Sequoia

Simple Extension for QUery-Oriented Integrated Analytics

Harris Georgiou (MSc,PhD), Data Science Lab,
University of Piraeus, Greece

@ FOSSCOMM 2020 (virtual)

Overview

1. Problem definition

What is **Sequoia**? Why do we need something like this?

2. Basic concept

How is it structured? What tools does it use? How does it work?

Being able to make sense of Big data in due time is the biggest challenge for Data Analytics and Machine Learning today.

“1.7MB of data is created every second by every person during 2020.”

“Over 2.5 quintillion (10^{18}) bytes of data are created every single day.”

The data “binding” problem

Data sources are diverse

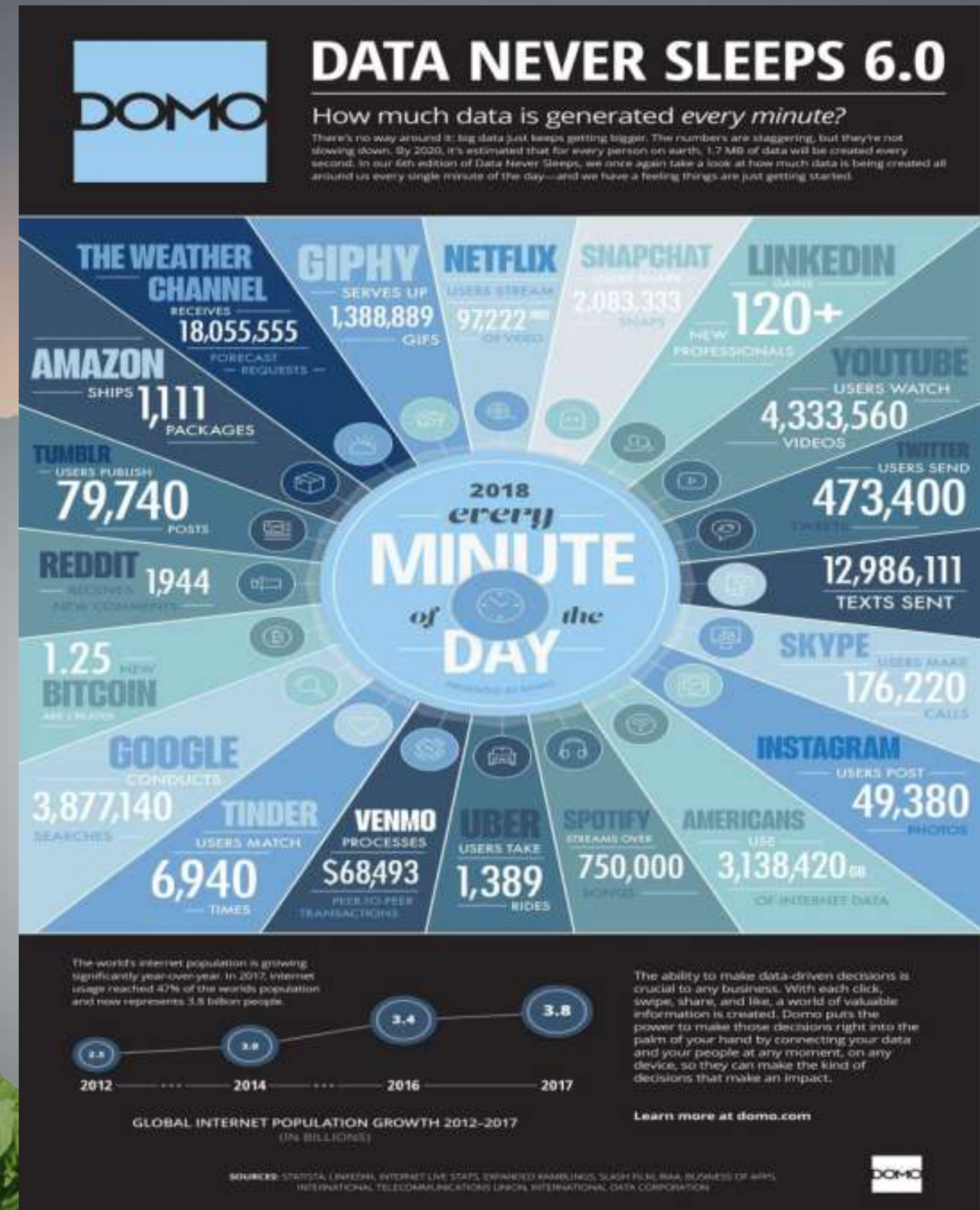
They come from various sensors, databases, modalities, domains, legacy archives.

Data themselves are diverse

May be tabular (.csv), XML/JSON, raw SQL results, images, 3D/4D medical data, time series.

Data processing pipeline is diverse

Missing values? Errors? Noise? Ranges? Rescaling? Filtering? Classification? Regression? Clustering?

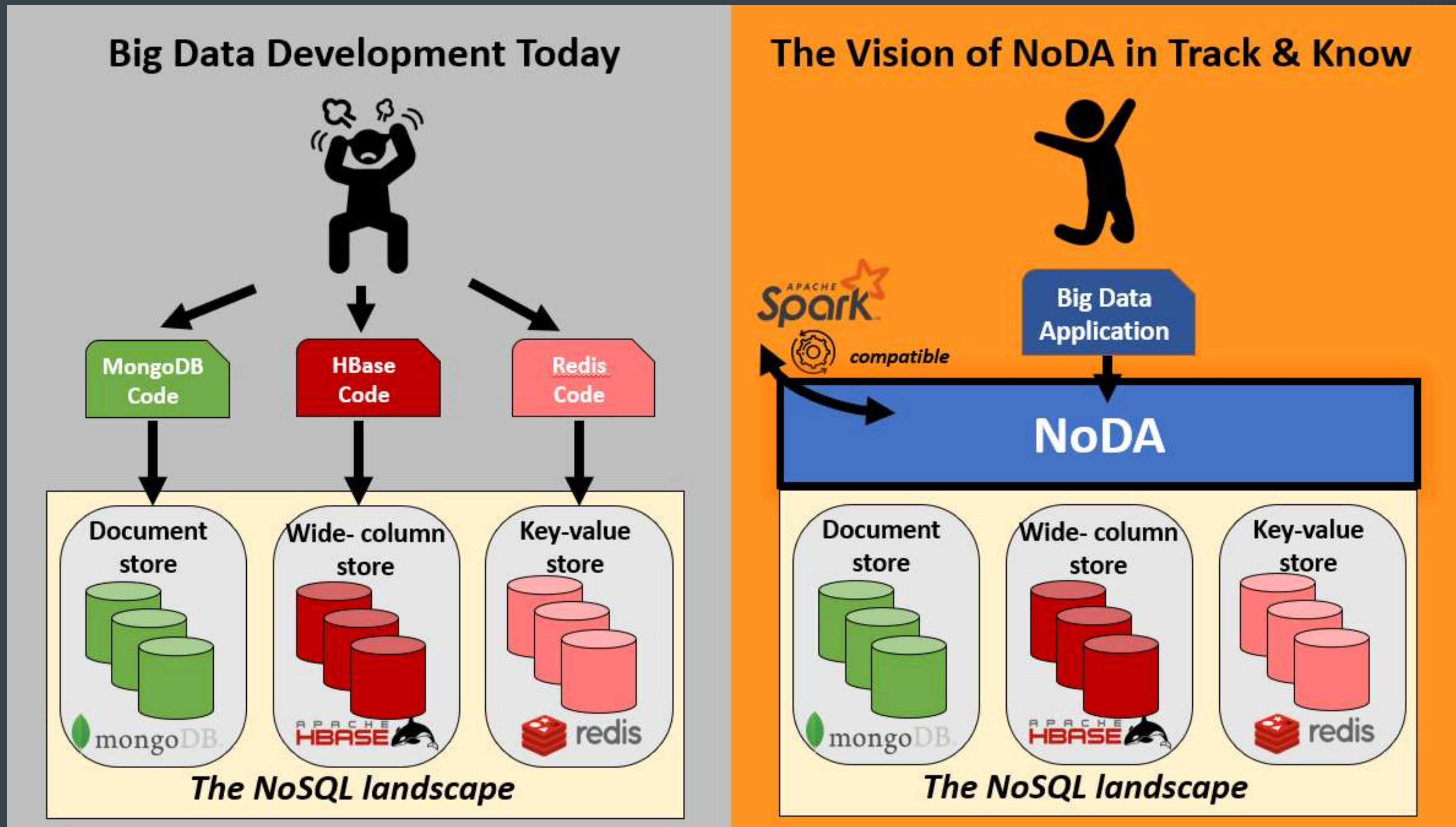


Big Data Analytics

We need “something” that enables:

- transparent access to various data sources
- generic handling of diverse data organization
- unified definition of processing pipelines
- modular design and implementation
- easy integration with programming languages

Big Data Analytics



Track & Know

Big Data for Mobility Tracking
Knowledge Extraction in Urban Areas

Overview

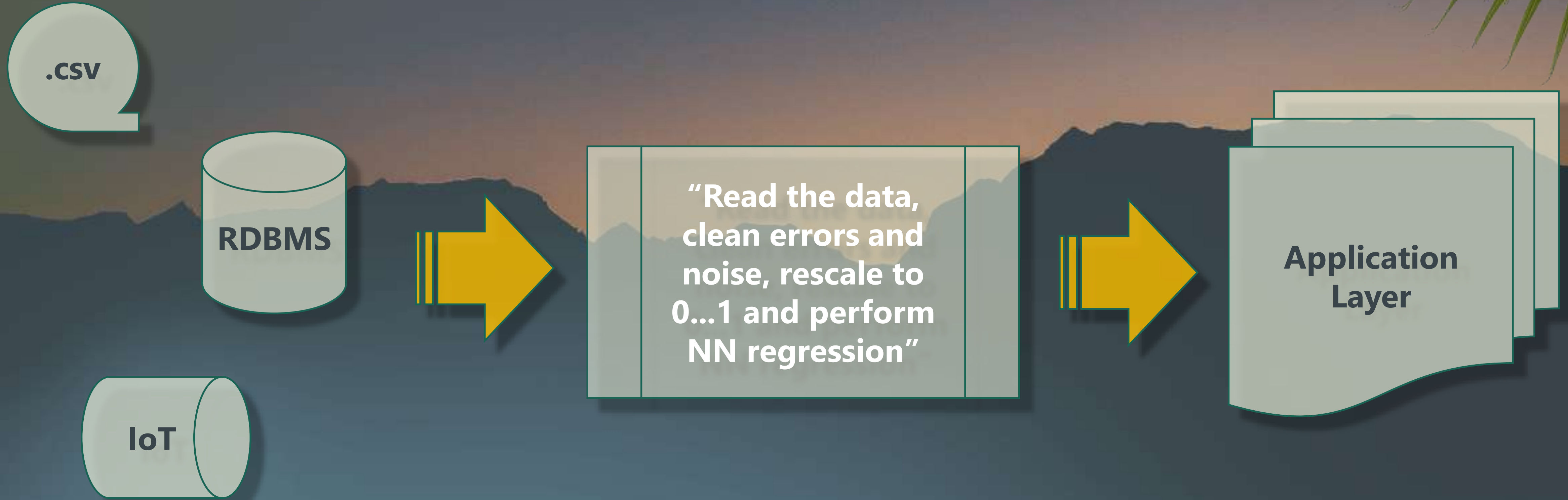
3. Architectural design

How is **Sequoia** designed? What is its underlying structure?

4. Platform and tools

How is **Sequoia** implemented? What tools does it use?

Sequoia – The concept



Functional "decoupling"

Introduce a functionally "rich" abstraction layer between data and the applications.

Sequoia – The concept

```
select avg(x) as x_bar,
       avg(y) as y_bar
from ols;
```

```
select x, avg(x) over () as x_bar,
       y, avg(y) over () as y_bar
from ols;
```

```
select sum((x - x_bar) * (y - y_bar)) / sum((x - x_bar) * (x - x_bar)) as slope
from (
    select x, avg(x) over () as x_bar,
           y, avg(y) over () as y_bar
    from ols) s;
```

```
select slope,
       y_bar_max - x_bar_max * slope as intercept
from (
    select sum((x - x_bar) * (y - y_bar)) / sum((x - x_bar) * (x - x_bar)) as slope,
           max(x_bar) as x_bar_max,
           max(y_bar) as y_bar_max
    from (
        select x, avg(x) over () as x_bar,
               y, avg(y) over () as y_bar
        from ols) s;
)
```

source: http://www.silota.com/docs/recipes/sql-linear-regression.html

Example:

Linear Regression

$$y = mx + b$$

$$m = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$$

$$b = \bar{y} - m\bar{x}$$

The SQL solution…

Same concept, different design target, i.e., improper for clean & efficient DA/ML processing pipelines.

Lexers and Parsers: Lex/Flex & Yacc/Bison

Lexer

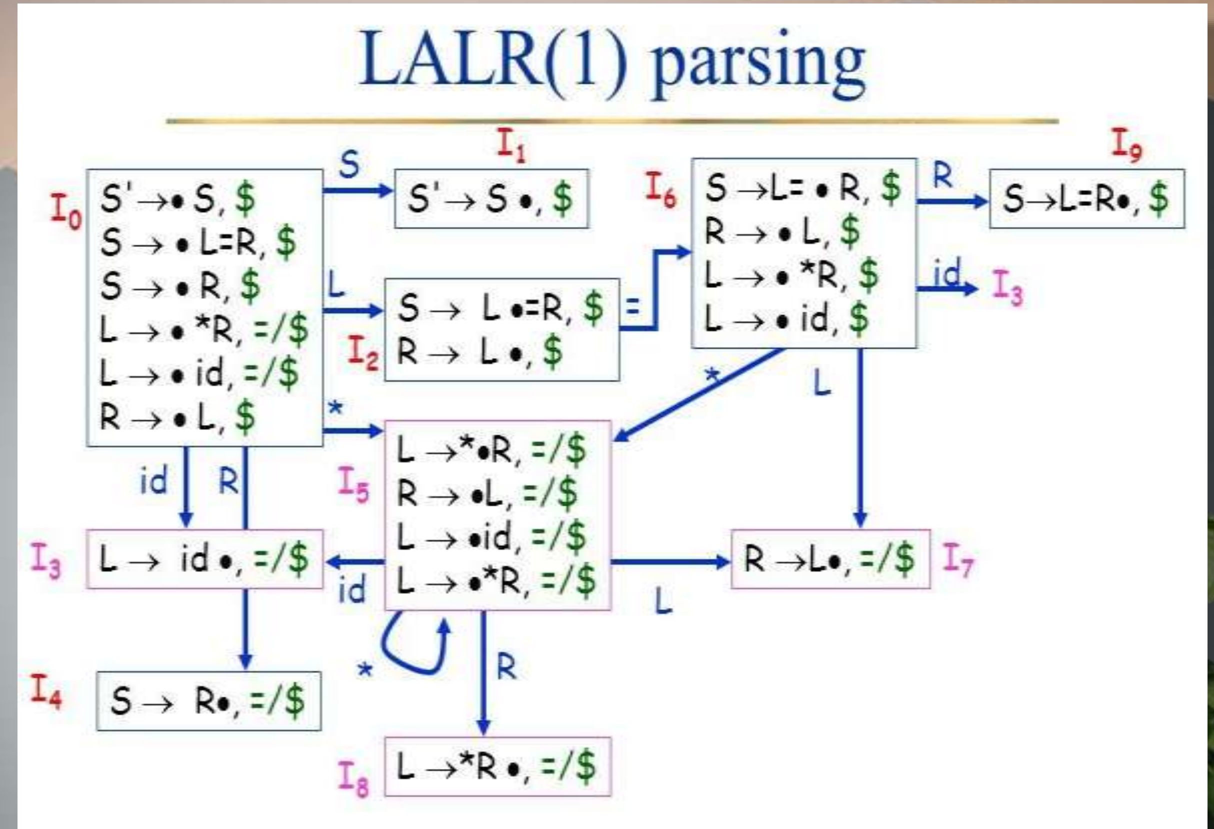
Lexical analyzer, processes text input and produces a stream of “tokens”, i.e., distinct valid items.

Parser

Syntax analyzer, processes a stream of “tokens” and matches patterns or “rules”, e.g. a language grammar.

LALR(*k*) parsers

Look-Ahead *k* Left-to-Right token stream “shift-reduce” analyzer, most compilers are designed like this.



source: https://slideplayer.com/slide/5370048/

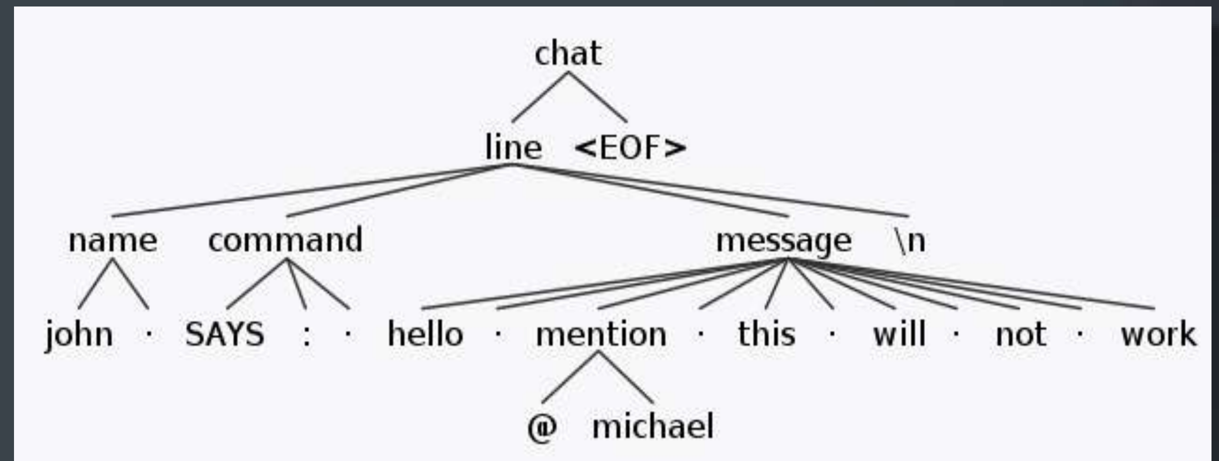
Python: “ply” and “sly”

Tools and packages available in Python:

- **cmd**: Simple framework for writing line-oriented command interpreters and command-line parsers (for options).

- **ANTLR**: “ANother Tool for Language Recognition”, generates parse trees from rules.

- **ply/sly**: Lex & Yacc native implementations in Python for creating “GNU” lexers and parsers.



source: https://tomassetti.me/antlr-mega-tutorial/

PLY (Python Lex-Yacc)

David M. Beazley
dave@dabeaz.com
PLY Version: 3.11

- Preface and Requirements
- Introduction
- PLY Overview
- Lex
    - Lex Example
    - The tokens list
    - Specification of tokens
    - Token values
    - Discarded tokens
    - Line numbers and positional information
    - Ignored characters
    - Literal characters
    - Error handling
    - EOF Handling

source: https://www.dabeaz.com/ply/ply.html

```
# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
```

SLY (Sly Lex Yacc)

THIS IS A WORK IN PROGRESS. NO OFFICIAL RELEASE HAS BEEN MADE. USE AT YOUR OWN RISK.

Requirements

SLY requires the use of Python 3.6 or greater. Older versions of Python are not supported.

Overview

SLY is a 100% Python implementation of the lex and yacc tools commonly used to write parsers and compilers. Parsing is based on the same LALR(1) algorithm used by many yacc tools. Here are a few notable features:

- SLY provides *very* extensive error reporting and diagnostic information to assist in parser construction. The original implementation was developed for instructional purposes. As a result, the system tries to identify the most common types of errors made by novice users.
- SLY provides full support for empty productions, error recovery, precedence specifiers, and moderately ambiguous grammars.
- SLY uses various Python metaprogramming features to specify lexers and parsers. There are no generated files or extra steps involved. You simply write Python code and run it.

```
precedence = (
    ('left', '+', '-'),
    ('left', '*', '/'),
    ('right', 'UMINUS'),
    )

def __init__(self):
    self.names = { }

@_('NAME "=" expr')
def statement(self, p):
    self.names[p.NAME] = p.expr

@_('expr')
def statement(self, p):
    print(p.expr)

@_('expr "+" expr')
def expr(self, p):
    return p.expr0 + p.expr1

@_('expr "-" expr')
def expr(self, p):
    return p.expr0 - p.expr1

@_('expr "*" expr')
def expr(self, p):
    return p.expr0 * p.expr1
```

source: https://sly.readthedocs.io/en/latest/index.html

Overview

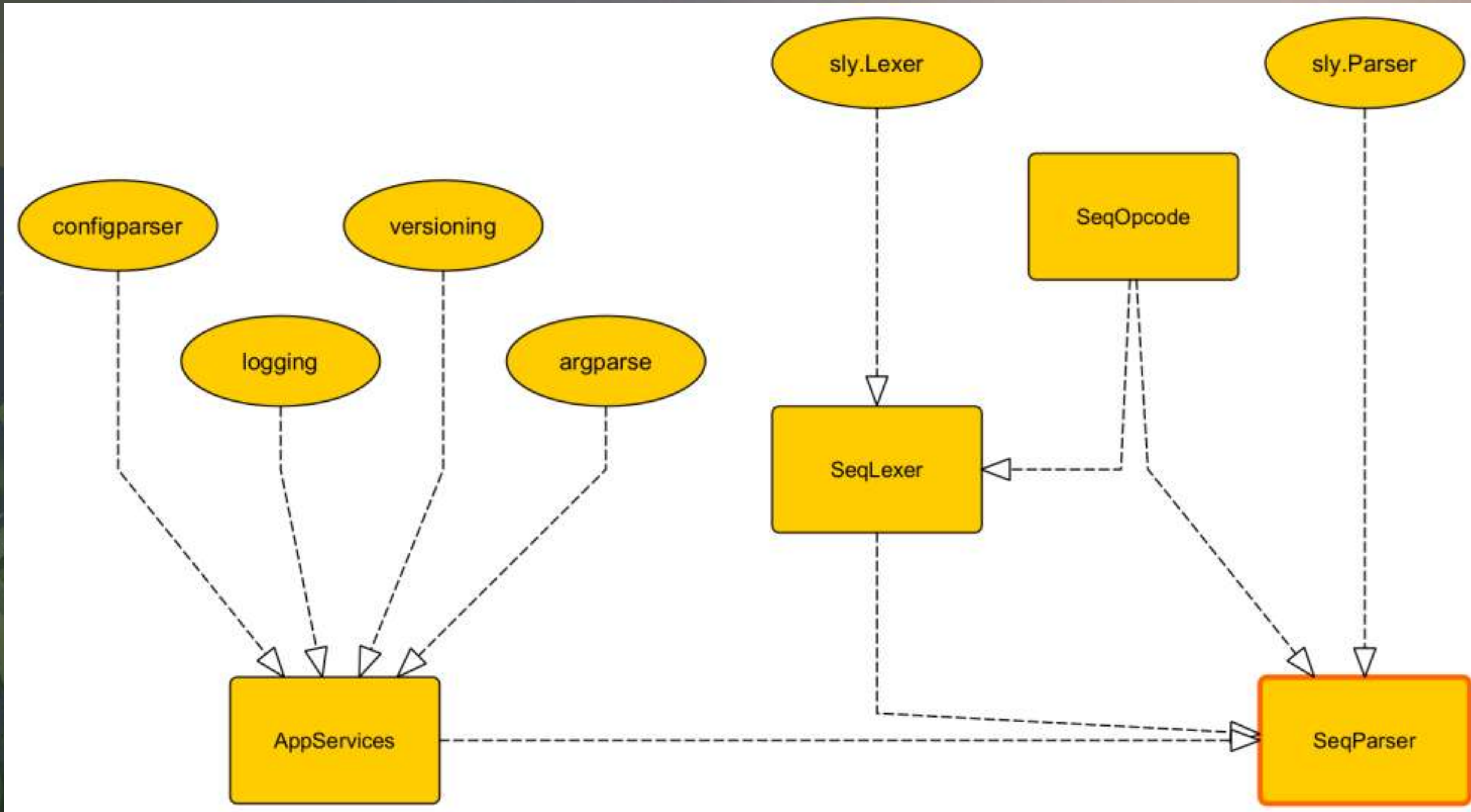
5. Classes and modules

What does **Sequoia** include? What are its core modules?

6. Application tools

Command-line options, configuration, logging, parsing.

Sequoia: Architectural design



The "parser" is (currently) the end-point of use, i.e., interactive (interpreter) and/or batch (compiler).

Sequoia: *AppServices.py – Initialization & versioning*

AppServices.py > ...

```python
import configparser, logging, logging.handlers, argparse, os
import app_info

class AppServices():

    def __init__(self, app_name='<ApplicationName>', app_version='<0.0.0>', app_copyright='<copyright>',
                 app_config='<default_cfg_filename>'):
        self.app_name=app_name
        self.app_version=app_version
        self.app_copyright=app_copyright
        self.app_config=app_config


    def get_sys_info(self):
        return (app_info.get_sys_info())


    def get_app_info(self):
        # retrieve application and system info, return as string
        return (app_info.get_app_info( self.app_name, self.app_version, self.app_copyright))


    def init_opt(self):
```

Sequoia: *AppServices.py – Command-line options*

AppServices.py > ...

```
17
18      def get_app_info(self):
19          # retrieve application and system info, return as string
20          return (app_info.get_app_info( self.app_name, self.app_version, self.app_copyright))
21
22
23      def init_opt(self):
24          self.cmdopt = argparse.ArgumentParser(description=self.app_name+' core functionality.')
25          # options '-h' and '--help' are implemented implicitly
26          self.cmdopt.add_argument('--version', action='version', version='%(prog)s '+self.app_version)
27          self.cmdopt.add_argument('-v', '--verbose', action='store_true', dest='verbose',
28                                  default=True, help='enable verbose mode, detailed logging is enabled')
29          self.cmdopt.add_argument('-q', '--quiet', action='store_true', dest='quiet',
30                                  default=True, help='enable quiet mode, no output is printed')
31          self.cmdopt.add_argument('-c', '--config', action='store', dest='cfgfile',
32                                 default=self.app_config, help=('configuration file (default=\'%s\')' % self.app_config))
33          args = app.cmdopt.parse_args()
34          self.opt = vars(args)
35
36
37      def init_cfg(self, fname=None):
38          if (fname != None):
```

Sequoia: *AppServices.py – Configuration service*

AppServices.py > ...

```python
31        self.cmdopt.add_argument('-c', '--config', action='store', dest='cfgfile',
32                      default=self.app_config, help=('configuration file (default=\'%s\')' % self.app_config))
33        args = app.cmdopt.parse_args()
34        self.opt = vars(args)
35
36
37    def init_cfg(self, fname=None):
38        if (fname != None):
39            cfg_filename = fname     # use the given filename if not null
40        else:
41            cfg_filename = self.opt['config']    # get from command-line options
42        # open configuration file for all application parameters
43        self.cfg = configparser.ConfigParser()
44        self.cfg.read_file(open(cfg_filename))      # read configuration from file
45
46
47    def init_log_timeR(self, fname, loglevel, logwhen, loginterval, logcycle):
48        # initialize logger with time-based rotation
49        self.log = logging.getLogger(__name__)      # use module name as logger id
50        self.log.setLevel(loglevel)                 # set logging level (from string)
51        fh = logging.handlers.TimedRotatingFileHandler(fname, when=logwhen, interval=int(loginterval), backupCount=
52        fmt = logging.Formatter('%(asctime)s; %(levelname)s; %(message)s')    # set default logging format
```

Sequoia: *AppServices.py – Configuration (example)*

settings.cfg

```
[DEFAULT]
; cfgfile = settings.cfg    ; not really useful as default in command-line option

[application]
; core application info embedded in the source files (these are unused)
; name = SEQUOIA Engine
; version = 0.0.1 (alpha)
; copyright = Harris Georgiou (c) 2020, Licence: CC-BY-SA/4.0i


[logging]
; filename base used for rotation
filename = error.log
; logging reporting level, valid values:
;   CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET
level = DEBUG
maxBytes = 10000000
when = M
interval = 1
backupCount = 5


```

Sequoia: *AppServices.py – Logging service*

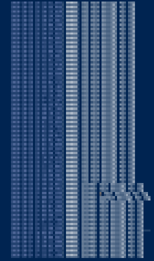
AppServices.py > ...

```
46
47     def init_log_timeR(self, fname, loglevel, logwhen, loginterval, logcycle):
48         # initialize logger with time-based rotation
49         self.log = logging.getLogger(__name__)      # use module name as logger id
50         self.log.setLevel(loglevel)                 # set logging level (from string)
51         fh = logging.handlers.TimedRotatingFileHandler(fname, when=logwhen, interval=int(loginterval), backupCount=
52         fmt = logging.Formatter('%(asctime)s; %(levelname)s; %(message)s')    # set default logging format
53         fh.setLevel(loglevel)
54         fh.setFormatter(fmt)
55         self.log.addHandler(fh)
56
57
58     def init_log_sizeR(self, fname, loglevel, loglimit, logcycle):
59         # initialize logger with size-based rotation
60         self.log = logging.getLogger(__name__)      # use module name as logger id
61         self.log.setLevel(loglevel)                 # set logging level (from string)
62         fh = logging.handlers.RotatingFileHandler(fname, maxBytes=int(loglimit), backupCount=int(logcycle))
63         #fh = logging.handlers.TimedRotatingFileHandler(fname, when='m', interval=1, backupCount=3)
64         fmt = logging.Formatter('%(asctime)s; %(levelname)s; %(message)s')    # set default logging format
65         fh.setLevel(loglevel)
66         fh.setFormatter(fmt)
67         self.log.addHandler(fh)
68
```

Sequoia: *AppServices.py – Logging (example)*

error.log

```
1   2020-10-26 14:12:31,843; DEBUG; new test message for log
2   2020-10-26 14:19:45,946; DEBUG; new test message for log
3   2020-10-26 14:20:13,829; DEBUG; new test message for log
4   2020-10-26 14:20:29,695; DEBUG; new test message for log
5   2020-10-26 14:20:37,681; DEBUG; new test message for log
6   2020-10-26 14:24:09,259; DEBUG; new test message for log
7   2020-10-26 15:29:25,949; DEBUG; new test message for log
8   2020-10-26 15:29:37,622; DEBUG; new test message for log
9   2020-10-26 15:30:06,602; DEBUG; new test message for log
10  2020-10-26 15:32:31,854; DEBUG; new test message for log
11  2020-10-26 15:32:45,334; DEBUG; new test message for log
12  2020-10-26 15:33:09,736; DEBUG; new test message for log
13  2020-10-26 15:42:15,881; DEBUG; new test message for log
14  2020-10-26 15:43:01,234; DEBUG; new test message for log
15  2020-10-26 15:52:23,910; DEBUG; new test message for log
16  2020-10-26 15:52:44,299; DEBUG; new test message for log
17  2020-10-26 15:53:06,680; DEBUG; new test message for log
18  2020-10-26 15:53:27,122; DEBUG; new test message for log
19  2020-10-26 15:54:04,713; DEBUG; new test message for log
20  2020-10-26 15:56:26,103; DEBUG; new test message for log
21  2020-10-26 15:56:29,788; DEBUG; new test message for log
22  2020-10-26 15:57:18,296; DEBUG; new test message for log
```



Sequoia: *AppServices.py – Example of command-line usage*

```
ter/sequoia/AppServices.py --help
usage: AppServices.py [-h] [--version] [-v] [-q] [-c CFGFILE]

SEQUOIA Engine core functionality.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -v, --verbose         enable verbose mode, detailed logging is enabled
  -q, --quiet           enable quiet mode, no output is printed
  -c CFGFILE, --config CFGFILE
                        configuration file (default='settings.cfg')
```

- Command-line options are automatically processed by a AppServices class instance

Overview

7. SeqOpcodes

What does the ‘compiled’ code look like in the end (JIT-ready)?

8. SeqLexer

How is the text input analyzed into lexical ‘tokens’?

Sequoia: *SeqOpcode.py – Instruction codes (for JIT)*

SeqOpcode.py > ...

```python
# no imports necessary


class SeqOpcode():
    'class for defining low-level opcodes for compiler-oriented output'

    version = '0.0.1'       # core engine/opcodes version

    opcodes = {
        'NOTHING' : 'NOP',
        'USE'     : 'USE',
        'CONNECT' : 'CONN',
        'ASSIGN'  : 'SET',
        'RETURN'  : 'RET'
    }

    stmtsepar = ';'

    def __init__(self):
        pass

```

- Opcodes are the validated pipeline ‘steps’ to be executed by the JIT engine.

Sequoia: *SeqLexer.py – Lexical analyzer (Lex/Flex-like)*

SeqLexer.py > SeqLexer

```
 4
 5
 6  class SeqLexer(Lexer):
 7      'class: lexical analyzer/tokenizer'
 8
 9      tokens = { IDENT, FLOAT, INT, ASSIGN, LPAREN, RPAREN, QUIT, RESET, CLEAR, LIST, CONNECT, USE, RETURN }#, SEMICO
10      ignore = ' \t'     # skip by default
11
12      # Tokens
13      IDENT = r'[a-zA-Z_][.a-zA-Z0-9_]*'      # valid identifiers and keywords
14      #NUMBER = r'\d+'
15      #FLOAT = r'/^(?!0\d)\d*(\.\d+)?$/mg'
16      FLOAT = r'[-]*\d+[.]\d+'                # signed decimal numbers (preceeding)
17      INT = r'[-]*\d+'                        # signed integers (after floats)
18
19      # Special symbols
20      ASSIGN  = r'='
21      LPAREN  = r'\('
22      RPAREN  = r'\)'
23      #SEMICOL = r';'         # default statement end, if multiples on the same text line
24
25      # literals (chars)
```

- All lexical tokens can be clearly defined by just three regular expressions.

Sequoia: *SeqLexer.py – Lexical analyzer (Lex/Flex-like)*

SeqLexer.py > SeqLexer

```python
42      def __init__(self):
43          'default constructor'
44          self.reset()    # reset all internal state/counters
45          #print('interactive: ',self.interactive,' , verbose: ',self.verbose)
46
47      def reset(self, interactive=True, verbose=True):
48          'reset lexer status'
49          self.hasErrors=False
50          self.verbose=verbose
51          self.interactive=interactive
52
53      # Extra action for newlines (keep track)
54      def ignore_newline(self, t):
55          'update line count, ignore as token'
56          self.lineno += t.value.count('\n')
57
58      def error(self, t):
59          'default error handler for illegal tokens (characters)'
60          print('Error (line %d): Illegal character %r' % (self.lineno, t.value[0]))
61          self.index += 1     # skip current position, continue with input
62          self.hasErrors=True
63
```

- Ignore newlines silently, but keep track of the count for error reporting.

Overview

9. SeqParser

How is the ‘token’ stream analyzed syntactically by the compiler?

10. Putting it all together

What does **Sequoia** look like in the interpreter mode?

Sequoia: *SeqParser.py – Syntax analyzer (Yacc/Bison-like)*

SeqParser.py > ...

```
58      def oplist_clear(self):
59          'clear all opcode data (reset)'
60          self.oplist = {          # main opcode listing, updated during parsing
61              'meta'    : {            # oplist: metadata section
62                  'name' : None,
63                  'version' : SeqOpcode.version,
64                  'mode'      : None,
65                  'author'    : None,
66                  'copyright' : None,
67                  'timestamp' : None,
68
69              },
70              'input'   : {            # oplist: input section
71                  #'dbname'   : None,
72                  #'dbtype'   : None,
73
74              },
75              'steps'   : {            # oplist: pipeline section
76
77              },
78              'output' : {             # oplist: output section
79                  #'return'   : None,
```

- Opcode 'compiled' listing is a multi-level dictionary updated on-the-fly by the interpreter.

Sequoia: *SeqParser.py – Syntax analyzer (Yacc/Bison-like)*

SeqParser.py > ...

```
83
84      def oplist_meta(self, name, author, copyright):
85          'define metadata section'
86          self.oplist['meta']['name'] = name
87          self.oplist['meta']['author'] = author
88          self.oplist['meta']['copyright'] = copyright
89          self.oplist['meta']['version'] = SeqOpcode.version
90          self.oplist['meta']['mode'] = self.mode
91          self.oplist['meta']['timestamp'] = str(datetime.now())
92
93      def oplist_validate(self):
94          'validate pipeline definition'
95          if (not (self.hasInput and ('dbname' in self.oplist['input']))):
96              print('Error: No \'USE\' or \'CONNECT\' input defined')
97              self.hasErrors=True
98
99          if (not (self.hasOutput and ('return' in self.oplist['output']))):
100             print('Error: No \'RETURN\' output defined')
101             self.hasErrors=True
102
103         return (self.hasErrors)
104
```

- Opcode listing can be validated at any time in interpreter (interactive) or compiler (batch) mode.

Sequoia: *SeqParser.py – Syntax analyzer (Yacc/Bison-like)*

SeqParser.py > ...

```
104
105     def cap_keywords(self, inptext):
106         'capitalize keywords for case-insensitive matching in rules'
107         for kw in self.keywords:        # convert all keywords to upper case
108             re_exp = re.compile(kw, re.IGNORECASE)
109             inptext = re_exp.sub(kw, inptext)
110         return(inptext)
111
112     def done(self, errcode=0):
113         'gracefully finish parsing process and exit'
114         if (self.verbose):
115             print('Finished parsing %d statements:' % len(self.oplist['steps']))
116             print('Identifiers: ',self.names)
117             print('oplist:\n',self.oplist)
118             if (self.hasErrors):
119                 print('Exiting with errors (code=%d)' % errcode)
120             else:
121                 print('No errors found in input')
122
123         if (self.hasErrors and errcode != 0):    # should also catch Lexer errors
124             exit(errcode)
125         elif (self.hasErrors):      # exit with errors, no code given
126             exit(1)
```

- Keywords and syntactical rules (but **not** identifiers) are converted to case-insensitive by default.

Sequoia: *SeqParser.py – Syntax analyzer (Yacc/Bison-like)*

SeqParser.py > ...

```
135
136     @_('RESET')          # Rule:  'RESET'
137     def statement(self, p):
138         #self.stmtno += 1       # do not count 'quit' as statement (interactive mode)
139         self.reset()
140         if (self.verbose):
141             print('Parser reset: OK')
142
143     @_('CLEAR')          # Rule:  'CLEAR'
144     def statement(self, p):
145         #self.stmtno += 1       # do not count 'quit' as statement (interactive mode)
146         self.clear_vars()
147         if (self.verbose):
148             print('Parser clear vars: OK')
149
150     @_('LIST')          # Rule:  'LIST'
151     def statement(self, p):
152         #self.stmtno += 1       # do not count 'quit' as statement (interactive mode)
153         self.list_vars()
154         if (self.verbose):
155             print('Parser list vars: OK')
156
```

- Interpreter-specific functionality can also be integrated in the core engine, without opcode generation.

Sequoia: *SeqParser.py – Syntax analyzer (Yacc/Bison-like)*

SeqParser.py > ...

```
156
157     @_('USE IDENT')     # Rule:  'USE <filename>'
158     def statement(self, p):
159         # if (p.IDENT not in self.names):
160         #     print(f'Error (line %d): \'USE\' with undefined identifier {p.IDENT!r}' % p.lineno)
161         # elif ('$dbname' in self.names):
162         # elif (self.hasInput):
163         if (self.hasInput):
164             print(f'Error (line %d): Input already defined as {p.IDENT!r}' % p.lineno)
165         else:
166             self.names['$dbname'] = p.IDENT                          # update identifiers dictionary (special $)
167             self.stmtno += 1       # count as valid statement (in oplist)
168             self.oplist['input']['dbname'] = self.names['$dbname']          # update oplist (input filename)
169             self.oplist['input']['dbtype'] = 'csv/text'                     # update oplist (input type)
170             #self.oplist['steps'][self.stmtno] = SeqOpcode.opcodes['USE']    # insert new pipeline step (opcode)
171             #self.oplist['steps'][self.stmtno] = (SeqOpcode.opcodes['USE'], self.names['$dbname'])  # opcode conter
172             self.hasInput=True
173             if (self.verbose):
174                 print('%s: \'%s\'' % (SeqOpcode.opcodes['USE'], self.names['$dbname']))
175
176     @_('CONNECT IDENT')     # Rule:  'USE <filename>'
177     def statement(self, p):
```

- Normal pipeline 'steps' produce opcodes, i.e., updates to the opcode listing.

Sequoia: *SeqParser.py – Syntax analyzer (Yacc/Bison-like)*

SeqParser.py > ...

```
218
219     @_('IDENT ASSIGN expr')     # Rule:  '<identifier> = <expression>|<identifier>'
220     def statement(self, p):
221         self.names[p.IDENT] = p.expr        # update identifiers dictionary (new) and set value
222         #self.names['$ans']=p.expr           # update latest-result identifier (auto)
223         self.stmtno += 1        # count as valid statement (in oplist)
224         self.oplist['steps'][self.stmtno] = SeqOpcode.opcodes['ASSIGN']    # insert new pipeline step (opcode)
225         self.oplist['steps'][self.stmtno] = (SeqOpcode.opcodes['ASSIGN'], p.IDENT, self.names[p.IDENT])  # opcode
226         # Note: on invalid <expr> the p.expr value becomes 'None', i.e. means 'unset <identifier>'
227         if (self.verbose):
228             print('set: \'%s\' = %s' % (p.IDENT,self.names[p.IDENT]))
229             #print('(auto): $ans = %s' % self.names['$ans'])
230
231     @_('expr')        # Rule:  '<expression>|<identifier>' (print current value)
232     def statement(self, p):
233         if (p.expr != None):
234             print(p.expr)        # useful only in interactive mode (no oplist update)
235         #self.names['$ans']=p.expr
236         #if (self.verbose):
237         #    print('(auto): $ans = %s' % self.names['$ans'])
238
239
```

➢ A sequence of pipeline 'steps' create numbered opcodes with a tuple of arguments.

Sequoia: *SeqParser.py – Syntax analyzer (Yacc/Bison-like)*

SeqParser.py > ...

```
262         #except LookupError:
263         else:
264             print(f'Error (line %d): Undefined identifier {p.IDENT!r}' % p.lineno)
265             self.hasErrors=True
266             self.stmtno += 1
267             #return 0
268
269     def error(self, p):
270         if (p==None):      # default error for incomplete rule matching
271             print('Error: Invalid or incomplete statement %d' % self.stmtno)
272         elif ((p.type=='INT') or (p.type=='FLOAT')):
273             print('Error (line %d): Invalid number expression' % p.lineno)
274         elif (p.type=='IDENT'):      # Note: keywords already catched by lexer/tokenizer
275             print('Error (line=%d): Invalid identifier \'%s\'' % (p.lineno, p.value))
276         else:              # default error for any other case
277             #print('Error: ',p)     # should be catched earlier
278             print('Error (line=%d): Invalid use of \'%s\'' % (p.lineno, p.value))
279
280         self.hasErrors=True
281         self.stmtno += 1           # count errors as statements
282
283
```

- Special note: Incomplete commands require rule-specific error definitions or (here) a global handler.

Sequoia: *SeqParser.py – Syntax analyzer (Yacc/Bison-like)*

seqparser.out

```
1    Grammar:
2
3    Rule 0     S' -> statement
4    Rule 1     statement -> expr
5    Rule 2     statement -> IDENT ASSIGN expr
6    Rule 3     statement -> RETURN IDENT
7    Rule 4     statement -> CONNECT IDENT
8    Rule 5     statement -> USE IDENT
9    Rule 6     statement -> LIST
10   Rule 7     statement -> CLEAR
11   Rule 8     statement -> RESET
12   Rule 9     statement -> QUIT
13   Rule 10    expr -> IDENT
14   Rule 11    expr -> INT
15   Rule 12    expr -> FLOAT
16   Rule 13    expr -> LPAREN expr RPAREN
17
18   Terminals, with rules where they appear:
19
20   ASSIGN               : 2
21   CLEAR                : 7
22   CONNECT              : 4
```

Sequoia: *SeqParser.py – Syntax analyzer (Yacc/Bison-like)*

SeqParser.py > ...

```
282
283
284  # MAIN: stand-alone mode, used for unit testing only
285  if __name__ == '__main__':
286      lexer = SeqLexer()
287      parser = SeqParser()
288
289      #print('same-line statement separator: ',SeqOpcode.stmtsepar)
290      parser.interactive=True
291      parser.oplist_meta('SeqParser','Harris Georgiou','CC-BY-SA (c) 2020')
292
293      while (True):
294          try:
295              text = input('seqp > ')
296              text = re.sub(SeqOpcode.stmtsepar,'\n',text)    # replace ';' with newline characters
297              #print('input: ',text)
298              for stmt in text.splitlines(True):              # split lines, input separately to parser
299                  #print('\tstmt: ',stmt)
300                  parser.parse( lexer.tokenize( parser.cap_keywords(stmt) ) )    # parse line, update internally
301          except EOFError:
302              break     # Ctrl+Z exits
303
```

- Interpreter: Main loop is just 8-10 lines of code all-inclusive (tokenizer, split lines, parser, JIT engine).

Sequoia: *SeqParser.py – Interpreter mode (interactive)*

```
ter/sequoia/SeqParser.py
Parser debugging for SeqParser written to seqparser.out
seqp > use some.db
USE: 'some.db'
seqp > a=3
set: 'a' = 3
seqp > t=a
set: 't' = 3
seqp > return t
return: 't'
seqp > list
'$ans'   :  <class 'NoneType'>
'$dbname'        :  <class 'str'>
'a'      :  <class 'int'>
't'      :  <class 'int'>
'$return'        :  <class 'str'>
Parser list vars: OK
seqp >
```

Sequoia: *SeqParser.py – Interpreter mode (interactive)*

```
seqp > t=a
set: 't' = 3
seqp > return t
return: 't'
seqp > list
'$ans'   :  <class 'NoneType'>
'$dbname'        :  <class 'str'>
'a'      :  <class 'int'>
't'      :  <class 'int'>
'$return'        :  <class 'str'>
Parser list vars: OK
seqp > quit
Finished parsing 2 statements:
Identifiers:  {'$ans': None, '$dbname': 'some.db', 'a': 3, 't': 3, '$return': 't
'}
oplist:
 {'meta': {'name': 'SeqParser', 'version': '0.0.1', 'mode': 'interactive', 'auth
or': 'Harris Georgiou', 'copyright': 'CC-BY-SA (c) 2020', 'timestamp': '2020-11-
16 08:52:32.222805'}, 'input': {'dbname': 'some.db', 'dbtype': 'csv/text'}, 'ste
ps': {2: ('SET', 'a', 3), 4: ('SET', 't', 3)}, 'output': {'return': 't'}}
No errors found in input
```

- Final opcode listing is JSON-compatible, validated and ready for the JIT engine.

Overview

11. Development status

What is the current status of the development?

12. Future work

What’s next?

Sequoia: Current development

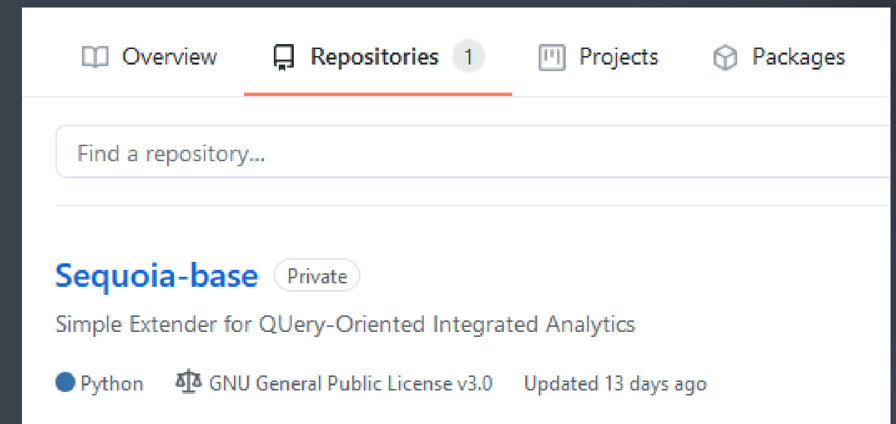
- **Timeline**: Currently one developer, working on it in free time.
- About 3-4 weeks in total, roughly 20 man-hours all-inclusive.

- **Codebase**: Currently very small, a little over 500 lines (22KB).
- ...But very packed and uses many external modules.

- **Python**: Using 3.8.x branch, various add-in packages (ply/sly).
- Note that 3.9.x still has setup/versioning problems with 'pip'.

- **JIT**: Internal engine is used by the interpreter (interactive) mode and the compiler (batch) mode, producing opcodes.

- Slowly establishing project repository and group (GitHub).



Sequoia: Future work

- **Timeline**:
  - Finish 2-3 more sprints by the end of the year.
  - Involve students and others for collaboration.

- **Functionality**:
  - Finish interpreter mode, used as tool for iterative testing.
  - Progress on compiler/opcodes/executor modules.
  - Support XML/JSON as default 'compiled' format.
  - Support opcode verification via hashing/blockchain.

- **Core engine**:
  - Connect to SQLite, PostgreSQL, flat-file (.csv).
  - Adaptive filtering: Kalman, LMS/RLS (for pre-processing).
  - GPU processing: TensorFlow/Keras (for matrix operations).

Sequoia

Simple Extension for QUery-Oriented Integrated Analytics

Harris Georgiou (MSc,PhD), Data Science Lab,
University of Piraeus, Greece

@ FOSSCOMM 2020 (virtual)

Email: hgeorgiou@unipi.gr
https://github.com/xgeorgio/Sequoia-base