Parsing, Precedence

Lecture #02

Charles Averill

Practical Compiler Design
The University of Texas at Dallas

Spring 2023

The Current State of the Compiler

We can scan in a file of recognized tokens and print out those tokens. We can also throw an error when we reach an unrecognized token.

```
(ecco) charlesaverill@pop-os:~/Desktop/ecco$ cat examples/scan_test_pass
222 + 3 * 5 - 8 / 3
(ecco) charlesaverill@pop-os:~/Desktop/ecco$ ./scripts run examples/scan_test_pass
-----RUN-----
/home/charlesaverill/anaconda3/envs/ecco/lib/python3.9/site-packages/setuptools/co
r standards-based tools.
  warnings.warn(
Token:
        TYPE = [integer literal] (5)
        VALUE = 222
Token:
        TYPE = [+] (1)
Token:
        TYPE = [integer literal] (5)
        VALUE = 3
Token:
        TYPE = [*] (3)
Token:
        TYPE = [integer literal] (5)
        VALUE = 5
Token:
        TYPE = [-] (2)
Token:
        TYPE = [integer literal] (5)
        VALUE = 8
Token:
        TYPE = [/] (4)
Token:
        TYPE = [integer literal] (5)
        VALUE = 3
```

```
(ecco) charlesaverill@pop-os:~/Desktop/ecco$ cat examples/scan_test_fail
23 +
18 -
45.6 * 2
/ 18
(ecco) charlesaverill@pop-os:~/Desktop/ecco$ ./scripts run examples/scan_test_fail
-----RUN-----
/home/charlesaverill/anaconda3/envs/ecco/lib/python3.9/site-packages/setuptools/co
r standards-based tools.
  warnings.warn(
Token:
        TYPE = [integer literal] (5)
        VALUE = 23
Token:
        TYPE = [+] (1)
Token:
        TYPE = [integer literal] (5)
        VALUE = 18
Token:
        TYPE = [-] (2)
Token:
        TYPE = [integer literal] (5)
        VALUE = 45
[SYNTAX ERROR] - Uncrecognized token "."
```

This is pretty good. But this doesn't really do anything! We've build a glorified, verbose, special-purpose regex. Let's get to parsing, and build a tiny interpreter along the way before we start generating LLVM pseudoassembly code in the next lecture.

Parsing

In the last lecture we said we'd start off with natural integer literals, and the four basic arithmetic operators: plus, minus, multiply, and divide. Our scanner is already built to recognize these.

Let's look at `ecco/ecco.py` to see what's changed in this iteration.

ecco/ecco.py

- Our first big change is that we now have a `GLOBAL_SCANNER` object. We're going to be looking at the current token and scanning new tokens in multiple places across this project, so we need a shared object across files.
- This required that we don't use the `with ... as` syntax we used for opening the input file, but to be frank that notation is a little cluttering anyways. Calling `GLOBAL_SCANNER.open()` and `GLOBAL_SCANNER.close()` is a sufficient replacement.
- We can see that instead of calling `GLOBAL_SCANNER.scan_file()` we call a new function called `parse_binary_expression()`. Before we get into this, a cursory review of language grammars and recursive parsing is necessary.

BNF Grammars

Language designers frequently make use of a recursive notation called "Backus-Naur Form" (BNF) to describe valid blocks of code in a language. A set of BNF statements makes up a "grammar" that describes all valid inputs to a language.

Our goal is to parse binary arithmetic expressions like $1 + 1$ or $5 * 3 + 2$, where an operator is accompanied by expressions on both sides (integer literals are expressions). Therefore, our BNF grammar will look like this:

```
expression: INTEGER_LITERAL
          | expression "*" expression
          | expression "/" expression
          | expression "+" expression
          | expression "-" expression
```

BNF Grammars

In this example, "expression" is called a **rule** or **non-terminal symbol**, while `INTEGER_LITERAL` is a **terminal symbol**. It is clear from this example that our grammar is recursive, as the subrules that deal with our operators require **expression**s to be present, while the subrules themselves are **expression**s. It is important to note that this grammar does not utilize precedence as we are used to it, we will introduce this feature soon.

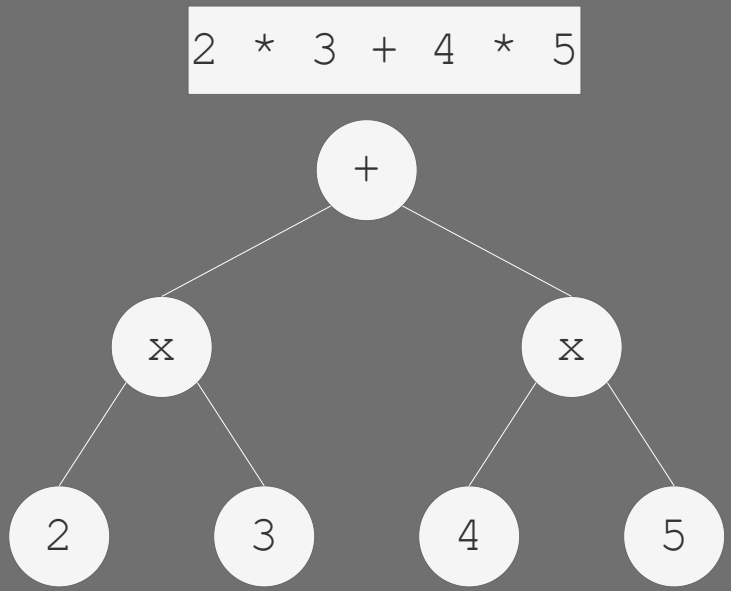
There exist "compiler compilers" that accept BNF as input, and output code that parses source code that conforms with the grammar. We won't be using a compiler compiler, as parsing is a large part of understanding how a compiler works. We'll be writing our own parser, however I will update the BNF of our language as it gets more complex.

Abstract Syntax Trees

One more important topic to cover before we get into recursive parsing is Abstract Syntax Trees, or ASTs. ASTs are a simplified representation of an expression or block of code. Our parser will analyze the tokens scanned by our Scanner and then construct an AST that represents our entire program. ACWJ recommends you read this article: https://medium.com/basecs/leveling-up-ones-parsing-game-with-asts-d7a6fc2400ff, however I don't think it's very necessary. We'll become intimately familiar with ASTs as we manipulate them in code.

Let's look at an AST example, then `ecco/parsing/ecco_ast.py` to see how we'll implement our ASTs. Side note - be careful with how you name your classes, Python's interpreter has many scanning and parsing utilities (The CPython interpreter is bootstrapped, so it has to interpret itself!). Make sure there aren't naming collisions, these will throw confusing errors.

AST Example



When we traverse our AST to generate assembly, we will do a Depth-First Traversal (DFT), so in this case we will generate the products of (2 and 3), and (4 and 5), and then sum them together. Again, this is ideal behavior. We haven't discussed precedence yet, this is just an example of what we would expect our compiler to do. (Actually, we will gloss over parsing without precedence, because it's a fairly easy problem to solve).

ecco/parsing/ecco_ast.py

- We've got an ASTNode class. Remember, the "T" in "AST" stands for "Tree". In this case, a binary tree (we will expand this later). Our ASTNode's data is a `Token` object, and it has `left` and `right` children that can either be `ASTNode`s or `None`.
- Notice that we are going to perform a deep copy of the `Token` object. The `Token` in many cases will be `GLOBAL_SCANNER.current_token`, which changes over time. If we don't deepcopy, our ASTNodes will be filled with references to the latest token read from our input file.
- The rest of the initialization is standard for a tree object. We're making our trees bidirectional for convenience, so we set the `left` and `right` childrens' `parent` to `self`.
- Finally, we have some functions to create specific forms of `ASTNode`s. These will become important later on.

Parsing Expressions

So, we want to do recursive parsing. What does a naïve arithmetic parser look like for the expression `3 + 5 + 6`?

1. Compiler starts. At this point, we should always reach a terminal node! `+ 3 5` is not a valid expression (we're only concerned with infix for ECCO, maybe your compiler is different). In `3 + 5`, the valid equivalent, `3` is a terminal symbol (int literal). So, **parse the terminal token `3`**. Place it into an AST Node and call it "`left`".
2. If we've reached the end of the file, return `left` as our expression. Otherwise, scan the next token (should be an operator!)
3. Recursively scan the next expression (start this routine again on the remaining sub-expression) and call it "`right`".
4. Create an AST Node with `left` and `right` as its children, and the parsed operator as the node's data.

Testing the algorithm out

1. Scan 3 into an AST Node, call it "left1". Scan +, call it "op1", recurse
2. Scan 5 into an AST Node, call it "left2". Scan +, call it "op2", recurse
3. Scan 6 into an AST Node, call it "left3". Reached EOF, so return left3.
4. Construct an AST Node with data op2 and children left2, left3, call it "temp" and return it
5. Construct an AST Node with data op1 and children left1 and temp, and return it.

We can see that this routine handles well for this expression. Students may find it helpful to see that this routine cannot handle precedence, and are invited to compute the AST for `2 * 3 + 4 * 5` using the routine to prove this. My implementation, ACWJ implementation.

Recap

So we've written a naïve parser. Soon we will write a tiny interpreter for our parsed ASTs, but I'd like to implement a parser with precedence first.

We started with a parser without precedence to get your mind thinking about parsing, we're about to replace a bunch of what we just talked about, so don't bother implementing that stuff!

Question before we move on: who knows a way we could implement parsing with precedence? (hint: think more recursion)

Precedence

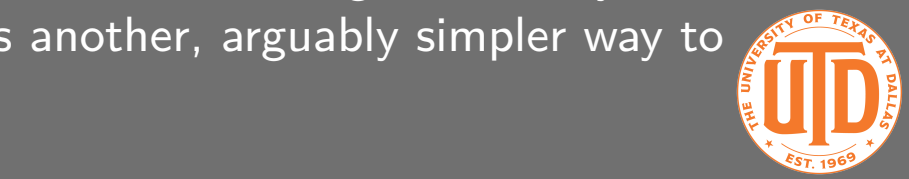
We want our BNF grammar to reflect precedence, so it should look like this:

```
expression: add_expression

add_expression: mul_expression
              | add_expression "+" mul_expression
              | add_expression "-" mul_expression

mul_expression: INTEGER_LITERAL
              | INTEGER_LITERAL '*' mul_expression
              | INTEGER_LITERAL '/' mul_expression
```

We could do this by writing a bunch of recursive parsers using the techniques we've already talked about, and chain them together. This is often how "compiler compilers" will generate your parser for you. That would be time consuming to write by hand, and it would be very messy. Fortunately there is another, arguably simpler way to parse with precedence: Pratt parsing.



Pratt Parsing

Pratt parsing is complicated. It is not as intuitive as recursive descent parsing, but arguably easier to extend, and more concise. Pratt parsing is very good at dealing with expressions, while recursive descent is very good at dealing with structured statements. A good compiler will mix Pratt parsing and recursive descent parsing where it makes sense.

The general idea behind Pratt parsing is that we have a table of precedence for operators rather than an explicit function-call order that encodes precedence. Read this article if you'd like to better understand the intuition behind Pratt parsers. We won't go over them too much in depth, we're just going to cover our implementation for ECCO.

Let's Implement Parsing with Precedence

First let's check out `ecco/ecco.py`. The only change here is that we passed a `0` to `parse_binary_expression()`. Let's inspect this function and see what's changed.

`parse_binary_expression()` actually looks very similar! This is because Pratt parsing is more or less an abstraction of recursive descent parsing. We can see that we now take the previous token's precedence value as input, and we have a `while` loop that ensures that the current token precedence is greater than that of the previous token's precedence.

Quick aside - we're using the C Operator Precedence table for ECCO. In this table, a lower precedence number means that the operator will be processed first (so multiplication has a lower precedence number than addition). ACWJ actually does the reverse, so be careful if you compare the two.

Let's Implement Parsing with Precedence

Inside of the loop, we're going to scan the next token, while holding onto the previous token's type with the `node_type` variable. Now, we will recurse to find the expression on the right hand side of the current operator. We pass in the precedence value of `node_type`, as it is our last-encountered symbol.

After assembling the right-hand-side of the current operator, we join the two sides into an ASTNode with the operator token as the node's type. If we've reached the end of the file, we can `break` the loop and return this newly-created node, otherwise we'll feed it back into the loop and continue parsing.

One last thing!

So we've built an AST, now let's interpret it. This is a small precursor to how we'll generate LLVM in the next lecture, just much simpler (and less powerful).

In `ecco/ecco.py`, we've defined an inline function `interpret_ast()`. This is a recursive function that will traverse our root ASTNode returned from `parse_binary_expression` and perform the appropriate arithmetic operations on the terminal node data it finds.

We call this function on our parsed AST, and tada! We can find the answers to simple arithmetic problems. We could've written a much simpler arithmetic calculator to solve this, but we have now laid the groundwork to compile more and more complex programs. By the beginning of February, we will have a fairly impressive subset of the C Programming Language able to be compiled!

End of Week 1

So that's the end of week 1. The optional homework for this week is to implement left and right bitshift operators. Comprehensive hint:

1. Create bitshift operator Tokens
2. Add their precedence values to the precedence table
3. Update the tiny interpreter we wrote

This is why Pratt parsing is so cool! To accomplish this with recursive descent parsing you'd need to write a new function to handle these operators, and you'd end up duplicating a lot of code.

If you found this week to be a lot of work, don't worry. Remember, the only due date is the end of this semester, and you don't have to complete every topic covered throughout the semester. This week's topics and next week's topics are required, but you have a long time to finish them. After that, most of what we cover will be optional (but encouraged). Please come see me during office hours if you have any questions. I'd love to hear your ideas for your compiler and I would love to help you out with any classwork questions you might have. Have a good weekend!