Javascript Lexer and Parser with python

Arghyadeep Giri [argiri@ucsc.edu]
Jiahua You [jiyou@ucsc.edu]

June 2018

1 Introduction

This project is primarily to build a Javascript Lexer and a hence a Parsing tool using a python library named PLY. It is a pure-Python implementation of the popular compiler construction tools lex and yacc. PLY consists of two separate modules; lex.py and yacc.py, both of which are found in a Python package called ply. The lex.py module is used to break input text into a collection of tokens specified by a collection of regular expression rules. yacc.py is used to recognize language syntax that has been specified in the form of a context free grammar. The challenges faced in this project is mostly taking care of different types of tokens with regular expressions. Also, parsing while handling error has been a challenge. The lexer covers almost entirity of javascript where as the parser covers a broad section. The output result creates a Block like structure which could be fed into an interpreter with symbol table for further evaluation.

2 The Lexer

The Lexer is a part of the interpreter that tokenizes the entire input code into atomic tokens. Also, along with these tokens the lexer determines the type of the token and the position of the token in the program. An example of token is $\langle token\ name, token\ type, line\ number, starting\ character\ number \rangle$

2.1 The tokens list

The lexer has been provided a list tokens that defines all of the possible token names that can be produced by the lexer. This list is always required and is used to perform a variety of validation checks. The tokens list is also used by the parser module to identify terminals.The entire list of tokens can be broken into identifiers, keywords, future keywords, punctuators and literal keywords.

2.2 Specification of tokens

Each token is specified by writing a regular expression rule compatible with Python's re module. Each of these rules are defined by making declarations with a special prefix t_ to indicate that it defines a token.For simple tokens, the regular expression can be specified as strings. Example: t_ AND = r'\+'
If some kind of action needs to be performed, a token rule can be specified as a function. For example, this rule matches numbers and converts the string into a Python integer.

```
def t_NUMBER(t):
        r'\d+'
        t.value = int(t.value)
        return t
```

When a function is used, the regular expression rule is specified in the function documentation string. The function always takes a single argument which is an instance of LexToken. This object has attributes of t.type which is the token type (as a string), t.value which is the lexeme (the actual text matched), t.lineno which is the current line number, and t.lexpos which is the position of the token relative to the beginning of the input text. By default, t.type is set to the name following the t _ prefix. The action function modifies the contents of the LexToken object as appropriate.

2.3 Discarded Tokens

The discarded tokens are the multi-line and the single line comments. Also we needed to get rid of the white spaces and tabs. To discard a token, such as a comment, we simply defined a token rule that returns no value.

```
def t_single_line_comment(t):
    r'∧//. ∗′
    pass
```

Alternatively, for white apaces we included the prefix "ignore_" in the token declaration to force a token to be ignored.

```
t_ignore = ' '
```

2.4 Token values

When tokens are returned by lex, they have a value that is stored in the value attribute. Normally, the value is the text that was matched. However, the value can be assigned to any Python object. For instance, when lexing identifiers, we may want to return both the identifier name and information from some sort of

symbol table. For example:

```
def t_ID(t):
    ...
     Look up symbol table information and return a tuple
    t.value = (t.value, symbol_lookup(t.value))
    ...
    return  t
```

2.5 Line number and positional information

By default, lex.py knows nothing about line numbers. This is because lex.py doesn't know anything about what constitutes a "line" of input (e.g., the newline character or even if the input is textual data). To update this information, we constructed a special rule. In the example, the t_newline() rule shows the implementation.

```
def t_newline(t):
        r'+'
        t.lexer.lineno += t.value.count("")
```

2.6 Literal characters

A literal character is simply a single character that is returned "as is" when encountered by the lexer. Literals are checked after all of the defined regular expression rules. Thus, if a rule starts with one of the literal characters, it will always take precedence. When a literal token is returned, both its type and value attributes are set to the character itself. For example, '+'.

It's possible to write token functions that perform additional actions when literals are matched. However, setting the token type appropriately is important. For example:

```
literals = [ '', '' ]

def t_lbrace(t):
    r'\{'
    t.type = '{' # Set token type to the expected literal
    return t

def t_rbrace(t):
    r'\}'
    t.type = '' # Set token type to the expected literal
    return t
```

2.7 Error Handling

The t_error() function is used to handle lexing errors that occur when illegal characters are detected. In this case, the t.value attribute contains the rest of the input string that has not been tokenized. The error function is defined as follows:

```
    # Error handling rule
def t_error(t):
        print("Illegal character '
        t.lexer.skip(1)
```

In this case, we simply print the offending character and skip ahead one character by calling t.lexer.skip(1).

3 The Parser

The parser takes the output of the lexer as its input and creates an abstract syntax tree(AST) out of it, which is further used by the interpreter for evaluation of a code. Yacc.py is used to parse language syntax. The syntax is usually specified in terms of a BNF grammar. For example, if we wanted to parse simple arithmetic expressions, we might first write an unambiguous grammar specification like this:

```
expression : expression + term
           — expression - term
           — term

term : term * factor
           — term / factor
           — factor

factor : NUMBER
           — ( expression )
```

In the grammar, symbols such as NUMBER, +, -, *, and / are known as terminals and correspond to raw input tokens. Identifiers such as term and factor refer to grammar rules comprised of a collection of terminals and other rules. These identifiers are known as non-terminals.

3.1 YACC

The ply.yacc module implements the parsing component of PLY. The name "yacc" stands for "Yet Another Compiler Compiler".Example of making a grammar for simple arithmetic expression.

```
def p_expression_plus(p):
        'expression : expression PLUS term'
        p[0] = p[1] + p[3]
```

each grammar rule is defined by a Python function where the doc string to that function contains the appropriate context-free grammar specification. The statements that make up the function body implement the semantic actions of the rule. Each function accepts a single argument p that is a sequence containing the values of each grammar symbol in the corresponding rule.For tokens, the "value" of the corresponding p[i] is the same as the p.value attribute assigned in the lexer module. For non-terminals, the value is determined by whatever is placed in p[0] when rules are reduced. This value can be anything at all.

3.2 Combining grammar rule functions

When grammar rules are similar, they can be combined into a single function. For example the functions for 'plus' and 'minus' can be combined t get one function.

```
def p_expression(p):
        "'expression : expression PLUS term — expression MINUS term"'
        if p[2] == '+':
                p[0] = p[1] + p[3]
        elif p[2] == '-':
                p[0] = p[1] - p[3]
```

When combining grammar rules into a single function, it is usually a good idea for all of the rules to have a similar structure (e.g., the same number of terms). Otherwise, the corresponding action code may be more complicated than necessary. If parsing performance is a concern, we should resist the urge to put too much conditional processing into a single grammar rule as shown in these examples. When we add checks to see which grammar rule is being handled, we are actually duplicating the work that the parser has already performed (i.e., the parser already knows exactly what rule it matched). We eliminated this overhead by using a separate p_rule() function for each grammar rule.

3.3 Character literals

a grammar may contain tokens defined as single character literals. For example:

```
def p_binary_operators(p):
        "'expression : expression '+' term — expression '-' term term : term '*'
factor — term '/' factor"'
        if p[2] == '+':
                p[0] = p[1] + p[3]
        elif p[2] == '-':
                p[0] = p[1] - p[3]
```

```
    elif p[2] == '*':
            p[0] = p[1] * p[3]
    elif p[2] == '/':
            p[0] = p[1] / p[3]
```

A character literal must be enclosed in quotes such as '+'. In addition, if literals are used, they must be declared in the corresponding lex file through the use of a special literals declaration.

3.4 Empty productions

A rule is said to be empty if its right-hand side (components) is empty. It means that result can match the empty string. yacc.py can handle empty productions by defining a rule like this:

```
def p_empty(p):
        'empty :'
        pass
```

Now to use the empty production, simply we use 'empty' as a symbol.

```
    def p_optitem(p):
        'optitem : item'
'           | empty'
        ...
```

3.5 Dealing with ambiguous grammar

The expression grammar given in the earlier example has been written in a special format to eliminate ambiguity. However, in many situations, it is extremely difficult or awkward to write grammars in this format. A much more natural way to express the grammar is in a more compact form like this:

```
expression : expression PLUS expression
        | expression MINUS expression
        | expression TIMES expression
        | expression DIVIDE expression
        | LPAREN expression RPAREN
        | NUMBER
```

Unfortunately, this grammar specification is ambiguous. For example, if we are parsing the string "3 * 4 + 5", there is no way to tell how the operators are supposed to be grouped. To resolve ambiguity, especially in expression grammars, we add a variable precedence to the grammar file like this:

precedence = (('left', 'PLUS', 'MINUS'), ('left', 'TIMES', 'DIVIDE'),) This declaration specifies that PLUS/MINUS have the same precedence level and are left-associative and that TIMES/DIVIDE have the same precedence and are left-associative. Within the precedence declaration, tokens are ordered from lowest to highest precedence. Thus, this declaration specifies that TIMES/DIVIDE

have higher precedence than PLUS/MINUS (since they appear later in the precedence specification).

4 Future Works

Future works include completion of the parser for handling sub cases in javascript code. Implementing switch statement and a for iteration statements would be the first priority. Once the parser is completely built the next target would be to build an interpreter. The chalenges would be Scopes and scoped symbol tables, Procedure declarations with formal parameters, Procedure symbols, Nested scopes, Scope tree: Chaining scoped symbol tables, Nested scopes and name resolution.

5 Conclusion

Implementing lexer and parser with ply was a challenging task since understanding how the modules work led to creating the lexer and the parser. Initially the attempt was to build a handmade lexer and a parser which was beyond the scope of this project. We tried implementing functions that removed white spaces, new lines and comments and also tokenized without the use of regular expressions. Using PLY we realized that using regular expressions for such task makes it a lot more easier and convenient. Handling errors and regular expressions also becomes easier and efficient this way.

References

[1] Aho, A.V., Sethi, R. and Ullman ,J.D. (1986) " Compilers: principles, techniques, and tools." Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[2] Frost, R., Hafiz, R. and Callaghan, P. (2008) " Parser Combinators for Ambiguous Left-Recursive Grammars." 10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN , Volume 4902/2008, Pages: 167 - 181, January 2008, San Francisco.

[3] Frost, R., Hafiz, R. and Callaghan, P. (2007) " Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars ." 10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE , Pages: 109 - 120, June 2007, Prague.

[4] http://www.drdobbs.com/web-development/prototyping-interpreters-using-python-le/184405580.

[5] https://github.com/PiotrDabkowski/pyjsparser/blob/master/pyjsparser/parser.py.

[6] https://github.com/jtolds/pynarcissus/blob/master/jsparser3.py

[7] https://github.com/rspivak/lsbasi/blob/master/part14/spi.py