Let's Build A Simple Interpreter. Part 8. (https://ruslanspivak.com/lsbasi-part8/)

Date Mon, January 18, 2016

Today we'll talk about **unary operators**, namely unary plus (+) and unary minus (–) operators.

A lot of today's material is based on the material from the previous article, so if you need a refresher just head back to Part 7 (http://ruslanspivak.com/lsbasi-part7/) and go over it again. Remember: repetition is the mother of all learning.

Having said that, this is what you are going to do today:

- extend the grammar to handle unary plus and unary minus operators
- add a new *UnaryOp* AST node class
- extend the parser to generate an AST with *UnaryOp* nodes
- extend the interpreter and add a new *visit_UnaryOp* method to interpret unary operators

Let's get started, shall we?

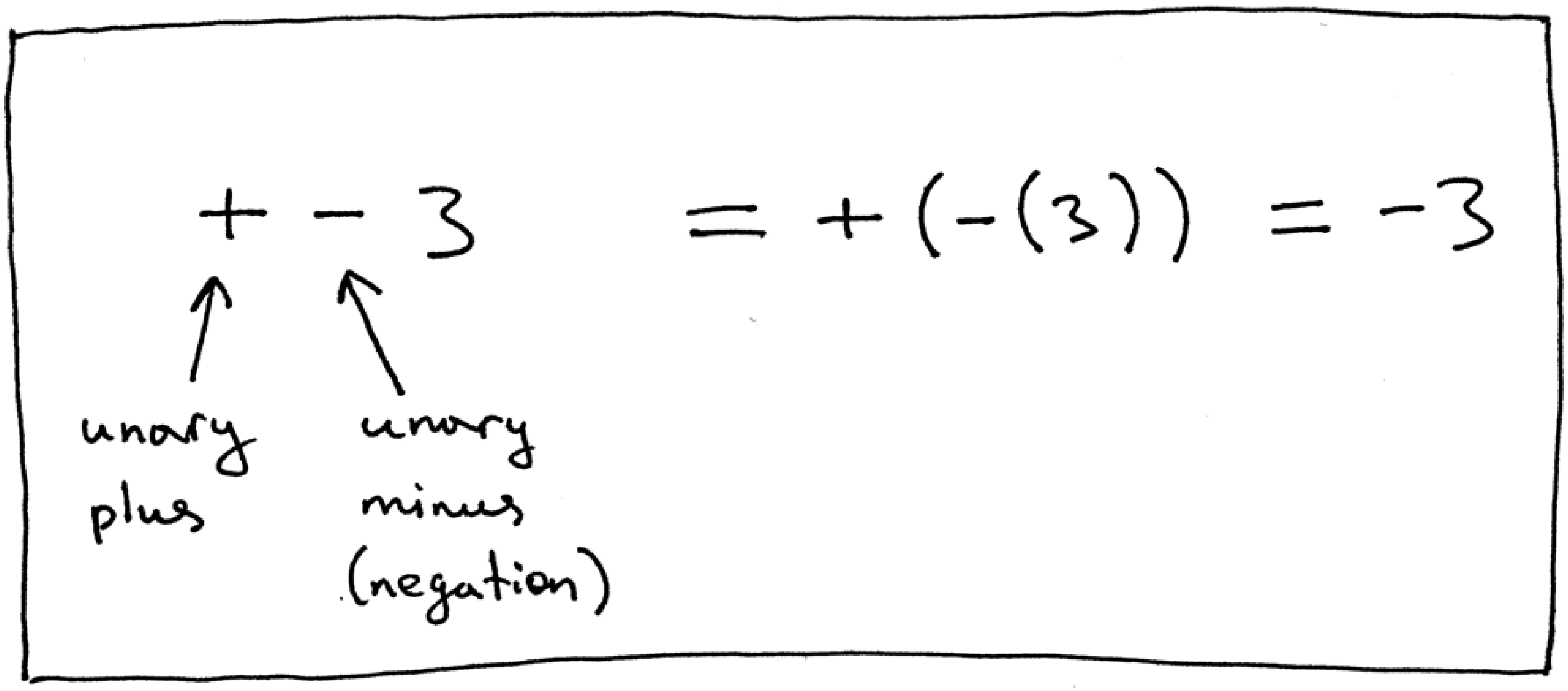
So far we've worked with binary operators only (+, –, *, /), that is, the operators that operate on two operands.

What is a unary operator then? A *unary operator* is an operator that operates on one *operand* only.

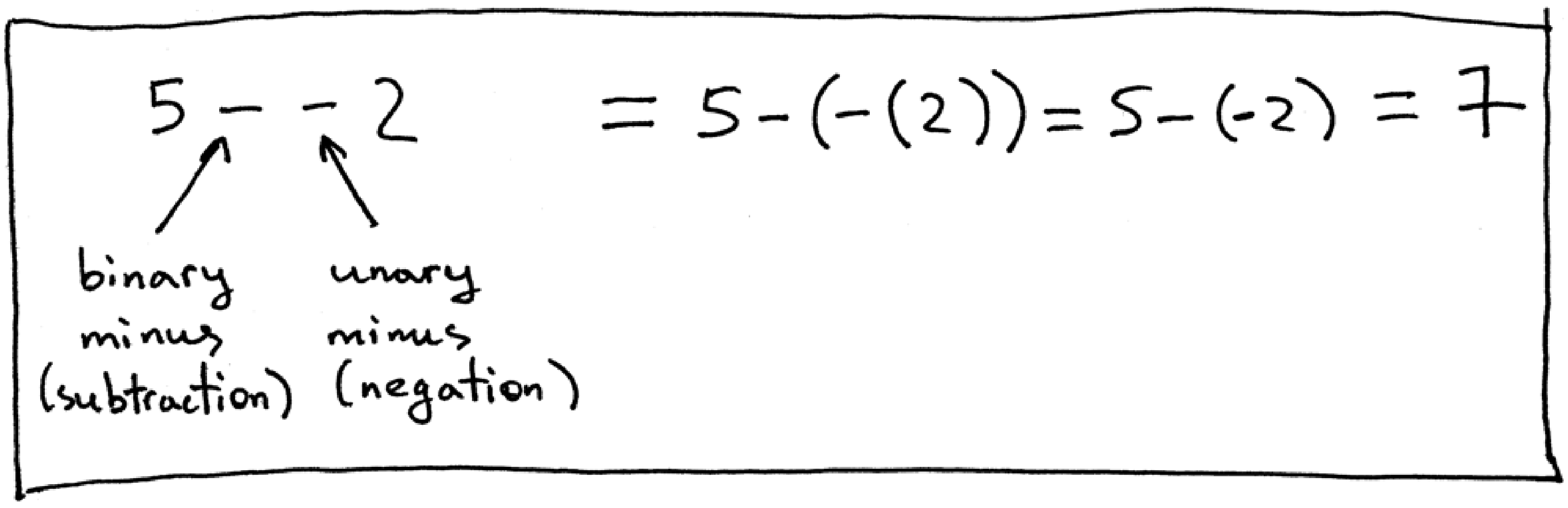
Here are the rules for unary plus and unary minus operators:

- The unary minus (–) operator produces the negation of its numeric operand
- The unary plus (+) operator yields its numeric operand without change
- The unary operators have higher precedence than the binary operators +, –, *, and /

In the expression "+ – 3" the first '+' operator represents the unary plus operation and the second '–' operator represents the unary minus operation. The expression "+ – 3" is equivalent to "+ (– (3))" which is equal to –3. One could also say that **–3** in the expression is a negative integer, but in our case we treat it as a unary minus operator with 3 as its positive integer operand:

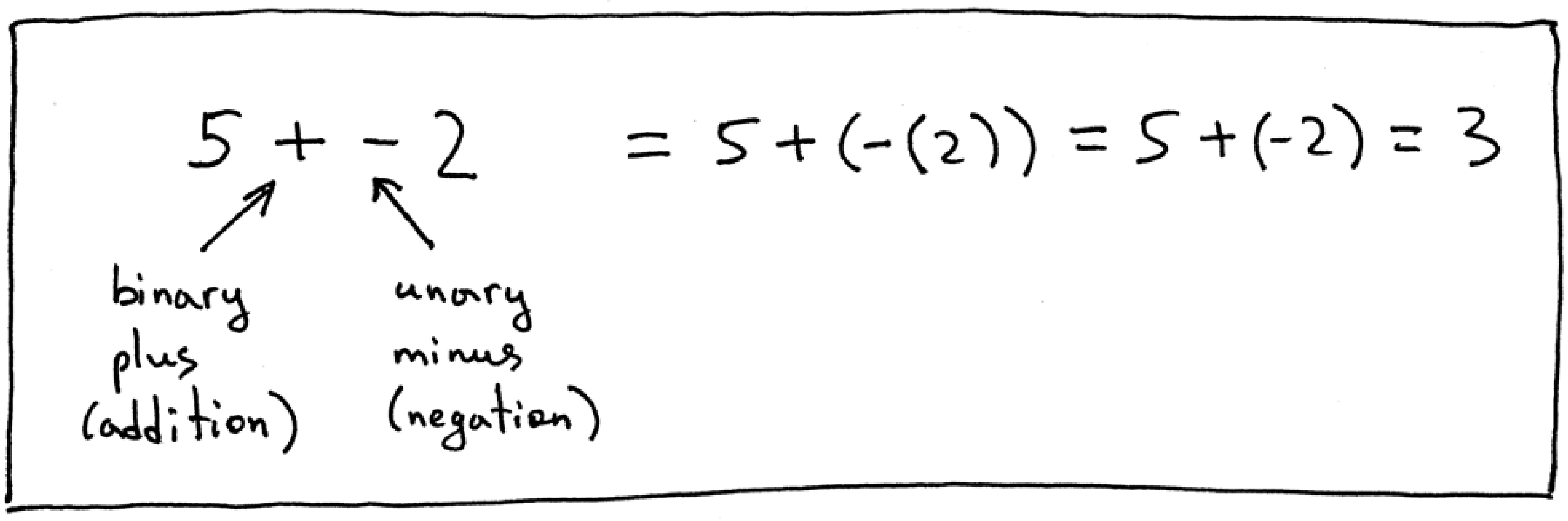


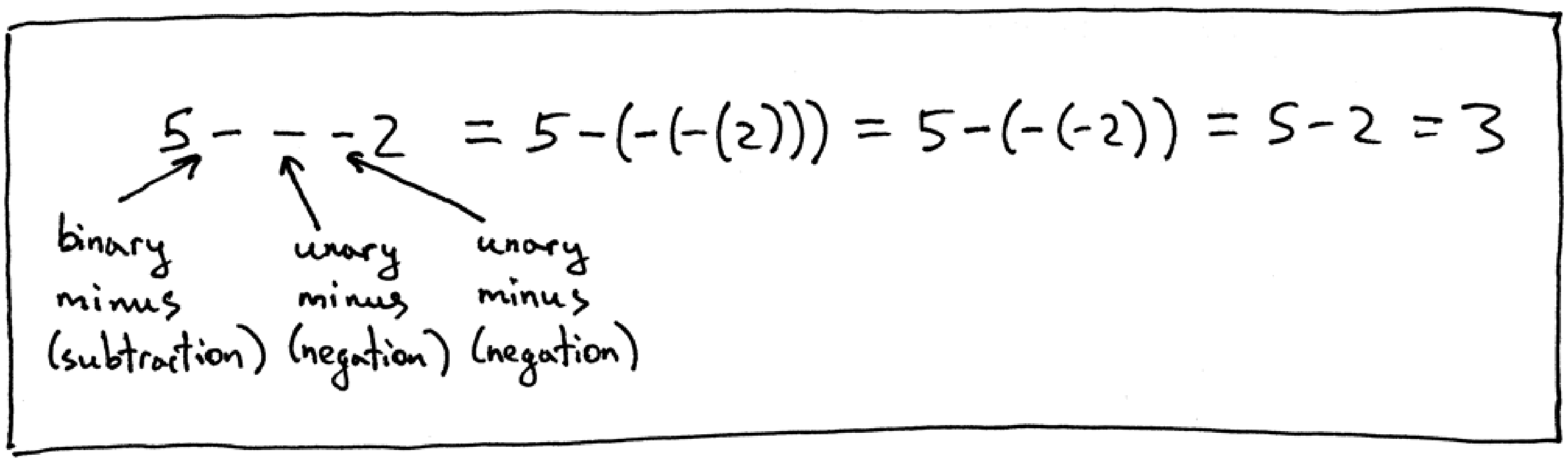
Let's take a look at another expression, "5 – – 2":



In the expression “5 – – 2” the first ‘–‘ represents the *binary* subtraction operation and the second ‘–‘ represents the *unary* minus operation, the negation.

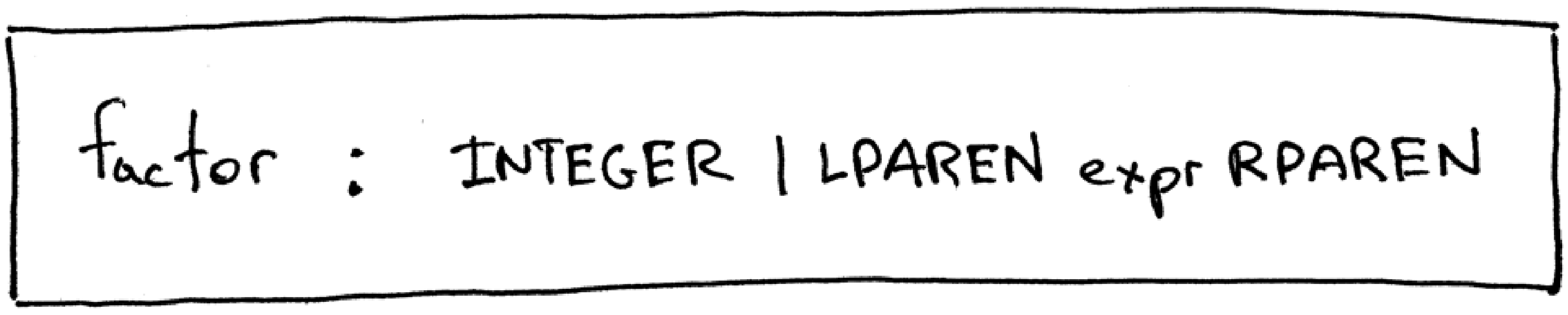
And some more examples:





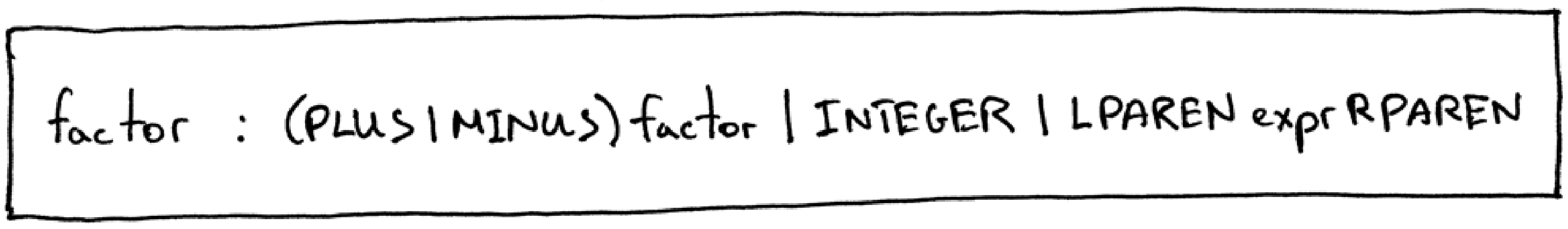
Now let’s update our grammar to include unary plus and unary minus operators. We’ll modify the *factor* rule and add unary operators there because unary operators have higher precedence than binary +, –, * and / operators.

This is our current *factor* rule:



```
factor : INTEGER | LPAREN expr RPAREN
```

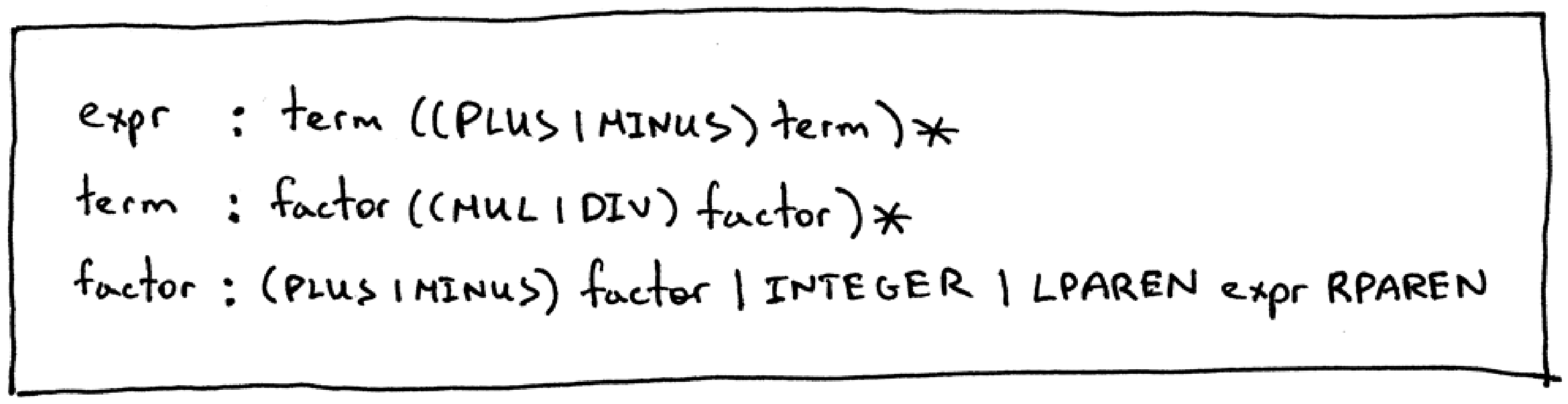
And this is our updated *factor* rule to handle unary plus and unary minus operators:



```
factor : (PLUS | MINUS) factor | INTEGER | LPAREN expr RPAREN
```

As you can see, I extended the *factor* rule to reference itself, which allows us to derive expressions like “– – – + – 3”, a legitimate expression with a lot of unary operators.

Here is the full grammar that can now derive expressions with unary plus and unary minus operators:



```
expr   : term ((PLUS | MINUS) term)*
term   : factor ((MUL | DIV) factor)*
factor : (PLUS | MINUS) factor | INTEGER | LPAREN expr RPAREN
```

The next step is to add an AST node class to represent unary operators.

This one will do:

```
class UnaryOp(AST):
    def __init__(self, op, expr):
        self.token = self.op = op
        self.expr = expr
```

The constructor takes two parameters: *op*, which represents the unary operator token (plus or minus) and *expr*, which represents an AST node.

Our updated grammar had changes to the *factor* rule, so that’s what we’re going to modify in our parser – the *factor* method. We will add code to the method to handle the “(PLUS | MINUS) factor” sub-rule:

```
def factor(self):
    """factor : (PLUS | MINUS) factor | INTEGER | LPAREN expr RPAREN"""
    token = self.current_token
    if token.type == PLUS:
        self.eat(PLUS)
        node = UnaryOp(token, self.factor())
        return node
    elif token.type == MINUS:
        self.eat(MINUS)
        node = UnaryOp(token, self.factor())
        return node
    elif token.type == INTEGER:
        self.eat(INTEGER)
        return Num(token)
    elif token.type == LPAREN:
        self.eat(LPAREN)
        node = self.expr()
        self.eat(RPAREN)
        return node
```

And now we need to extend the *Interpreter* class and add a *visit_UnaryOp* method to interpret unary nodes:

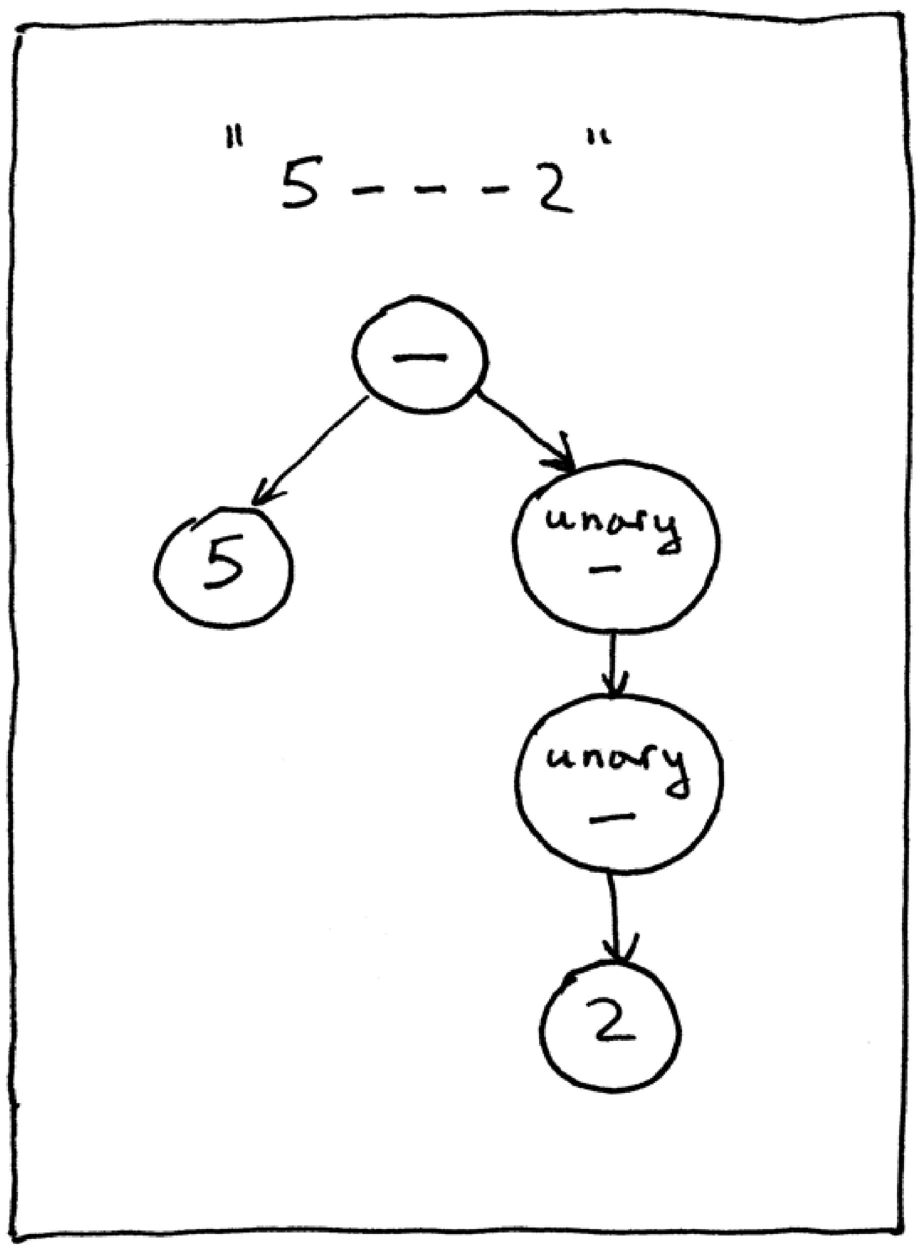
```
def visit_UnaryOp(self, node):
    op = node.op.type
    if op == PLUS:
        return +self.visit(node.expr)
    elif op == MINUS:
        return -self.visit(node.expr)
```

Onward!

Let’s manually build an AST for the expression “5 – – – 2” and pass it to our interpreter to verify that the new *visit_UnaryOp* method works. Here is how you can do it from the Python shell:

```
>>> from spi import BinOp, UnaryOp, Num, MINUS, INTEGER, Token
>>> five_tok = Token(INTEGER, 5)
>>> two_tok = Token(INTEGER, 2)
>>> minus_tok = Token(MINUS, '-')
>>> expr_node = BinOp(
...     Num(five_tok),
...     minus_tok,
...     UnaryOp(minus_token, UnaryOp(minus_token, Num(two_tok)))
... )
>>> from spi import Interpreter
>>> inter = Interpreter(None)
>>> inter.visit(expr_node)
3
```

Visually the above AST tree looks like this:



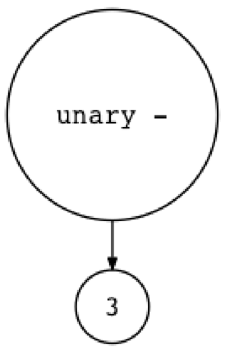
Download the full source code of the interpreter for this article directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part8/python/spi.py). Try it out and see for yourself that your updated tree-based interpreter properly evaluates arithmetic expressions containing unary operators.

Here is a sample session:

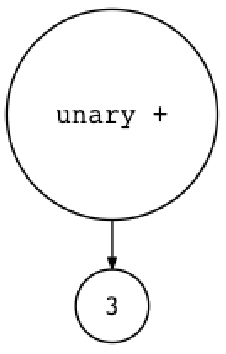
```
$ python spi.py
spi> - 3
-3
spi> + 3
3
spi> 5 - - - + - 3
8
spi> 5 - - - + - (3 + 4) - +2
10
```

I also updated the genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part8/python/genastdot.py) utility to handle unary operators. Here are some of the examples of the generated AST images for expressions with unary operators:

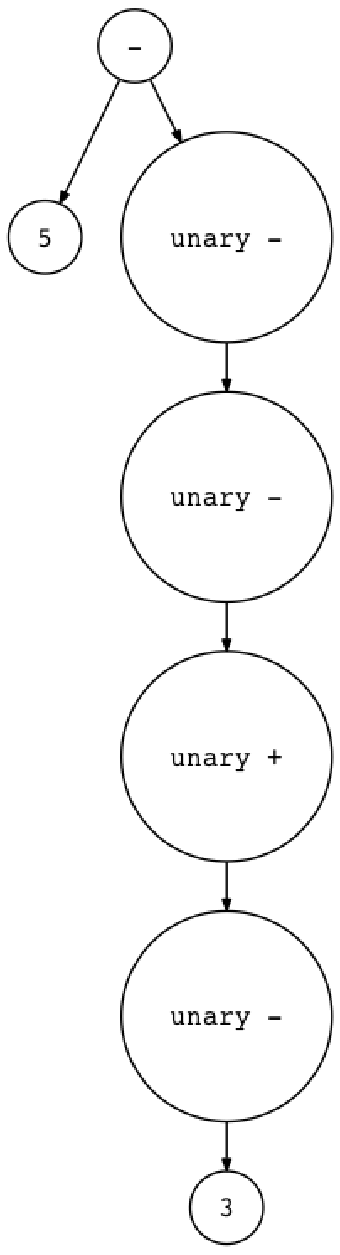
```
$ python genastdot.py "- 3" > ast.dot && dot -Tpng -o ast.png ast.dot
```



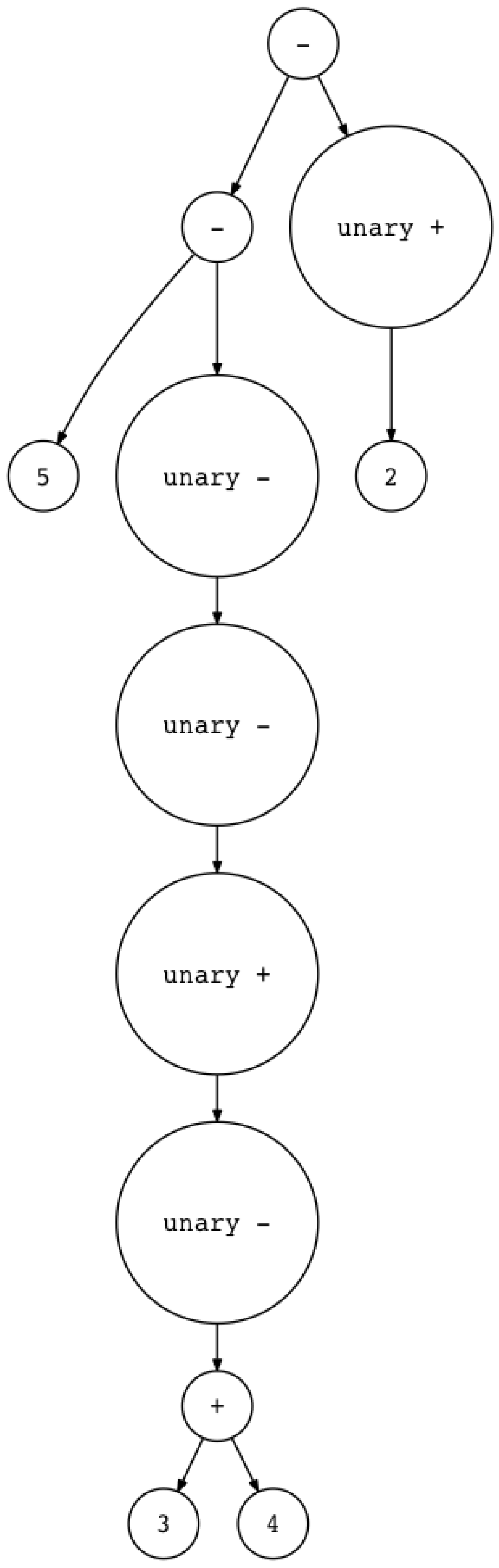
```
$ python genastdot.py "+ 3" > ast.dot && dot -Tpng -o ast.png ast.dot
```



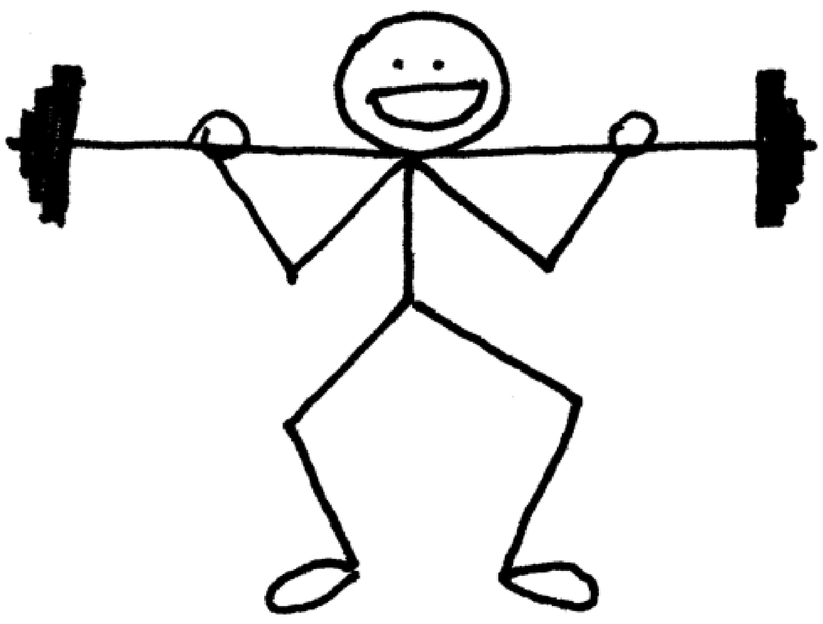
```
$ python genastdot.py "5 - - - + - 3" > ast.dot && dot -Tpng -o ast.png ast.dot
```



```
$ python genastdot.py "5 - - - + - (3 + 4) - +2" \
  > ast.dot && dot -Tpng -o ast.png ast.dot
```



And here is a new exercise for you:



- Install Free Pascal (http://www.freepascal.org/), compile and run testunary.pas (https://github.com/rspivak/lsbasi/blob/master/part8/python/testunary.pas), and verify that the results are the same as produced with your spi (https://github.com/rspivak/lsbasi/blob/master/part8/python/spi.py) interpreter.

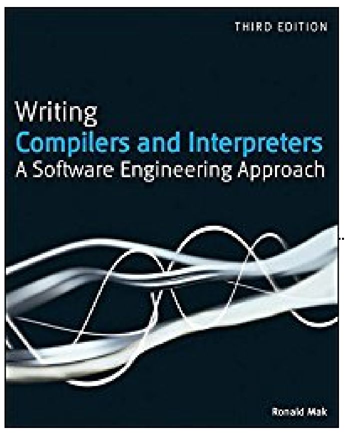
That’s all for today. In the next article, we’ll tackle assignment statements. Stay tuned and see you soon.

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)

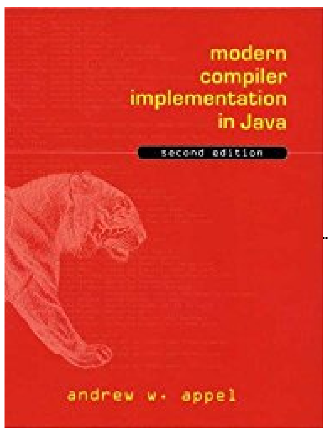
   (http://www.amazon.com/gp/product/B00QMJQHYG/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00QMJQHYG&linkCode=as2&tag=russblo0b-20&linkId=I53DN2FPOSCOLBXA)

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)

   (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=FYAZBCVOB66PGR6J)

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)

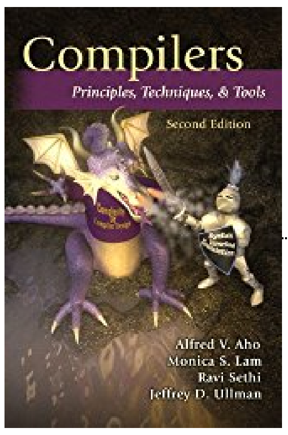
   

   (http://www.amazon.com/gp/product/052182060X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=052182060X&linkCode=as2&tag=russblo0b-20&linkId=GPMSWTZYFC2M6MJE)

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)

   Dick Grune · Kees van Reeuwijk
   Henri E. Bal · Ceriel J.H. Jacobs
   Koen Langendoen
   Modern Compiler
   Design
   Second Edition
   Springer
   (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=DZVYHZHDHYAPOQOD)

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)



(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=MD7L2CQHFXDYKOG6)

By the way, I'm writing a book **"Let's Build A Web Server: First Steps"** that explains how to write a basic web server from scratch. You can get a feel for the book here (http://ruslanspivak.com/lsbaws-part1/), here (http://ruslanspivak.com/lsbaws-part2/), and here (http://ruslanspivak.com/lsbaws-part3/). Subscribe to the mailing list to get the latest updates about the book and the release date.

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. (/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. (/lsbasi-part13/)
- Let's Build A Simple Interpreter. Part 14. (/lsbasi-part14/)

Comments

14 Comments Ruslan's Blog 1 Login

Recommend 5 Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

danialgoodwin · 2 years ago
Great series! Thank you very much for taking the time to write this. I highly recommend it!
5 ^ | v · Reply · Share ›

AJ · a year ago
Thank you for your work.

There is a mistake in typing "Let's manually build an AST for the expression "5 - - - 2" ".
You need to correct the following:

UnaryOp(minus_token, UnaryOp(minus_token, Num(two_tok)))

to:

UnaryOp(minus_tok, UnaryOp(minus_tok, Num(two_tok)))

I hope this help.

2 ^ | v • Reply • Share ›

Edward Lee • 3 months ago
Hi, thank you for the blog, really enjoying it. Maybe a fine point I didn't get, where you go:

class UnaryOp(AST):
def __init__(self, op, expr):
self.token = self.op = op
self.expr = expr

Wasn't clear about the decision to use of the name expr and self.expr as opposed to factor, self.factor?

And when using node.op.type, is it to be explicit that this token is of op type? Or is there more to it?
^ | v • Reply • Share ›

Christian Alexander (TheNewCom • a year ago
I've been implementing this in Kotlin for a low level esolang of mine, but I seem to be stuck on how to parse a ternary operator. Would I create a new grammar rule for this, or append it to an existing grammar rule? If the latter, which one?
^ | v • Reply • Share ›

Mamoon Ahmed → Christian Alexander (TheNewCom • 7 months ago
same rule but make a function that is responsible for parsing an expression only, for eg
on encountering '?'
parse expression until u encounter token ':'
parse expression again.... hope u get the idea ....
^ | v • Reply • Share ›

Marko Mackic • a year ago
Thanks for these awsome tutorials :)
^ | v • Reply • Share ›

Sfonxs • 2 years ago
I found these really interesting, thanks!
Do you have plans on continuing this serie?
^ | v • Reply • Share ›

rspivak Mod → Sfonxs • 2 years ago
You're welcome and thanks for reading!
The next article is already in the making and I'll publish it soon.
^ | v • Reply • Share ›

sfifs → rspivak • 2 years ago
Great to hear! I've been checking every week :-)

I've been following your tutorial in golang and for someone without formal theory background, your series has been the simplest & clearest explanation of compilers & interpreters I've seen.
^ | v • Reply • Share ›

Colin Wang • 2 years ago
Nice! It's very usefull for me.
^ | v • Reply • Share ›

Qiang Li • 2 years ago
I guess it is not that necessary to create a new UnaryOp class, since you can simply assign 0 to left for BinaryOp to act like an UnaryOp, BTW i implemented a javascript version(https://github.com/Drbelfas..., which uses regular expression to help tokenize in case somebody may find help
^ | v • Reply • Share ›

慕容小明 • 2 years ago
Thanks for your sharing. I can't wait for the next part ^_^
^ | v • Reply • Share ›

anula • 2 years ago
Is function `eat(Token)` somehow "traditional" for writing parser? And I mean in this form, where it takes Token and fails when the token is not equal to the current one.
In Python implementation it makes sense in some places, but in Rust it is just redundant, since you anyway use pattern matching to determine what exact token you need to pass there, and panic if the token is not as expected. Wouldn't just `advance_current_token()` (without token as argument) be better? You anyway never get to the panic inside eat.

BTW. here is my code (Rust and Python, though the Python one is almost identical to the one from posts): https://bitbucket.org/anula...
^ | v • Reply • Share ›

anula • 2 years ago
Wow, that is a really nice series. Can't wait for the next part ;)Thank you, for all the work you put in this.
^ | v • Reply • Share ›

Social

github (https://github.com/rspivak/)