Let’s Build A Simple Interpreter. Part 3. (https://ruslanspivak.com/lsbasi-part3/)

Date Wed, August 12, 2015

I woke up this morning and I thought to myself: “Why do we find it so difficult to learn a new skill?”

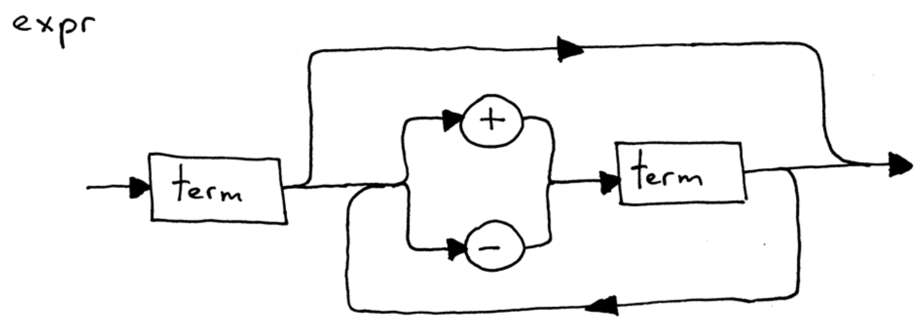
I don’t think it’s just because of the hard work. I think that one of the reasons might be that we spend a lot of time and hard work acquiring knowledge by reading and watching and not enough time translating that knowledge into a skill by practicing it. Take swimming, for example. You can spend a lot of time reading hundreds of books about swimming, talk for hours with experienced swimmers and coaches, watch all the training videos available, and you still will sink like a rock the first time you jump in the pool.

The bottom line is: it doesn’t matter how well you think you know the subject - you have to put that knowledge into practice to turn it into a skill. To help you with the practice part I put exercises into Part 1 (http://ruslanspivak.com/lsbasi-part1/) and Part 2 (http://ruslanspivak.com/lsbasi-part2/) of the series. And yes, you will see more exercises in today’s article and in future articles, I promise :)

Okay, let’s get started with today’s material, shall we?

So far, you’ve learned how to interpret arithmetic expressions that add or subtract two integers like “7 + 3” or “12 - 9”. Today I’m going to talk about how to parse (recognize) and interpret arithmetic expressions that have any number of plus or minus operators in it, for example “7 - 3 + 2 - 1”.

Graphically, the arithmetic expressions in this article can be represented with the following syntax diagram:



What is a syntax diagram? A **syntax diagram** is a graphical representation of a programming language's syntax rules. Basically, a syntax diagram visually shows you which statements are allowed in your programming language and which are not.

Syntax diagrams are pretty easy to read: just follow the paths indicated by the arrows. Some paths indicate choices. And some paths indicate loops.

You can read the above syntax diagram as following: a term optionally followed by a plus or minus sign, followed by another term, which in turn is optionally followed by a plus or minus sign followed by another term and so on. You get the picture, literally. You might wonder what a *"term"* is. For the purpose of this article a *"term"* is just an integer.

Syntax diagrams serve two main purposes:

- They graphically represent the specification (grammar) of a programming language.
- They can be used to help you write your parser - you can map a diagram to code by following simple rules.

You've learned that the process of recognizing a phrase in the stream of tokens is called **parsing**. And the part of an interpreter or compiler that performs that job is called a **parser**. Parsing is also called **syntax analysis**, and the parser is also aptly called, you guessed it right, a **syntax analyzer**.

According to the syntax diagram above, all of the following arithmetic expressions are valid:

- 3
- 3 + 4
- 7 - 3 + 2 - 1

Because syntax rules for arithmetic expressions in different programming languages are very similar we can use a Python shell to "test" our syntax diagram. Launch your Python shell and see for yourself:

```
>>> 3
3
>>> 3 + 4
7
>>> 7 - 3 + 2 - 1
5
```

No surprises here.

The expression "3 + " is not a valid arithmetic expression though because according to the syntax diagram the plus sign must be followed by a *term* (integer), otherwise it's a syntax error. Again, try it with a Python shell and see for yourself:

```
>>> 3 +
  File "<stdin>", line 1
    3 +
      ^
SyntaxError: invalid syntax
```

It's great to be able to use a Python shell to do some testing but let's map the above syntax diagram to code and use our own interpreter for testing, all right?

You know from the previous articles (Part 1 (http://ruslanspivak.com/lsbasi-part1/) and Part 2 (http://ruslanspivak.com/lsbasi-part2/)) that the *expr* method is where both our parser and interpreter live. Again, the parser just recognizes the structure making sure that it corresponds to some specifications and the interpreter actually evaluates the expression once the parser has successfully recognized (parsed) it.

The following code snippet shows the parser code corresponding to the diagram. The rectangular box from the syntax diagram (*term*) becomes a *term* method that parses an integer and the *expr* method just follows the syntax diagram flow:

```
def term(self):
    self.eat(INTEGER)

def expr(self):
    # set current token to the first token taken from the input
    self.current_token = self.get_next_token()

    self.term()
    while self.current_token.type in (PLUS, MINUS):
        token = self.current_token
        if token.type == PLUS:
            self.eat(PLUS)
            self.term()
        elif token.type == MINUS:
            self.eat(MINUS)
            self.term()
```

You can see that *expr* first calls the *term* method. Then the *expr* method has a *while* loop which can execute zero or more times. And inside the loop the parser makes a choice based on the token (whether it’s a plus or minus sign). Spend some time proving to yourself that the code above does indeed follow the syntax diagram flow for arithmetic expressions.

The parser itself does not interpret anything though: if it recognizes an expression it’s silent and if it doesn’t, it throws out a syntax error. Let’s modify the *expr* method and add the interpreter code:

```python
def term(self):
    """Return an INTEGER token value"""
    token = self.current_token
    self.eat(INTEGER)
    return token.value

def expr(self):
    """Parser / Interpreter """
    # set current token to the first token taken from the input
    self.current_token = self.get_next_token()

    result = self.term()
    while self.current_token.type in (PLUS, MINUS):
        token = self.current_token
        if token.type == PLUS:
            self.eat(PLUS)
            result = result + self.term()
        elif token.type == MINUS:
            self.eat(MINUS)
            result = result - self.term()

    return result
```

Because the interpreter needs to evaluate an expression the *term* method was modified to return an integer value and the *expr* method was modified to perform addition and subtraction at the appropriate places and return the result of interpretation. Even though the code is pretty straightforward I recommend spending some time studying it.

Le’s get moving and see the complete code of the interpreter now, okay?

Here is the source code for your new version of the calculator that can handle valid arithmetic expressions containing integers and any number of addition and subtraction operators:

```python
# Token types
#
# EOF (end-of-file) token is used to indicate that
# there is no more input left for lexical analysis
INTEGER, PLUS, MINUS, EOF = 'INTEGER', 'PLUS', 'MINUS', 'EOF'


class Token(object):
    def __init__(self, type, value):
        # token type: INTEGER, PLUS, MINUS, or EOF
        self.type = type
        # token value: non-negative integer value, '+', '-', or None
        self.value = value

    def __str__(self):
        """String representation of the class instance.

        Examples:
            Token(INTEGER, 3)
            Token(PLUS, '+')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()


class Interpreter(object):
    def __init__(self, text):
        # client string input, e.g. "3 + 5", "12 - 5 + 3", etc
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        # current token instance
        self.current_token = None
        self.current_char = self.text[self.pos]

    ##########################################################
    # Lexer code                                             #
    ##########################################################
    def error(self):
        raise Exception('Invalid syntax')

    def advance(self):
        """Advance the `pos` pointer and set the `current_char` variable."""
        self.pos += 1
        if self.pos > len(self.text) - 1:
            self.current_char = None  # Indicates end of input
        else:
            self.current_char = self.text[self.pos]

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()
    def integer(self):
        """Return a (multidigit) integer consumed from the input."""
        result = ''
        while self.current_char is not None and self.current_char.isdigit():
            result += self.current_char
            self.advance()
        return int(result)

    def get_next_token(self):
        """Lexical analyzer (also known as scanner or tokenizer)

        This method is responsible for breaking a sentence
        apart into tokens. One token at a time.
        """
        while self.current_char is not None:

            if self.current_char.isspace():
                self.skip_whitespace()
                continue

            if self.current_char.isdigit():
                return Token(INTEGER, self.integer())

            if self.current_char == '+':
                self.advance()
                return Token(PLUS, '+')

            if self.current_char == '-':
                self.advance()
                return Token(MINUS, '-')

            self.error()

        return Token(EOF, None)

    ##########################################################
    # Parser / Interpreter code                              #
    ##########################################################
    def eat(self, token_type):
        # compare the current token type with the passed token
        # type and if they match then "eat" the current token
        # and assign the next token to the self.current_token,
        # otherwise raise an exception.
        if self.current_token.type == token_type:
            self.current_token = self.get_next_token()
        else:
            self.error()

    def term(self):
        """Return an INTEGER token value."""
        token = self.current_token
        self.eat(INTEGER)
        return token.value

    def expr(self):
        """Arithmetic expression parser / interpreter."""
        # set current token to the first token taken from the input
        self.current_token = self.get_next_token()
        result = self.term()
        while self.current_token.type in (PLUS, MINUS):
            token = self.current_token
            if token.type == PLUS:
                self.eat(PLUS)
                result = result + self.term()
            elif token.type == MINUS:
                self.eat(MINUS)
                result = result - self.term()

        return result


def main():
    while True:
        try:
            # To run under Python3 replace 'raw_input' call
            # with 'input'
            text = raw_input('calc> ')
        except EOFError:
            break
        if not text:
            continue
        interpreter = Interpreter(text)
        result = interpreter.expr()
        print(result)


if __name__ == '__main__':
    main()
```

Save the above code into the *calc3.py* file or download it directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part3/calc3.py). Try it out. See for yourself that it can handle arithmetic expressions that you can derive from the syntax diagram I showed you earlier.

Here is a sample session that I ran on my laptop:

```
$ python calc3.py
calc> 3
3
calc> 7 - 4
3
calc> 10 + 5
15
calc> 7 - 3 + 2 - 1
5
calc> 10 + 1 + 2 - 3 + 4 + 6 - 15
5
calc> 3 +
Traceback (most recent call last):
  File "calc3.py", line 147, in <module>
    main()
  File "calc3.py", line 142, in main
    result = interpreter.expr()
  File "calc3.py", line 123, in expr
    result = result + self.term()
  File "calc3.py", line 110, in term
    self.eat(INTEGER)
  File "calc3.py", line 105, in eat
    self.error()
  File "calc3.py", line 45, in error
    raise Exception('Invalid syntax')
Exception: Invalid syntax
```

Remember those exercises I mentioned at the beginning of the article: here they are, as promised :)

- Draw a syntax diagram for arithmetic expressions that contain only multiplication and division, for example “7 * 4 / 2 * 3”. Seriously, just grab a pen or a pencil and try to draw one.
- Modify the source code of the calculator to interpret arithmetic expressions that contain only multiplication and division, for example “7 * 4 / 2 * 3”.
- Write an interpreter that handles arithmetic expressions like “7 - 3 + 2 - 1” from scratch. Use any programming language you’re comfortable with and write it off the top of your head without looking at the examples. When you do that, think about components involved: a *lexer* that takes an input and converts it into a stream of tokens, a *parser* that feeds off the stream of the tokens provided by the *lexer* and tries to recognize a structure in that stream, and an *interpreter* that generates results after the *parser* has successfully parsed (recognized) a valid arithmetic expression. String those pieces together. Spend some time

translating the knowledge you've acquired into a working interpreter for arithmetic expressions.

Check your understanding.

1. What is a syntax diagram?
2. What is syntax analysis?
3. What is a syntax analyzer?

Hey, look! You read all the way to the end. Thanks for hanging out here today and don't forget to do the exercises. :) I'll be back next time with a new article - stay tuned.

Here is a list of books I recommend that will help you in your study of interpreters and compilers:

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)
2. Writing Compilers and Interpreters: A Software Engineering Approach (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=UCLGQTPIYSWYKRRM)
3. Modern Compiler Implementation in Java (http://www.amazon.com/gp/product/052182060X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=052182060X&linkCode=as2&tag=russblo0b-20&linkId=ZSKKZMV7YWR22NMW)
4. Modern Compiler Design (http://www.amazon.com/gp/product/1461446988/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=1461446988&linkCode=as2&tag=russblo0b-20&linkId=PAXWJP5WCPZ7RKRD)
5. Compilers: Principles, Techniques, and Tools (2nd Edition) (http://www.amazon.com/gp/product/0321486811/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321486811&linkCode=as2&tag=russblo0b-20&linkId=GOEGDQG4HIHU56FQ)