Homework – LL(1) Parser

Goal:

We wish to complete the <EXP> portion of the BNF into an LL(1) parser. We won't code the BNF directly, its rules are not good enough, but will serve as the inspiration for this assignment. Normally this process would start by either manually or automatically turning the BNF into useful productions. We will skip the BNF -> right-recursive context-free grammar steps for two reasons: 1) the BNF is not sufficient for operator precedence, 2) better productions have already been given to us in the book on page 101.

The goal is to complete the steps necessary to have a functioning LL(1) parser and computing valid arithmetic expressions.

Purposes:

- Practice implementing another's work described in text (often much more difficult than it would seem at first glance).
- Gain experience automating parts of a compiler.
- Utilize pre-existing information to help you double check and debug your work.
- Verify that parsing was success in preparation for a later IR.

Background:

- The LL(1) parser avoids backtracking and allows for an automated tool which can be fed a context-free grammar (CFG) as input and both verify if an expression is in that grammar.
- Our LL(1) parser is a necessary piece of the compiler puzzle. We can use it to assist with building a complete intermediate representation (IR) for later compilation efforts (but that is not necessary for this assignment).
- The <EXP> from our BNF is given below. It helps, but has a big flaw in that it doesn't assist with operator precedence. (Note that BNF can be viewed as a kind of context-free grammar, with non-terminals, terminals, and productions.)

```
<WHITESPACE> ::= \s  // A whitespace character

<DIGIT> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<NUMBER> ::= <DIGIT><BETA>

<BETA> ::= <NUMBER> | e

<NEGATIVE NUMBER> ::= -<NUMBER>

<POS OR NEG> ::= <NUMBER> | <NEGATIVE NUMBER>

<EXP> ::= <LI><AE>

<AE> ::= +<RI><AE> | -<RI><AE> | /<RI><AE> | *<RI><AE> | ^<RI><AE> | e

<LI> ::= <OP> | <PAREN>

<RI> ::= <WHITESPACE><OP> | <PAREN>

<PAREN> ::= (<EXP>)

<OP> ::= <POS OR NEG> | <VARIABLE NAME><REF>
```

- This homework merges together the right-recursive grammar indicated in Figure 3.4 (page 101) and the BNF. They mostly merge together fine, but a few issues crop up. Specifically, these issues:
  - `3^4` is not in Section 3.4
  - `3 / 4` (divide was not in the BNF)
  - This assignment will use `*` and `/` for multiply and divide, the book uses different syntax here
  - `2.051` is not listed as a number in the BNF, this assignment desires that as a number.
  - Spaces are ignored in Section 3.4, while the BNF requires it in one spot.
  - The hyphen `-` character presents a big challenge. It can be subtraction, or negative numbers. We can allow an expression to have both, but we can't allow two hyphens to be immediately adjacent. In other words, we can't support any expression containing `--` as being both a minus and a negative, or somehow a double negative. Likewise, `+-` is also not allowed. For example, consider these test cases:
    - `2-3` (valid)
    - `2 - 3` (valid)
    - `2 - -3` (valid)
    - `2- -3` (valid)
    - `2--3` (invalid)
    - `2+-3` (invalid)
    - `2 -- 3` (invalid)
    - `-2-3` (valid)
    - `-2- -3` (valid)
    - `-2 - -3` (valid)
    - `(-2 - -3)` (valid)
    - `( -2 - -3)` (valid)
    - `-2-(-3)` (valid)
    - `-2-( -3)` (valid)
    - `-2 - (-3)` (valid)
    - `-2 - (-3 - (4))` (valid)
    - `-2 - (-3 - (4))` (valid)
    - `-2 - ( -3 -( -4))` (valid)

    A number or variable can have a `-` character preceding, but only if the word preceding the negative sign is the start of a word, a `(`, or a space.

Instructions:

These instructions are given in the order you should complete them. You want to use the book's approach for the first several bullet points before the assignment adds on support for negative numbers and an exponent operation.

Implement book's solution:

- Start by implementing a data structure to hold all the productions given in Figure 3.4 on page 101. Most solutions are some kind of collection of collections, where you can obtain a production via an index number. A vector of a vector of strings is perhaps the most popular.
- Implement the pre-defined the LL(1) skeleton parser and hard code the table found in Figure 3.11 on page 112.
  - Note that this algorithm utilizes a NextWord() function. You may use whatever solution you want for your NextWord(), whether it be ad-hoc or fully derived from insights in Chapter 2.
  - Note that the table's integer values refer back to the production rules in Figure 3.4 on page 101.

- Verify your code is working for all inputs that don't test negative numbers or exponent operations. An example of how an expression should parse is found in Figure 3.3 on page 114.
- Implement the code to create FIRST, FOLLOW, and FIRST$^+$ sets. These are described in the section Backtrack-Free Parsing (page 103), and are covered in Figures 3.7, 3.8, and the text. Note that when you have FIRST, you can build FOLLOW, and once those two are built, you can build FIRST$^+$. Please reach out if you struggle to follow the book's syntax here, and I will assist in explaining it to you.
- Implement the table-construction on Figure 3.12 on page 113. Make sure the table you generate matches the table on page 112.

Extend book's solution:

- Now prepare to verify all expressions with negative values. Below is a fix for the productions. (You may use your own set of productions if they work, I have a nagging suspicion these can be condensed by several productions.)

```
0. Goal          -> Expr
1. Expr          -> LTerm Expr'
2. LTerm         -> LFactor Term'
3. RTerm         -> RFactor Term'
4. Expr'         -> + RTerm Expr'
5.               |  - RTerm Expr'
6.               |  ϵ
7. Term'         -> * RTerm Expr'
8.               |  / RTerm Expr'
9.               |  ϵ
10. LFactor      -> GFactor
11.              |  negnum    //negative val without space only left term
12.              |  negname   //negative name without space only left term
13. RFactor      -> GFactor
14. GFactor      -> ( Expr )
15.              |  PosVal
16.              |  SpaceNegVal
17. PosVal       -> num
18.              |  name
19. SpaceNegVal  -> spacenegnum
20.              |  spacenegname
```

  - You will want to modify your nextWord() function to support leading spaces prior to negative numbers. The easiest approach is to first leave leading spaces in the word. Then perform a cleanup check, if the sequence of characters is spaces, followed by a hypen, and followed by a digit, then keep one leading space. Otherwise, remove all leading spaces. So " -32" turns into " -32", while " +" turns into "+". In the above productions, "32" is a num, "-32" is a negnum, " -32" is a spacenegnum, "var3" is a name, "-var3" is a negname, and " -var3" is a spacenegnum.
  - Modify your code with these productions. Generate a new table.
  - Test all the expressions in the .txt file that contain negative values.
- Prepare to verify all expressions with power/exponent operators. Modify the prior productions as necessary to support them. Implement this into your code, and generate a new table.
  - Test all the expressions in the .txt files that contain the ^ symbol.
- This next section *is not specified directly in the textbook and is open for your implementation.* You

must verify that you are not just validating syntactically correct expressions, but rather verifying you can determine the correct order of instructions. To accomplish this, you must compute expressions that have enough information to be computed (such as any expression without a variable name), these expressions must be computed and an answer displayed. `3 + 4 * 5` should compute the same solution as `3 + (4 * 5)`. The expression `(3 + 4) * 5` should be different.

Verification:

- Verify that you can parse the all the inputs provided in the .txt files
- Show in the output the expression you parsed, if it is valid or invalid, and what its computed value is (if valid and if it can be computed). For example, 5 + 4 * 3 should compute to 17, not 27. 5 + 4 * 3 ^ 2 should compute to 41, and not 729 or 3600.