.	Design a CFG and implement it by (generally) making a function for each rule.	Carefully, carefully build a tree as you parse so that the resulting tree reflects the "rules" of the arithmetic. Once you have the tree, it is VERY EASY to evaluate correctly.	Debug, do some extra stuff.

For me, the most frustrating part was part(1). The second most would be part(4). Both of them are seemingly quite unrelated to the actual stuff that we are learning: scanning and parsing, DFA and CFG.

Here comes the line that is the most important in this guide.

Do one part at one time, sequentially!

Never, never look ahead of the current part. Of course, it is absolutely vital that in the very beginning you plan out the entire process clearly. But that is so that you know where to go *next*. At the given moment, you must concentrate on the given task. Say, if you are in part(2), you never worry about part(3). *But you must have completed part(1) perfectly!* Otherwise, the flaws will accumulate, and at some point the bugs will be impossible to subside.

Pictorially, if you carefully break this gigantic thing into an organized set of pieces that you can easily follow, you will feel

But if you worry about all the parts at the same time, you will feel hopeless. In my case, I bulldozed through part(1), part(2), part(3) without making sure that they are 100% correct and without absolutely understanding them. In part(4), I was not able to build the parse tree, because I had to face a myriad of problems that had been gathered. I felt

I cannot emphasize strongly enough that you should not let yourself be bogged down by the weight of the whole world. Plan wisely first, do not rush (I know, even though the time is pressing), lay out the future and *only look at the current part!*

HOW TO PROCEED

Now I will give tips on how to actually implement the plan. I realized that understanding the concepts is one thing and implementing it in reality is another. If you are like me, you will hate having to deal with all kinds of stupid, miscellaneous details of programming language. But stand firm, because you have no choice anyway (ha!).

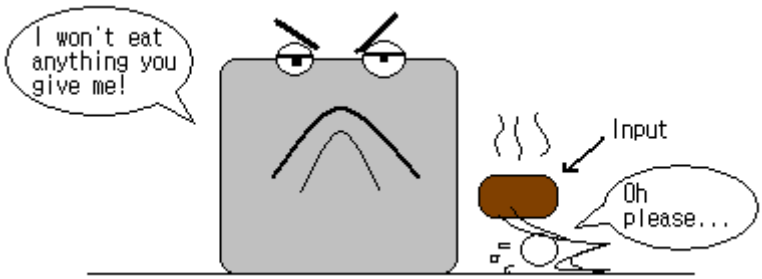
I will mention the obstacles I faced when I did the project. I so much wished there were clear guidelines to overcome those obstacles, but there were none. So I will give them to you now. You may find them valuable, or you may find them a junk.

In **part(1),** to be honest, I could not even begin the project because I did not understand the ways C gets compiled and run. I was supposed to use some utterly alien-sounding called “makefile” to group the different C files together, compile, and run them. Well, I did not know much about the linux environment and so had to learn what was going on pretty hard.

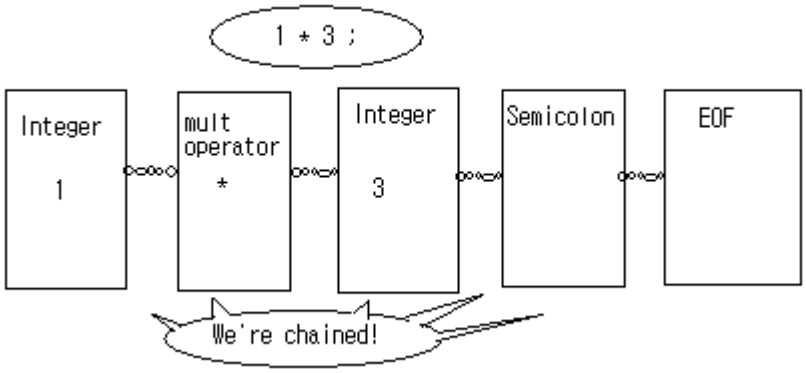
Worse yet, for some mysterious reason I still do not know, the standard way of running the linux environment on the windows did not work on my system at all. Thus I had to swim and gulp and drown for uncountable hours until I somehow figured out “how to compile C programs!” Geez, that was exciting.

My piece of advice: try not to get stuck in technical details, although it is hard to avoid sometimes. If you are one of those who have lived in the world of linux from the age three, good for you! Otherwise, push your way out. Rather than getting caught by that web, I would suggest just using your favorite way of running the C language, whatever that is. In my case, I simply used Microsoft’s Visual C++ for compilation this time, and the programming experience was much more pleasant.

If you are given a pack of stranger’s codes with which to start, make sure you understand what is going on in them. Classify each program file and analyze its purpose. In my case, there were several files that were just used to receive input from the user. You must completely digest how they work! In sum, know how the C language works. You must be able to feed input at your will, manipulate the accepted data, and wield data structures such as linked lists and trees in your language. Learn how to do these before you get even started on this project! Know how to feed.



Part(2) is quite easy if you have a correct DFA and hold an idea how it might be implemented with a series of switch statements. Visualize you are “hopping” from one state to the other as you read the input. You must come up with a way to “hold” the input data that you have just read while scanning. In my case, I put the tokens in a linked list, along with their token type, operation character (if applicable), and value (if applicable). Know how to hold and manipulate information.



In **part(3)**, I had a lot of trouble with writing a CFG. I was scared when I saw all those exotic terms like “Exprtail” or “Atom,” but now I realize that you have to first see *why* a CFG works and *how* to come up with it.

When dealing with a CFG, you must always remember the two essential properties:

1. **Left-associativity** among the operators + and -, among the operators * and /.
2. **Precedence** (i.e. exponentiation > multiplication > addition, et cetera)

The place where the disruptive and rude Mr. Parenthesis comes in may not be so intuitive for some like me. The conclusion I drew is that there is no "set way" to create a CFG. One has to keep the key properties in mind and experiment. Fortunately, we do not need a very complex one in our case. In the beginning, you will no doubt end up with something like this:

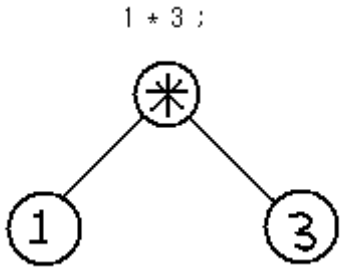
Expr → Expr (+,-) Term | Term
Term → Term (*,/) Factor | Factor
Factor → Atom (^) Factor | Atom
Atom → Num | (+,-) Atom | (Expr)

It is very important that you understand 100% why this CFG preserves the two properties described above. In order to do so, you have to practice with numerous examples and see for yourself it works, along with the understanding of how and why it does.

But you are probably aware that, *although this grammar is perfectly correct*, we cannot directly implement this with a computer language because of the **left-recursion** and **common prefix** problems. To be able to actually implement the CFG, you must fix these two problems. The fix is actually quite easy; there is a formula to do it.

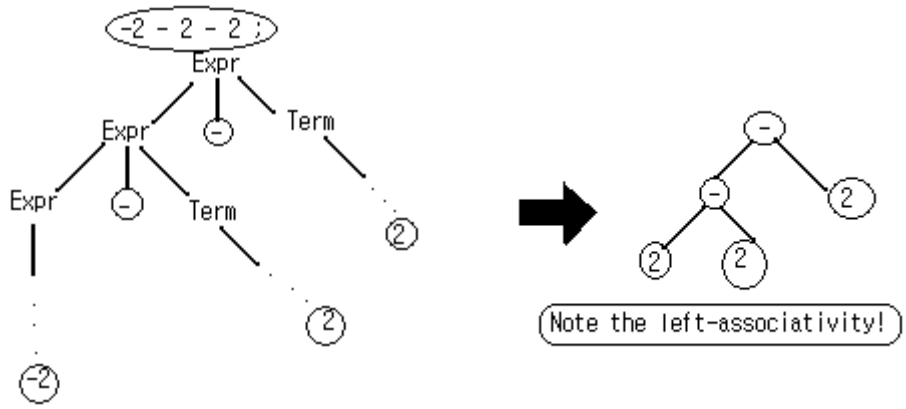
***Part(4)** is my least favorite, though I finally appreciate its educational value. The reason is that although it gives a deep insight into the workings of the parser and data structures, it does not really fit into the stage of scanning and parsing (read GET THE OVERVIEW).

Basically, you have to build a tree from parsing that reflects the arithmetic rules and traverse the tree to evaluate the given expression. My tip: if you have the right tree, evaluating it is a piece of cake. I did not realize it, and I succumbed to the annoyingness of evaluation. Do not repeat my mistake. Know what you need to do, and do it. Your mental image of a parse tree (for arithmetic evaluation) should be something like this:

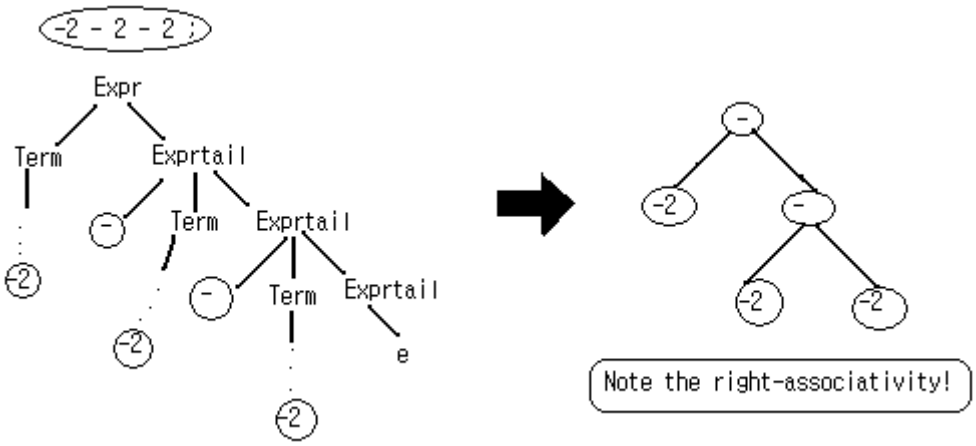


My mistake (rather stupid, really) was that I tried to make every non-terminal of the CFG into a node. Such a tree will be formidably hard to traverse indeed. But once you build this kind of a tree, you can evaluate really easily by recursing, because it is nothing but a simple binary tree (to see why, you have to experiment with other more complicated examples). The trickier part is *how to build such a tree*. And this is where I failed.

The chief reason I failed was that I got completely confused by the behavior of my CFG. The original one (that is, before fixing the left-recursion and common prefix problems) produces a tree that complies to the left-associative property of (+,-) and (*,/) in itself during derivation; you just need to "pick up" the items along the way of parsing. For example, before the fix you will have a branched structure that already has the correct tree shape.



However, after you fix the left-recursion and common prefix problems (that is, after you add a bunch of those "Tail" non-terminals), you will no longer have this nice property.



It turns out that *when you fix the left-recursion, you lose the left-associative property of your CFG structure*. Thus you have no choice but to do some technical works to "force" a correct tree out of the CFG. One way to do it is to pass a *parameter for the function that represents one of these unfortunate non-terminals. In the picture above, say, if you pass a tree built to the left as a parameter to the function that represents Exprtail, and if that tree is added as the left child, we will recover a correct tree. Try to see why.

As for **part(5)**, I will stay silent since I have not done any extra work. Knock yourself out as a reward for having waded so patiently through the bloody pool of part(1), (2), (3), and (4).

CONCLUSION

All in all, this is a very big project. If you do not have an extensive background in programming projects and languages, I presume you will at least partly agree with me that there are a lot of things to take into account. You have to first figure out how to *get started*. Then you need all sorts of code analysis to understand what the heck is going on. You also need to know the theoretical aspects such as the DFA and CFG. There are countless pitfalls along the way, and if you fall into one of them, you will have a very unpleasant experience. Yet building a scanner and parser with your own hands, along with a mechanism to come up with a parse tree for evaluation, is I (now) think the best way to learn the material. So hopefully you will not give up. Even if it seems that some people find it really easy and you are the only one who does not have a clue, do lose your heart. You will be richly rewarded.