CSE211: Compiler Design

Oct. 4, 2021

- **Topic**: Parser Generator Example (PLY)
- **Questions**:
    - *Do you have any experience with a parser generator?*

from: https://en.wikipedia.org/wiki/Yak

Announcements

- Homework 1 is released
  - Cover PLY today
  - Cover symbol tables next class
  - Cover parsing with derivatives on Friday
- Pushes us back 1 day in the schedule
- if you have ideas for projects, we can start discussing!
- Join the slack for discussions!

From the discussion

is == associative?

((0 == 1) == 0)

(7 == (0 == 0))

CSE211: Compiler Design

Oct. 4, 2021

- **Topic**: Parser Generator Example (PLY)

- **Questions**:
  - *Do you have any experience with a parser generator?*

from: https://en.wikipedia.org/wiki/Yak

Parser generators

- Specify:
  - Tokens
  - Production Rules
  - Production Actions

- Parser generator gives you a function in which you can pass strings
  - Executes production actions
  - Error reporting

Historically

- Lex
    - lexer
    - released in 1975
    - co-developed by Eric Schmidt
    - ”Flex” is a common open-source implementation
    - historically outputs a .c file

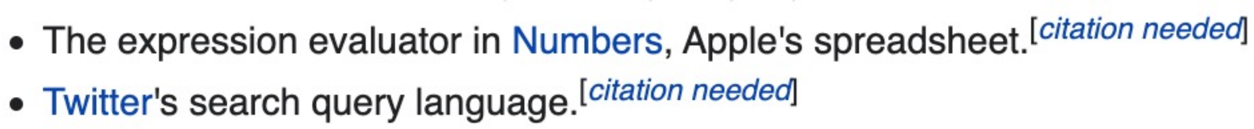
- Yacc (Yet Another Compiler Compiler)
    - parser
    - released in 1975
    - originally written in B, but soon rewritten in C
    - interface is widely supported, but newer implementations are more widely used now
    - historically outputs a .c file

Historically

- Bison
  - Parser only, often coupled with flex
  - Released in 1985: latest release was Sept. 2021
  - better error tracking and debugging
  - compatible with yacc rules
  - outputs C/++, Java

More modern

- Antlr
  - Lexer and Parser
  - Released 1992, latest release was March 2021
  - BSD License
  - From Wikipedia, used in:



- The expression evaluator in Numbers, Apple's spreadsheet.[citation needed]
- Twitter's search query language.[citation needed]

  - Outputs: Python, Javascript, C#, Swift

- Others: https://en.wikipedia.org/wiki/Comparison_of_parser_generators

PLY

- An implementation of Lex and Yacc in Python

- links:
  - source: https://github.com/dabeaz/ply
  - docs: https://ply.readthedocs.io/en/latest/

- We are going to build several parsers today

- Your homework augments this example in several ways:
  - *Variables, Scope, Precedence, Associativity*

Demo

- *Lots of thanks to the excellent PLY documentation! Some functions are copied from there*

- *Setup:*
    - *clone the ply repo*
    - *make a new directory*
    - *copy the ply/ directory into the directory*

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

Lexer Demo

- *Library import*

```
import ply.lex as lex
```

- *Token list*

```
tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE"]
```

- *Token specification*

```
t_ADJECTIVE = "old|purple|spotted"
t_NOUN = "dog|computer|car"
t_ARTICLE = "the|my|a|your"
t_VERB = "ran|crashed|accelerated"
```

Lexer Demo

- *Build the lexer*

```
    lexer = lex.lex()
```

what happens?

- *Need an error function*

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

Lexer Demo

- *Now give the lexer some input*

```
lexer.input("dog")
```

- *The lexer streams the input, we need to stream the tokens:*

```
# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break       # No more input
    print(tok)
```

Lexer Demo

- *output:*

line number (1 indexed)

```
LexToken(NOUN, 'dog’, 1, 0)
```

number of characters streamed
(0 indexed)

- *try a longer string:*

```
lexer.input("dog computer")
```

What happens?

Lexer Demo

- *Need to add a token for whitespace!*

```
tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE", "WHITESPACE"]
…
t_WHITESPACE = '\ '
```

- *Now we can lex:*

```
LexToken(NOUN,'dog',1,0)
LexToken(WHITESPACE,' ',1,3)
LexToken(NOUN,'computer',1,4)
```

Lexer Demo

- *Now we can do a sentence*

```
lexer.input("my spotted dog ran")
```

```
LexToken(ARTICLE,'my',1,0)
LexToken(WHITESPACE,' ',1,2)
LexToken(ADJECTIVE,'spotted',1,3)
LexToken(WHITESPACE,' ',1,10)
LexToken(NOUN,'dog',1,11)
LexToken(WHITESPACE,' ',1,14)
LexToken(VERB,'ran',1,15)
```

Can we clean this up?

Lexer Demo

- *We can ignore whitespace*

```
#t_WHITESPACE = '\
t_ignore = ' '
```

gets simplified to:

```
LexToken(ARTICLE,'my',1,0)
LexToken(WHITESPACE,' ',1,2)
LexToken(ADJECTIVE,'spotted',1,3)
LexToken(WHITESPACE,' ',1,10)
LexToken(NOUN,'dog',1,11)
LexToken(WHITESPACE,' ',1,14)
LexToken(VERB,'ran',1,15)
```

```
LexToken(ARTICLE,'my',1,0)
LexToken(ADJECTIVE,'spotted',1,3)
LexToken(NOUN,'dog',1,11)
LexToken(VERB,'ran',1,15)
```

Lexer Demo

- *What about newlines?*

```
lexer.input("""
my spotted dog ran
the old computer crashed
""")
```

- *Need to add a newline token!*

Lexer Demo

- *What about newlines?*

```
lexer.input("""
my spotted dog ran
the old computer crashed
""")
```

- *Need to add a newline token!*

```
tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE", "NEWLINE"]

t_NEWLINE = "\\n"
```

Lexer Demo

```
LexToken(NEWLINE,'\n',1,0)
LexToken(ARTICLE,'my',1,1)
LexToken(ADJECTIVE,'spotted',1,4)
LexToken(NOUN,'dog',1,12)
LexToken(VERB,'ran',1,16)
LexToken(NEWLINE,'\n',1,19)
LexToken(ARTICLE,'the',1,20)
```

Line numbers are not updating

Lexer Demo

- *Token actions, similar to production actions*

```
t_NEWLINE = "\\n"
```

Changes into:

```
def t_NEWLINE(t):
    "\\n"
    t.lexer.lineno += 1
    return t
```

docstring is the regex, lexer object which has a linenumber attribute.

If we don’t return anything, then it is ignored.

Lexer Demo

- *Example: changing gendered pronouns into gender neutral pronouns*

```
tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE", "NEWLINE", "PRONOUN"]
t_PRONOUN = "her|his|their"
```

```
lexer.input("""
his spotted dog ran
her old computer crashed
""")
```

Lexer Demo

- *Add a token action:*

```
def t_PRONOUN(t):
    "her|his|their"
    if t.value in ["his", "her"]:
        t.value = "their"
    return t
```

Now output will have all gender neutral pronouns!

Multiline calculator example

- For this, we will use lexer and parser

- input:
  - 1 or more mathematical expressions separated by a ;
  - mathematical expressions can have non-negative integers as operands
  - mathematical operators are +,-,*,/ and ()

- output:
  - the solution to each expression

Multiline calculator example

```
import ply.lex as lex

tokens = ["NUM", "MULT", "PLUS", "MINUS", "DIV", "LPAR", "RPAR", "SEMI", "NEWLINE"]

t_NUM = '[0-9]+'
t_MULT = '\*'
t_PLUS = '\+'
t_MINUS = '-'
t_DIV = '/'
t_LPAR = '\('
t_RPAR = ‘\)'
t_SEMI = ";"

t_ignore = ' '

def t_NEWLINE(t):
    "\\n"
    t.lexer.lineno += 1

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

lexer = lex.lex()
```

Set up the lexer

Multiline calculator example

- *Import the library*

```python
import ply.yacc as yacc
```

- Simple rule

```python
def p_expr_num(p):
    "expr : NUM"
    p[0] = int(p[1])
```

functions are given prefixed by `p_`

production rules are the doc string

return values are stored in p[0]
children values are in p[1], p[2], etc.

Multiline calculator example

- *Try it out*

```
parser = yacc.yacc(debug=True)

result = parser.parse("5")
print(result)
```

Multiline calculator example

- *Next rule*

```
def p_expr_plus(p):
    "expr : expr PLUS expr"
    p[0] = p[1] + p[3]
```

- Try it again

```
result = parser.parse("5 + 4")
print(result)
```

What errors are we getting? Can we look into them?

Multiline calculator example

- *Set an error function*

```
def p_error(p):
    print("Syntax error in input!")
```

- Set associativity (and precedence)

```
precedence = (
    ('left', 'PLUS'),
)
```

Multiline calculator example

- *Next rules*

```
def p_expr_minus(p):
    "expr : expr MINUS expr"
    p[0] = p[1] – p[3]


def p_expr_mult(p):
    "expr : expr MULT expr"
    p[0] = p[1] * p[3]


def p_expr_div(p):
    "expr : expr DIV expr"
    p[0] = p[1] / p[3]
```

```
precedence = [
    ('left', 'PLUS', 'MINUS'),
    ('left', 'MULT', 'DIV'),
]
```

Multiline calculator example

- *Last rule for expressions*

```
def p_expr_par(p):
    "expr : LPAR expr RPAR"
    p[0] = p[2]
```

Multiline calculator example

- *An extra we can easily implement*

```
def p_expr_div(p):
    "expr : expr DIV expr"
    if p[3] == 0:
        print("divide by 0 error:")
        print("cannot divide: " + str(p[1]) + " by 0")
        exit(1)
    p[0] = p[1] / p[3]
```

Multiline calculator example

- *Combining rules:*

```
def p_expr_plus(p):
    "expr : expr PLUS expr"
    p[0] = p[1] + p[3]


def p_expr_minus(p):
    "expr : expr MINUS expr"
    p[0] = p[1] - p[3]


def p_expr_mult(p):
    "expr : expr MULT expr"
    p[0] = p[1] * p[3]
```

```
def p_expr_bin(p):
    """
    expr : expr PLUS expr
         | expr MINUS
         | expr MULT expr
    """
    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]
    else:
        assert(False)
```

See you on Wednesday!

- Talk more about symbol tables and start talking about parsing with derivatives

- Might have to finish up Module 1 on Friday
  - put us 1 day behind the schedule

- Homework 1 is released
  - From today’s lecture you should be able to get started on part 1