Isabella Sims

Hw 2: Understanding the parser

- **Due** Wednesday by 11:59pm
- **Points** 100
- **Submitting** a text entry box, a website url, or a file upload

Like the previous assignment on understanding the scanner, this one asks you to investigate how the parser works. As before, we're starting it in class on October 15, and you can either work in groups or individually. If you start in a group and don't finish in class, you can either continue in the group outside of class, or finish it individually. If you start and finish in a group, you can have just one person in the group turn in the assignment, with everyone's names on it.

Q1: Different parse errors

Consider the following two expressions that both produce parse errors:

(2

(2 +

Why does only the first one, but not the second one, complain about the unclosed parenthesis, even though this is an error that's present in both of them?

Because the parser will always throw it's error at the end of the line (aka the last token evaluated) and only throws one error at a time.

The first error comes from the if statement of line 92 in the parser. The if statement here is satisfied because the last paren is the last "grammer" token we see here. So, the parser remembers the '(' as the last token that needs to be matched. In this case, it looks for a closing paren, doesn't encounter it, and thus throws an error about unclosed parentheses. Because the 2 is not followed by any instruction, it can stand alone. The parser does not require that anything follow the 2, so the unclosed paren is the only concern. Close the paren, no remaining errors.

The second statement also throws an error regarding the last token of the line: in this case it is ‘+’. We get one error at a time so this is to be expected. Unlike the first statement though, closing the paren of the second statement will still result in an error as the interpreter did not expect to see a closing paren yet. At this point, the interpreter is only concerned with finding a match for the ‘2’ on the other side of the ‘+’ sign. A ‘+’ must be followed by another integer, which is why the error is concerned with an ‘expression’ rather than a closing paren. This is a more complex error, it needs to satisfy the formula `(expr, operator, right)` whereas the previous statement is allowed to exist as its own expression `(expr)` and the error knows exactly what is missing. Both errors are syntax errors.

Additionally, the statements throw different errors because (2) would be acceptable, whereas (2+) would not be. In theory, (2+) should throw the same error as (2+ would, but the error would be at the closing ‘)’ paren instead of “at end” because a paren is not an acceptable way to end the line when the expression hasn’t been satisfied. Errors are thrown “at end” when the last token does not require a match to be acceptable, but the parser remembers a previous token that did not receive a match by the end of the line.

Q2: Different types of syntax errors

We have seen two kinds of syntax errors so far: input that doesn’t scan/tokenize correctly, and input that scans/tokenizes correctly but does not produce a valid parse tree. Give two examples of each kind of error.

Type 1: doesn’t scan/tokenize

^

Error: Unexpected character.

%

Error: Unexpected character.

Type 2: scans/tokenizes but does not parse

(1+3 =)

Error at '=': Expect ')' after expression.

(1 / 2 = +)

Error at '=': Expect ')' after expression.

Q3: Trace a successful parse

Trace how the following expression is parsed, including the parentheses as part of the input:

(2 + 7)

What series of functions are called to produce the end result? You can skip built-in Java functions. In addition, for the purposes of this question, you can start from the point where Main.run() calls parser.parse(), i.e. you can assume the Scanner has already been run, and don't have to trace it.

`Main.run() calls parser.parse():`

```
    Parser parser = new Parser(tokens);

    Expr expression = parser.parse();
```

`Parser.parse() executes the try statement to return expression();`

```
parser.parse(){return expression();} → calls parser.expression()
```

`parser.expression() calls parser.equality()`

```
parser.equality() {while statement not executed → calls
parser.comparison()}
```

`parser.comparison() calls parser.term() calls parser.factor():`

```
parser.factor() returns expression
```

```
calls parser.unary():

      operator and right are assigned values

      calls primary()

            Calls match()

                  Left paren is evaluated (consumed)

                  2 is evaluated as a literal (consumed)

                  7 is evaluated as a literal (consumed)

                  Right paren is evaluated (consumed)
```

Q4: Parsing other kinds of things

The parser right now only parses expressions. In Parser.java, parse() tries to parse the token stream as an expression, using the recursive-descent parser starting with the call expression(). It returns an Expr if it succeeds.

What would need to added and/or change for other kinds of things to be parsed? For example, what if we wanted to parse variable declarations like “var x = 5;”? You don't have to implement this (we'll be doing that in a week or two), but describe in general terms what types of changes would be needed, and where.

Hint 1: there would be 3 main additions/changes needed.

Hint 2: You might skim Chapter 8 as a preview.

1. Extend grammar with statements so you can work with states- interpreter needs to be able to remember, not just process
2. Define variable syntax
    a. Add declaration grammar to the parser
3. Store the variable values
    a. The var values need to be stored somewhere called an “environment”

b. Create Environment.java, implement a hash map in environment.java
c. Create an operation to define variables that binds its name to its value
    i. Optional: add functionality to prohibit redeclaration of variables (throw an error) ---> personally wouldn't use this with the keyword var but could be useful if I wanted to implement constants
d. Create an operation to get the value from a variable name