Let’s Build A Simple Interpreter. Part 15. (https://ruslanspivak.com/lsbasi-part15/)

Date Fri, June 21, 2019

> *“I am a slow walker, but I never walk back.” — Abraham Lincoln*

And we’re back to our regularly scheduled programming! :)

Before moving on to topics of recognizing and interpreting procedure calls, let’s make some changes to improve our error reporting a bit. Up until now, if there was a problem getting a new token from text, parsing source code, or doing semantic analysis, a stack trace would be thrown right into your face with a very generic message. We can do better than that.

To provide better error messages pinpointing where in the code an issue happened, we need to add some features to our interpreter. Let’s do that and make some other changes along the way. This will make the interpreter more user friendly and give us an opportunity to flex our muscles after a “short” break in the series. It will also give us a chance to prepare for new features that we will be adding in future articles.

Goals for today:

- Improve error reporting in the lexer, parser, and semantic analyzer. Instead of stack traces with very generic messages like *“Invalid syntax”*, we would like to see something more useful like *“SyntaxError: Unexpected token -> Token(TokenType.SEMI, ‘;’, position=23:13)”*
- Add a “—scope” command line option to turn scope output on/off
- Switch to Python 3. From here on out, all code will be tested on Python 3.7+ only

Let’s get cracking and start flexing our coding muscles by changing our lexer first.

Here is a list of the changes we are going to make in our lexer today:

1. We will add error codes and custom exceptions: *LexerError*, *ParserError*, and *SemanticError*
2. We will add new members to the *Lexer* class to help to track tokens' positions: *lineno* and *column*
3. We will modify the *advance* method to update the lexer's *lineno* and *column* variables
4. We will update the *error* method to raise a *LexerError* exception with information about the current line and column
5. We will define token types in the *TokenType* enumeration class (Support for enumerations was added in Python 3.4)
6. We will add code to automatically create reserved keywords from the *TokenType* enumeration members
7. We will add new members to the *Token* class: *lineno* and *column* to keep track of the token's line number and column number, correspondingly, in the text
8. We will refactor the *get_next_token* method code to make it shorter and have a generic code that handles single-character tokens

1. Let's define some error codes first. These codes will be used by our parser and semantic analyzer. Let's also define the following error classes: *LexerError*, *ParserError*, and *SemanticError* for lexical, syntactic, and, correspondingly, semantic errors:

```
from enum import Enum

class ErrorCode(Enum):
    UNEXPECTED_TOKEN = 'Unexpected token'
    ID_NOT_FOUND     = 'Identifier not found'
    DUPLICATE_ID     = 'Duplicate id found'

class Error(Exception):
    def __init__(self, error_code=None, token=None, message=None):
        self.error_code = error_code
        self.token = token
        # add exception class name before the message
        self.message = f'{self.__class__.__name__}: {message}'

class LexerError(Error):
    pass

class ParserError(Error):
    pass

class SemanticError(Error):
    pass
```

ErrorCode is an enumeration class, where each member has a name and a value:

```
>>> from enum import Enum
>>>
>>> class ErrorCode(Enum):
...     UNEXPECTED_TOKEN = 'Unexpected token'
...     ID_NOT_FOUND     = 'Identifier not found'
...     DUPLICATE_ID     = 'Duplicate id found'
...
>>> ErrorCode
<enum 'ErrorCode'>
>>>
>>> ErrorCode.ID_NOT_FOUND
<ErrorCode.ID_NOT_FOUND: 'Identifier not found'>
```

The *Error* base class constructor takes three arguments:

- *error_code*: ErrorCode.ID_NOT_FOUND, etc
- *token*: an instance of the *Token* class

- *message*: a message with more detailed information about the problem

As I've mentioned before, *LexerError* is used to indicate an error encountered in the lexer, *ParserError* is for syntax related errors during the parsing phase, and *SemanticError* is for semantic errors.

2. To provide better error messages, we want to display the position in the source text where the problem happened. To be able do that, we need to start tracking the current line number and column in our lexer as we generate tokens. Let's add *lineno* and *column* fields to the *Lexer* class:

```
class Lexer(object):
    def __init__(self, text):
        ...
        # self.pos is an index into self.text
        self.pos = 0
        self.current_char = self.text[self.pos]
        # token line number and column number
        self.lineno = 1
        self.column = 1
```

3. The next change that we need to make is to reset *lineno* and *column* in the *advance* method when encountering a new line and also increase the *column* value on each advance of the *self.pos* pointer:

```
def advance(self):
    """Advance the `pos` pointer and set the `current_char` variable."""
    if self.current_char == '\n':
        self.lineno += 1
        self.column = 0

    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]
        self.column += 1
```

With those changes in place, every time we create a token we will pass the current *lineno* and *column* from the lexer to the newly created token.

4. Let's update the *error* method to throw a *LexerError* exception with a more detailed error message telling us the current character that the lexer choked on and its location in the text.

```
def error(self):
    s = "Lexer error on '{lexeme}' line: {lineno} column: {column}".format(
        lexeme=self.current_char,
        lineno=self.lineno,
        column=self.column,
    )
    raise LexerError(message=s)
```

5. Instead of having token types defined as module level variables, we are going to move them into a dedicated enumeration class called *TokenType*. This will help us simplify certain operations and make some parts of our code a bit shorter.

Old style:

```
# Token types
PLUS  = 'PLUS'
MINUS = 'MINUS'
MUL   = 'MUL'
...
```

New style:

```
class TokenType(Enum):
    # single-character token types
    PLUS          = '+'
    MINUS         = '-'
    MUL           = '*'
    FLOAT_DIV     = '/'
    LPAREN        = '('
    RPAREN        = ')'
    SEMI          = ';'
    DOT           = '.'
    COLON         = ':'
    COMMA         = ','
    # block of reserved words
    PROGRAM       = 'PROGRAM'  # marks the beginning of the block
    INTEGER       = 'INTEGER'
    REAL          = 'REAL'
    INTEGER_DIV   = 'DIV'
    VAR           = 'VAR'
    PROCEDURE     = 'PROCEDURE'
    BEGIN         = 'BEGIN'
    END           = 'END'      # marks the end of the block
    # misc
    ID            = 'ID'
    INTEGER_CONST = 'INTEGER_CONST'
    REAL_CONST    = 'REAL_CONST'
    ASSIGN        = ':='
    EOF           = 'EOF'
```

6. We used to manually add items to the *RESERVED_KEYWORDS* dictionary whenever we had to add a new token type that was also a reserved keyword. If we wanted to add a new STRING token type, we would have to

- (a) create a new module level variable STRING = 'STRING'
- (b) manually add it to the *RESERVED_KEYWORDS* dictionary

Now that we have the *TokenType* enumeration class, we can remove the manual step **(b)** above and keep token types in one place only. This is the "two is too many (https://www.codesimplicity.com/post/two-is-too-many/)" rule in action - going forward, the only change you need to make to add a new keyword token type is to put the keyword between PROGRAM and END in the *TokenType* enumeration class, and the *_build_reserved_keywords* function will take care of the rest:

```
def _build_reserved_keywords():
    """Build a dictionary of reserved keywords.

    The function relies on the fact that in the TokenType
    enumeration the beginning of the block of reserved keywords is
    marked with PROGRAM and the end of the block is marked with
    the END keyword.

    Result:
        {'PROGRAM': <TokenType.PROGRAM: 'PROGRAM'>,
         'INTEGER': <TokenType.INTEGER: 'INTEGER'>,
         'REAL': <TokenType.REAL: 'REAL'>,
         'DIV': <TokenType.INTEGER_DIV: 'DIV'>,
         'VAR': <TokenType.VAR: 'VAR'>,
         'PROCEDURE': <TokenType.PROCEDURE: 'PROCEDURE'>,
         'BEGIN': <TokenType.BEGIN: 'BEGIN'>,
         'END': <TokenType.END: 'END'>}
    """
    # enumerations support iteration, in definition order
    tt_list = list(TokenType)
    start_index = tt_list.index(TokenType.PROGRAM)
    end_index = tt_list.index(TokenType.END)
    reserved_keywords = {
        token_type.value: token_type
        for token_type in tt_list[start_index:end_index + 1]
    }
    return reserved_keywords


RESERVED_KEYWORDS = _build_reserved_keywords()
```

As you can see from the function's documentation string, the function relies on the fact that a block of reserved keywords in the *TokenType* enum is marked by PROGRAM and END keywords.

The function first turns *TokenType* into a list (the definition order is preserved), and then it gets the starting index of the block (marked by the PROGRAM keyword) and the end index of the block (marked by the END keyword). Next, it uses dictionary comprehension to build a dictionary where the keys are string values of the enum members and the values are the *TokenType* members themselves.

```
>>> from spi import _build_reserved_keywords
>>> from pprint import pprint
>>> pprint(_build_reserved_keywords())  # 'pprint' sorts the keys
{'BEGIN': <TokenType.BEGIN: 'BEGIN'>,
 'DIV': <TokenType.INTEGER_DIV: 'DIV'>,
 'END': <TokenType.END: 'END'>,
 'INTEGER': <TokenType.INTEGER: 'INTEGER'>,
 'PROCEDURE': <TokenType.PROCEDURE: 'PROCEDURE'>,
 'PROGRAM': <TokenType.PROGRAM: 'PROGRAM'>,
 'REAL': <TokenType.REAL: 'REAL'>,
 'VAR': <TokenType.VAR: 'VAR'>}
```

7. The next change is to add new members to the *Token* class, namely *lineno* and *column,* to keep track of a token's line number and column number in a text

```
class Token(object):
    def __init__(self, type, value, lineno=None, column=None):
        self.type = type
        self.value = value
        self.lineno = lineno
        self.column = column

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

        Example:
            >>> Token(TokenType.INTEGER, 7, lineno=5, column=10)
            Token(TokenType.INTEGER, 7, position=5:10)
        """
        return 'Token({type}, {value}, position={lineno}:{column})'.format(
            type=self.type,
            value=repr(self.value),
            lineno=self.lineno,
            column=self.column,
        )

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

8. Now, onto *get_next_token* method changes. Thanks to enums, we can reduce the amount of code that deals with single character tokens by writing a generic code that generates single character tokens and doesn't need to change when we add a new single character token type:

Instead of a lot of code blocks like these:

```
if self.current_char == ';':
    self.advance()
    return Token(SEMI, ';')

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

if self.current_char == ',':
    self.advance()
    return Token(COMMA, ',')
...
```

We can now use this generic code to take care of all current and future single-character tokens

```
# single-character token
try:
    # get enum member by value, e.g.
    # TokenType(';') --> TokenType.SEMI
    token_type = TokenType(self.current_char)
except ValueError:
    # no enum member with value equal to self.current_char
    self.error()
else:
    # create a token with a single-character lexeme as its value
    token = Token(
        type=token_type,
        value=token_type.value,  # e.g. ';', '.', etc
        lineno=self.lineno,
        column=self.column,
    )
    self.advance()
    return token
```

Arguably it's less readable than a bunch of *if* blocks, but it's pretty straightforward once you understand what's going on here. Python enums allow us to access enum members by values and that's what we use in the code above. It works like this:

- First we try to get a *TokenType* member by the value of *self.current_char*
- If the operation throws a *ValueError* exception, that means we don't support that token type
- Otherwise we create a correct token with the corresponding token type and value.

This block of code will handle all current and new single character tokens. All we need to do to support a new token type is to add the new token type to the *TokenType* definition and that's it. The code above will stay unchanged.

The way I see it, it's a win-win situation with this generic code: we learned a bit more about Python enums, specifically how to access enumeration members by values; we wrote some generic code to handle all single character tokens, and, as a side effect, we reduced the amount of repetitive code to handle those single character tokens.

The next stop is parser changes.

Here is a list of changes we'll make in our parser today:

1. We will update the parser's *error* method to throw a *ParserError* exception with an error code and current token
2. We will update the *eat* method to call the modified *error* method
3. We will refactor the *declarations* method and move the code that parses a procedure declaration into a separate method.

1. Let's update the parser's *error* method to throw a *ParserError* exception with some useful information

```
def error(self, error_code, token):
    raise ParserError(
        error_code=error_code,
        token=token,
        message=f'{error_code.value} -> {token}',
    )
```

2. And now let's modify the *eat* method to call the updated *error* method

```
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(
            error_code=ErrorCode.UNEXPECTED_TOKEN,
            token=self.current_token,
        )
```

3. Next, let's update the *declaration*'s documentation string and move the code that parses a procedure declaration into a separate method, *procedure_declaration*:

```
def declarations(self):
    """
    declarations : (VAR (variable_declaration SEMI)+)? procedure_declaration*
    """
    declarations = []

    if self.current_token.type == TokenType.VAR:
        self.eat(TokenType.VAR)
        while self.current_token.type == TokenType.ID:
            var_decl = self.variable_declaration()
            declarations.extend(var_decl)
            self.eat(TokenType.SEMI)

    while self.current_token.type == TokenType.PROCEDURE:
        proc_decl = self.procedure_declaration()
        declarations.append(proc_decl)

    return declarations

def procedure_declaration(self):
    """procedure_declaration :
         PROCEDURE ID (LPAREN formal_parameter_list RPAREN)? SEMI block SEMI
    """
    self.eat(TokenType.PROCEDURE)
    proc_name = self.current_token.value
    self.eat(TokenType.ID)
    params = []

    if self.current_token.type == TokenType.LPAREN:
        self.eat(TokenType.LPAREN)
        params = self.formal_parameter_list()
        self.eat(TokenType.RPAREN)

    self.eat(TokenType.SEMI)
    block_node = self.block()
    proc_decl = ProcedureDecl(proc_name, params, block_node)
    self.eat(TokenType.SEMI)
    return proc_decl
```

These are all the changes in the parser. Now, we'll move onto the semantic analyzer.

And finally here is a list of changes we'll make in our semantic analyzer:

1. We will add a new *error* method to the *SemanticAnalyzer* class to throw a *SemanticError* exception with some additional information

2. We will update *visit_VarDecl* to signal an error by calling the *error* method with a relevant error code and token
3. We will also update *visit_Var* to signal an error by calling the *error* method with a relevant error code and token
4. We will add a *log* method to both the *ScopedSymbolTable* and *SemanticAnalyzer*, and replace all *print* statements with calls to *self.log* in the corresponding classes
5. We will add a command line option "—-scope" to turn scope logging on and off (it will be off by default) to control how "noisy" we want our interpreter to be
6. We will add empty *visit_Num* and *visit_UnaryOp* methods

1. First things first. Let's add the *error* method to throw a *SemanticError* exception with a corresponding error code, token and message:

```
def error(self, error_code, token):
    raise SemanticError(
        error_code=error_code,
        token=token,
        message=f'{error_code.value} -> {token}',
    )
```

2. Next, let's update *visit_VarDecl* to signal an error by calling the *error* method with a relevant error code and token

```
def visit_VarDecl(self, node):
    type_name = node.type_node.value
    type_symbol = self.current_scope.lookup(type_name)

    # We have all the information we need to create a variable symbol.
    # Create the symbol and insert it into the symbol table.
    var_name = node.var_node.value
    var_symbol = VarSymbol(var_name, type_symbol)

    # Signal an error if the table already has a symbol
    # with the same name
    if self.current_scope.lookup(var_name, current_scope_only=True):
        self.error(
            error_code=ErrorCode.DUPLICATE_ID,
            token=node.var_node.token,
        )

    self.current_scope.insert(var_symbol)
```

3. We also need to update the *visit_Var* method to signal an error by calling the *error* method with a relevant error code and token

```
def visit_Var(self, node):
    var_name = node.value
    var_symbol = self.current_scope.lookup(var_name)
    if var_symbol is None:
        self.error(error_code=ErrorCode.ID_NOT_FOUND, token=node.token)
```

Now semantic errors will be reported as follows:

```
SemanticError: Duplicate id found -> Token(TokenType.ID, 'a', position=21:4)
```

Or

```
SemanticError: Identifier not found -> Token(TokenType.ID, 'b', position=22:9)
```

4. Let's add the *log* method to both the *ScopedSymbolTable* and *SemanticAnalyzer*, and replace all *print* statements with calls to *self.log*:

```
def log(self, msg):
    if _SHOULD_LOG_SCOPE:
        print(msg)
```

As you can see, the message will be printed only if the global variable _SHOULD_LOG_SCOPE is set to true. The —*scope* command line option that we will add in the next step will control the value of the _SHOULD_LOG_SCOPE variable.

5. Now, let's update the *main* function and add a command line option "—scope" to turn scope logging on and off (it's off by default)

```
parser = argparse.ArgumentParser(
    description='SPI - Simple Pascal Interpreter'
)
parser.add_argument('inputfile', help='Pascal source file')
parser.add_argument(
    '--scope',
    help='Print scope information',
    action='store_true',
)
args = parser.parse_args()
global _SHOULD_LOG_SCOPE
_SHOULD_LOG_SCOPE = args.scope
```

Here is an example with the switch on:

```
$ python spi.py idnotfound.pas --scope
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: INTEGER. (Scope name: global)
Lookup: a. (Scope name: global)
Insert: a
Lookup: b. (Scope name: global)
SemanticError: Identifier not found -> Token(TokenType.ID, 'b', position=6:9)
```

And with scope logging off (default):

```
$ python spi.py idnotfound.pas
SemanticError: Identifier not found -> Token(TokenType.ID, 'b', position=6:9)
```

6. Add empty *visit_Num* and *visit_UnaryOp* methods

```
def visit_Num(self, node):
    pass

def visit_UnaryOp(self, node):
    pass
```

These are all the changes to our semantic analyzer for now.

See GitHub (https://github.com/rspivak/lsbasi/tree/master/part15/) for Pascal files with different errors to try your updated interpreter on and see what error messages the interpreter generates.