Let’s Build A Simple Interpreter. Part 2. (https://ruslanspivak.com/lsbasi-part2/)

Date Fri, July 03, 2015

In their amazing book “The 5 Elements of Effective Thinking” the authors Burger and Starbird share a story about how they observed Tony Plog, an internationally acclaimed trumpet virtuoso, conduct a master class for accomplished trumpet players. The students first played complex music phrases, which they played perfectly well. But then they were asked to play very basic, simple notes. When they played the notes, the notes sounded childish compared to the previously played complex phrases. After they finished playing, the master teacher also played the same notes, but when he played them, they did not sound childish. The difference was stunning. Tony explained that mastering the performance of simple notes allows one to play complex pieces with greater control. The lesson was clear - to build true virtuosity one must focus on mastering simple, basic ideas.[1]

The lesson in the story clearly applies not only to music but also to software development. The story is a good reminder to all of us to not lose sight of the importance of deep work on simple, basic ideas even if it sometimes feels like a step back. While it is important to be proficient with a tool or framework you use, it is also extremely important to know the principles behind them. As Ralph Waldo Emerson said:

> *“If you learn only methods, you’ll be tied to your methods. But if you learn principles, you can devise your own methods.”*

On that note, let’s dive into interpreters and compilers again.

Today I will show you a new version of the calculator from Part 1 (http://ruslanspivak.com/lsbasi-part1/) that will be able to:

1. Handle whitespace characters anywhere in the input string
2. Consume multi-digit integers from the input
3. Subtract two integers (currently it can only add integers)

Here is the source code for your new version of the calculator that can do all of the above:

```
# 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", 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]

    def error(self):
        raise Exception('Error parsing input')

    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.
        """
        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)

    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 expr(self):
        """Parser / Interpreter

        expr -> INTEGER PLUS INTEGER
        expr -> INTEGER MINUS INTEGER
        """
        # set current token to the first token taken from the input
        self.current_token = self.get_next_token()

        # we expect the current token to be an integer
        left = self.current_token
        self.eat(INTEGER)

        # we expect the current token to be either a '+' or '-'
        op = self.current_token
        if op.type == PLUS:
            self.eat(PLUS)
```

```
        else:
            self.eat(MINUS)

        # we expect the current token to be an integer
        right = self.current_token
        self.eat(INTEGER)
        # after the above call the self.current_token is set to
        # EOF token

        # at this point either the INTEGER PLUS INTEGER or
        # the INTEGER MINUS INTEGER sequence of tokens
        # has been successfully found and the method can just
        # return the result of adding or subtracting two integers,
        # thus effectively interpreting client input
        if op.type == PLUS:
            result = left.value + right.value
        else:
            result = left.value - right.value
        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 *calc2.py* file or download it directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part2/calc2.py). Try it out. See for yourself that it works as expected: it can handle whitespace characters anywhere in the input; it can accept multi-digit integers, and it can also subtract two integers as well as add two integers.

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

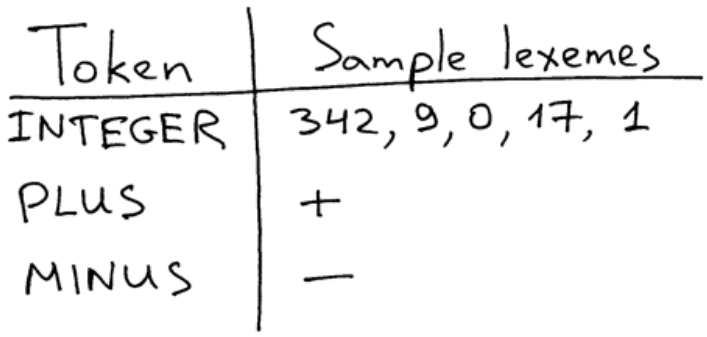
```
$ python calc2.py
calc> 27 + 3
30
calc> 27 - 7
20
calc>
```

The major code changes compared with the version from Part 1 (http://ruslanspivak.com/lsbasi-part1/) are:

1. The *get_next_token* method was refactored a bit. The logic to increment the *pos* pointer was factored into a separate method *advance*.
2. Two more methods were added: *skip_whitespace* to ignore whitespace characters and *integer* to handle multi-digit integers in the input.
3. The *expr* method was modified to recognize INTEGER → MINUS → INTEGER phrase in addition to INTEGER → PLUS → INTEGER phrase. The method now also interprets both addition and subtraction after having successfully recognized the corresponding phrase.

In Part 1 (http://ruslanspivak.com/lsbasi-part1/) you learned two important concepts, namely that of a **token** and a **lexical analyzer**. Today I would like to talk a little bit about **lexemes**, **parsing**, and **parsers**.

You already know about tokens. But in order for me to round out the discussion of tokens I need to mention lexemes. What is a lexeme? A **lexeme** is a sequence of characters that form a token. In the following picture you can see some examples of tokens and sample lexemes and hopefully it will make the relationship between them clear:

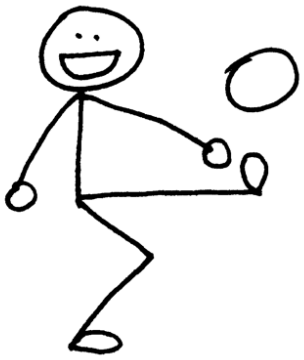


Now, remember our friend, the *expr* method? I said before that that's where the interpretation of an arithmetic expression actually happens. But before you can interpret an expression you first need to recognize what kind of phrase it is, whether it is addition or subtraction, for example. That's what the *expr* method essentially does: it finds the structure in the stream of tokens it gets from the *get_next_token* method and then it interprets the phrase that is has recognized, generating the result of the arithmetic expression.

The process of finding the structure in the stream of tokens, or put differently, the process of recognizing a phrase in the stream of tokens is called **parsing**. The part of an interpreter or compiler that performs that job is called a **parser**.

So now you know that the *expr* method is the part of your interpreter where both **parsing** and **interpreting** happens - the *expr* method first tries to recognize (**parse**) the INTEGER → PLUS → INTEGER or the INTEGER → MINUS → INTEGER phrase in the stream of tokens and after it has successfully recognized (**parsed**) one of those phrases, the method interprets it and returns the result of either addition or subtraction of two integers to the caller.

And now it's time for exercises again.



1. Extend the calculator to handle multiplication of two integers
2. Extend the calculator to handle division of two integers
3. Modify the code to interpret expressions containing an arbitrary number of additions and subtractions, for example “9 - 5 + 3 + 11”

Check your understanding.

1. What is a lexeme?
2. What is the name of the process that finds the structure in the stream of tokens, or put differently, what is the name of the process that recognizes a certain phrase in that stream of tokens?
3. What is the name of the part of the interpreter (compiler) that does parsing?

I hope you liked today’s material. In the next article of the series you will extend your calculator to handle more complex arithmetic expressions. Stay tuned.

And 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)

If you want to get my newest articles in your inbox, then enter your email address below and click "Get Updates!"

Enter Your First Name *

Enter Your Best Email *

Get Updates!

All articles in this series:

- Let's Build A Simple Interpreter. Part 1. (/lsbasi-part1/)
- Let's Build A Simple Interpreter. Part 2. (/lsbasi-part2/)
- Let's Build A Simple Interpreter. Part 3. (/lsbasi-part3/)
- Let's Build A Simple Interpreter. Part 4. (/lsbasi-part4/)
- Let's Build A Simple Interpreter. Part 5. (/lsbasi-part5/)
- Let's Build A Simple Interpreter. Part 6. (/lsbasi-part6/)
- Let's Build A Simple Interpreter. Part 7: Abstract Syntax Trees (/lsbasi-part7/)
- Let's Build A Simple Interpreter. Part 8. (/lsbasi-part8/)
- Let's Build A Simple Interpreter. Part 9. (/lsbasi-part9/)
- Let's Build A Simple Interpreter. Part 10. (/lsbasi-part10/)
- Let's Build A Simple Interpreter. Part 11. (/lsbasi-part11/)
- Let's Build A Simple Interpreter. Part 12. (/lsbasi-part12/)
- Let's Build A Simple Interpreter. Part 13: Semantic Analysis (/lsbasi-part13/)
- Let's Build A Simple Interpreter. Part 14: Nested Scopes and a Source-to-Source Compiler (/lsbasi-part14/)
- Let's Build A Simple Interpreter. Part 15. (/lsbasi-part15/)
- Let's Build A Simple Interpreter. Part 16: Recognizing Procedure Calls (/lsbasi-part16/)
- Let's Build A Simple Interpreter. Part 17: Call Stack and Activation Records (/lsbasi-part17/)
- Let's Build A Simple Interpreter. Part 18: Executing Procedure Calls (/lsbasi-part18/)
- Let's Build A Simple Interpreter. Part 19: Nested Procedure Calls (/lsbasi-part19/)

1. The 5 Elements of Effective Thinking (http://www.amazon.com/gp/product/0691156662/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0691156662&linkCode=as2&tag=russblo0b-20&linkId=B7GSVLONUPCIBIVY) ↩

Comments