ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

BÁO CÁO BÀI TẬP LỚN MỞ RỘNG

NGUYÊN LÝ NGÔN NGỮ LẬP TRÌNH (CO300C)

Parser Generator: PLY

GV hướng dẫn: ThS. Trần Ngọc Bảo Duy
SV thực hiện: Lê Duy Anh 2112762
Nguyễn Trần Bảo Ngọc 2111860

Ho Chi Minh City, January, 2024

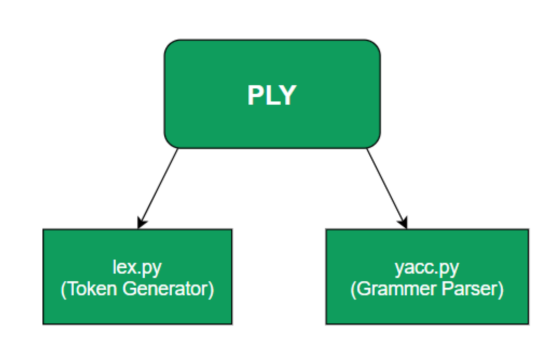
Contents

1 PLY - Python Lex-Yacc

PLY [1] là bản triển khai bằng ngôn ngữ lập trình **Python** của các công cụ **Lex (Lexical analyzer generator)** và **YACC (Yet Another Compiler Compiler)** thường được sử dụng để hiện thực trình phân tích cú pháp và trình biên dịch.

Trong PLY, hai công cụ này được triển khai dưới dạng hai modules riêng biệt: `lex.py` và `yacc.py` trong một package Python `ply` [2]. Các modules này hoạt động tương tự như hai công cụ **Lex** và **YACC** nguyên bản nói trên.

Hai công cụ này làm việc cùng nhau và hỗ trợ lẫn nhau. Cụ thể, module lex.py được sử dụng để chia văn bản đầu vào thành một tập hợp các token được xác định bởi các quy tắc biểu thức chính quy. Còn yacc.py sử dụng các token đó để nhận diện cú pháp của ngôn ngữ được mô tả dưới dạng văn phạm phi ngữ cảnh (context-free grammar). Đầu ra của yacc.py thường là cây cú pháp trừu tượng (Abstract Syntax Tree - AST). Tuy nhiên, điều này hoàn toàn phụ thuộc vào người lập trình. Nếu muốn, yacc.py cũng có thể được sử dụng để triển khai các trình biên dịch một lần đơn giản (simple one-pass compilers).



Hình 1: Hai module lex.py và yacc.py

1.1 Lex

Module lex.py được sử dụng trong phase đầu tiên **Lexical Analyzer** của quá trình biên dịch, dùng để tách chuỗi đầu vào thành các token riêng lẻ.

- Input: source program (trong các trường hợp đơn giản sẽ là các chuỗi đầu vào - input string)
- Output: các token riêng lẻ được tách ra từ input

Ví dụ, giả sử bạn đang viết một ngôn ngữ lập trình và người dùng cung cấp chuỗi đầu vào sau:

```
x = 3 + 42 * (s - t)
```

Tokenizer sẽ chia chuỗi đầu vào trên thành các token riêng lẻ:

```
'x', '=', '3', '+', '42', '*', '(', 's', '-', 't', ')'
```

Từng token tương ứng với một "tên" do người lập trình đặt, dùng để xác định xem chúng là gì:

```
'ID', 'EQUALS', 'NUMBER', 'PLUS', 'NUMBER',
'TIMES', 'LPAREN', 'ID', 'MINUS', 'ID', 'RPAREN'
```

Để cụ thể hơn, chuỗi đầu vào sẽ được chia thành các cặp loại token (token type) và giá trị của chúng như sau:

```
('ID', 'x'), ('EQUALS', '='), ('NUMBER', '3'), ('PLUS', '+'),
('NUMBER', '42'), ('TIMES', '*'),('LPAREN', '('), ('ID', 's'),
('MINUS', '-'), ('ID', 't'), ('RPAREN', ')')
```

1.2 Yacc

Module `yacc.py` được sử dụng trong phase tiếp theo **Syntax Analyzer** của quá trình biên dịch, dùng để phân tích cú pháp của ngôn ngữ lập trình.

- Input: chuỗi các token (token stream) được tách ra từ `lex.py`
- Output: tuỳ vào người lập trình, có thể là cây cú pháp (parse tree), hoặc cây cú pháp trừu tượng (abstract syntax tree - AST)

1.2.1 Input Grammar Notation

Module `yacc.py` chỉ hỗ trợ cú pháp (syntax) viết dưới dạng **BNF (Backus-Naur Form)**. Ví dụ:

```
expression : expression + term
           | expression - term
           | term

term       : term * factor
           | term / factor
           | factor

factor     : NUMBER
           | ( expression )
```

1.2.2 Output Languages

Output languages của PLY là chính là **mã nguồn Python** chứa các hàm và logic xử lý cú pháp, bởi PLY là một công cụ trong Python được thiết kế để hỗ trợ việc xây dựng lexer và parser. Vì vậy, khi sử dụng PLY để triển khai lexer và parser cho một ngôn ngữ cụ thể, tức là ta đang sử dụng Python để mô tả cú pháp của ngôn ngữ đó.

Ví dụ: sử dụng các hàm Python để xử lý và thực hiện các hành động khi một quy tắc ngữ cảnh được nhận diện (đối với luật sinh `expression` ở trên) như sau:

```
def p_expression(p):
'''
expression : expression + term
           | expression - term
           | term
'''
if len(p) == 4:
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
else:
    p[0] = p[1]
```

1.2.3 Parsing Algorithm

Module `yacc.py` sử dụng một kỹ thuật phân tích cú pháp được biết đến là **LR-parsing hoặc shift-reduce parsing**. LR parsing là một kỹ thuật **bottom-up**: **cố gắng nhận diện phần bên phải (right-hand-side)** của các grammar rules được định nghĩa. Cụ thể, khi một phần bên phải hợp lệ của một grammar rule được tìm thấy trong input, hành động thích hợp sẽ được thực hiện để **thay thế chúng bằng phần bên trái (left-hand-side) của grammar rule đó**.

Thuật toán này thường được triển khai bằng cách đẩy các grammar synbols vào một ngăn xếp (stack), sau đó sẽ xem xét trạng thái stack và ký tự đầu vào tiếp theo để tìm các pattern phù hợp với một trong các grammar rules.

Khi phân tích biểu thức, một máy trạng thái ẩn (underlying state machine) và ký tự đầu vào hiện tại xác định những gì sẽ xảy ra tiếp theo:

- Nếu ký tự đầu vào tiếp theo giống như một phần của một grammar rule hợp lệ (dựa trên các mục khác trên stack), nó thường sẽ được đẩy vào stack.
- Nếu đỉnh stack chứa phần bên phải hợp lệ của một grammar rule, nó sẽ được reduce và chúng được thay thế bằng phần bên trái của grammar rule đó.
- Nếu ký tự đầu vào không thể được đẩy vào và đỉnh ngăn xếp không khớp với bất kỳ quy tắc ngữ cảnh nào, một lỗi cú pháp đã xảy ra và trình phân tích cú pháp phải thực hiện một bước khôi phục nào đó (hoặc thoát ra).
- Một phân tích chỉ thành công nếu trình phân tích cú pháp đạt đến một trạng thái trong đó ngăn xếp ký tự trống và không còn ký tự đầu vào nào.

Sau đây là một ví dụ khi phân tích biểu thức cú pháp biểu thức $3 + 5 * (10 - 20)$ sử dụng các grammar rules được định nghĩa ở trên (ký tự đặc biệt $ đại diện cho kết thúc input):

Step	Symbol Stack	Input Tokens	Action
1		$3 + 5 * (10 - 20)$$	Shift 3
2	3	$+5 * (10 - 20)$$	Reduce factor : NUMBER
3	factor	$+5 * (10 - 20)$$	Reduce term : factor
4	term	$+5 * (10 - 20)$$	Reduce expr : term
5	expr	$+5 * (10 - 20)$$	Shift +
6	expr +	$5 * (10 - 20)$$	Shift 5
7	expr + 5	$*(10 - 20)$$	Reduce factor : NUMBER
8	expr + factor	$*(10 - 20)$$	Reduce term : factor
9	expr + term	$*(10 - 20)$$	Shift *
10	expr + term *	$(10 - 20)$$	Shift (
11	expr + term * (	$10 - 20)$$	Shift 10
12	expr + term * (10	$-20)$$	Reduce factor : NUMBER
13	expr + term * (factor	$-20)$$	Reduce term : factor
14	expr + term * (term	$-20)$$	Reduce expr : term
15	expr + term * (expr	$-20)$$	Shift -
16	expr + term * (expr -	$20)$$	Shift 20
17	expr + term * (expr - 20	$)$$	Reduce factor : NUMBER
18	expr + term * (expr - factor	$)$$	Reduce term : factor
19	expr + term * (expr - term	$)$$	Reduce expr : expr - term
20	expr + term * (expr	$)$$	Shift)
21	expr + term * (expr)	$$	Reduce factor : (expr)
22	expr + term * factor	$$	Reduce term : term * factor
23	expr + term	$$	Reduce expr : expr + term
24	expr	$$	Reduce expr
25		$$	Success!

Bảng 1: Bảng phân tích cú pháp LALR cho biểu thức $3 + 5 * (10 - 20)$

1.3 Comparison with ANTLR4

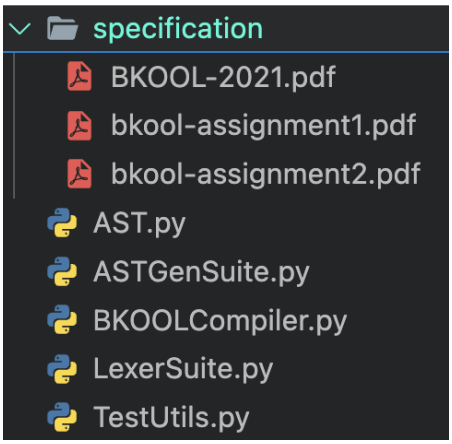
	PLY	ANTLR4
Parsing Algorithm	- LALR (Look-Ahead LR) - Bottom-up parsing technique	- Adaptive LL (LL(*)) - Top-down parsing technique
Input Grammar Notation	- BNF (Backus-Naur Form) - Định nghĩa các token và grammar rule bằng Python code.	- EBNF (Extended Backus-Naur Form) - Định nghĩa các grammar rule bằng ANTLR grammar.
Output Languages	- Sử dụng ngôn ngữ Python để mô tả cú pháp và hành động của ngữ cảnh. - Code được viết bằng Python và tích hợp chặt chẽ với môi trường Python.	- Có khả năng sinh mã nguồn cho nhiều ngôn ngữ lập trình khác nhau, không chỉ giới hạn trong môi trường Java. - Hỗ trợ nhiều ngôn ngữ đầu ra như Java, C, Python, và nhiều ngôn ngữ khác.

Bảng 2: So sánh PLY và ANTLR4

2 BKOOL Implementation by PLY

2.1 Source code organization

Project Github link: https://github.com/duyanhhh1811/CO300C-bkool-parser-generator-by-PLY



Hình 2: Tổ chức mã nguồn của project

Trong đó:

- folder `specification` : Chứa đặc tả ngôn ngữ `BKOOL` cùng các yêu cầu của assignment 1 (Recognizer) và assignment 2 (AST Generator).
- `AST.py` : chứa các dataclass là đại diện cho các node (`Stmt`, `Expr`,...) của cây AST dùng trong AST Generation phase.
- `BKOOLCompiler.py` : chứa class `BKOOLCompiler`, là nơi hiện thực Lexer, Parser Generator và AST Generator của ngôn ngữ BKOOL bằng **PLY**.
- `LexerSuite.py` : chứa các testcases để kiểm thử Lexer của ngôn ngữ BKOOL.
- `ASTGenSuite.py` : chứa các testcases để kiểm thử AST Generator của ngôn ngữ BKOOL.
- `TestUtils.py` : chứa các phương thức phục vụ cho việc kiểm thử.

2.2 BKOOLCompiler implementation

Class `BKOOLCompiler.py` chứa 2 attributes chính: `lexer` và `parser` chính là **Lexer** và **Parser** của ngôn ngữ BKOOL hoạt động dựa vào các Lexer Rules, Grammar Rules và Parser Rule được định nghĩa trong cùng class:

Listing 1: BKOOLCompiler Class Constructor

```
class BKOOLCompiler:
    def __init__(self):
        # Create lexer
        self.lexer = lex.lex(module=self)
```

```
        # Create parser
        self.parser = yacc.yacc(module=self)
```

2.2.1 Lexer

Đầu tiên, định nghĩa danh sách các `tokens`, dùng để xác định tất cả các loại token có thể được Lexer nhận biết:

Listing 2: BKOOL Token List

```
    # Token list
        tokens = (
            # Keywords
            'BOOLEAN', 'BREAK', 'CLASS', 'CONTINUE', 'DO',
            'ELSE', 'EXTENDS', 'FLOAT', 'IF', 'INT', 'NEW',
            'STRING', 'THEN', 'FOR', 'RETURN', 'TRUE', 'FALSE',
            'VOID', 'NIL', 'THIS', 'FINAL', 'STATIC', 'TO', 'DOWNTO',
            # Operators
            'ADD_OP', 'SUB_OP', 'MUL_OP', 'FLODIV_OP', 'INTDIV_OP',
            'MOD_OP', 'EQUAL_OP', 'NEQUAL_OP', 'LT_OP', 'GT_OP', 'LTE_OP',
            'GTE_OP', 'OR_OP', 'AND_OP', 'NOT_OP', 'CONCAT_OP', 'ASSIGN_OP',
            'EQUAL_SIGN',
            # Seperators
            'LSB', 'RSB', 'LP', 'RP', 'LB', 'RB', 'SEMI', 'COLON', 'DOT', 'COMMA',
            # Atom
            'ID', 'INTLIT', 'FLOATLIT', 'STRINGLIT',
        )
```

Trong **PLY**, mỗi token được xác định bằng một cú pháp `t_<TOKEN_NAME>`, với `<TOKEN_NAME>` phải thuộc vào một trong những token đã khai báo trong danh sách `tokens` phía trên. Với một số token có quy tắc đơn giản như các OPERATOR hay SEPERATOR, có thể sử dụng chuỗi thô `r'...'` để xác định các token đó:

Listing 3: BKOOL OPEARATOR and SEPERATOR rules

```
    # Operator
    t_ADD_OP = r'\+'
    t_SUB_OP = r'-'
    t_MUL_OP = r'\*'
    t_FLODIV_OP = r'/'
    t_INTDIV_OP = r'\\'
    t_MOD_OP = r'%'
    t_EQUAL_OP = r'=='
    t_NEQUAL_OP = r'!='
    t_LT_OP = r'<'
    t_GT_OP = r'>'
    t_LTE_OP = r'<='
    t_GTE_OP = r'>='
    t_OR_OP = r'\|\|'
    t_AND_OP = r'&&'
    t_NOT_OP = r'!'
    t_CONCAT_OP = r'\^'
    t_ASSIGN_OP = r':='
```

```
t_EQUAL_SIGN = r'='

# Seperator
t_LSB = r'\['
t_RSB = r'\]'
t_LP = r'{'
t_RP = r'}'
t_LB = r'\('
t_RB = r'\)'
t_SEMI = r';'
t_COLON = r':'
t_DOT = r'\.'
t_COMMA = r','
```

Với các token có biểu thức chính quy phức tạp, quy tắc của token đó có thể được xác định bằng một hàm, với tên hàm chính là tên cú pháp biểu diễn cho token đó. Cụ thể, dưới đây là biểu diễn các quy tắc cho INTLIT, FLOATLIT, STRINGLIT và IDENTIFIER, lần lượt là đại diện cho giá trị của số nguyên, số thực, chuỗi và định danh:

Listing 4: BKOOL IDENTIFIER and LITERAL rules

```
# IDENTIFIER
def t_ID(self, t):
    r'[_a-zA-Z][_a-zA-Z0-9]*'
    t.type = self.reserved.get(t.value,'ID')
    return t

# FLOATLIT
def t_FLOATLIT(self, t):
    r'(\d+\.\d*[eE][-+]?\d+)|(\d+[eE][-+]?\d+)|(\d+\.\d*)'
    t.value = float(t.value)
    return t

# INTLIT
def t_INTLIT(self, t):
    r'\d+'
    t.value = int(t.value)
    return t

# STRINGLIT
def t_STRINGLIT(self, t):
    r'"([^"\\]|\\.)*"'
    t.value = t.value[1:-1]
    return t
```

Ngoài ra, cần phải bỏ qua các token *space*, *tab* và các token nằm trong comment (bao gồm in-line comment và multi-line comment):

Listing 5: BKOOL IDENTIFIER and LITERAL rules

```
# Ignore space ( ) and tab (\t)
t_ignore = ' \t'

# Ignore in-line comment (#)
```

```
def t_LINE_COMMENT(self, t):
    r'\#.*'
    pass

# Ignore multi-line comment (/* ... */)
def t_BLOCK_COMMENT(self, t):
    r'/\*(.|\n)*?\*/'
    pass
```

Cuối cùng, cần phải khai báo một dictionary đại diện cho các từ khoá được khai báo trước (reserved words) trong ngôn ngữ BKOOL. Mỗi từ khóa được ánh xạ đến một chuỗi tương ứng là tên của loại token tương ứng. Việc này giúp cho việc phân biệt giữa Identifier và Reserved words trở nên dễ dàng hơn.

Listing 6: BKOOL Reserved words list

```
# keyword list (reserved)
reserved = {
    'boolean': 'BOOLEAN', 'break': 'BREAK', 'class': 'CLASS',
    'continue': 'CONTINUE', 'do': 'DO', 'else': 'ELSE',
    'extends': 'EXTENDS', 'float': 'FLOAT', 'if': 'IF', 'int': 'INT',
    'new': 'NEW', 'string': 'STRING', 'then': 'THEN', 'for': 'FOR',
    'return': 'RETURN', 'true': 'TRUE', 'false': 'FALSE',
    'void': 'VOID', 'nil': 'NIL', 'this': 'THIS', 'final': 'FINAL',
    'static': 'STATIC', 'to': 'TO', 'downto': 'DOWNTO',
}
```

2.2.2 Parser/AST Generator

Các luật sinh được xác định bởi các hàm với tên `p_<PRODUCTION_NAME>`. Trong hầu hết mọi trường hợp, `<PRODUCTION_NAME>` là tên của luật bắt đầu (nằm bên trái dấu :). Dưới đây là một số hàm đại diện cho các luật sinh tương ứng:

Listing 7: BKOOL Grammar Rules

```
def p_program(self, ctx):
    '''
    program : classDeclList
    '''
    ctx[0] = Program(ctx[1])

def p_classDeclList(self, ctx):
    '''
    classDeclList : classDecl classDeclList
                  | classDecl
    '''
    if (len(ctx) == 3): ctx[0] = ctx[1] + ctx[2]
    elif (len(ctx) == 2): ctx[0]= ctx[1]

def p_classDecl(self, ctx):
    '''
    classDecl : CLASS ID classExtends LP classMemberList RP
```

```
        '''
        ctx[0] = [ClassDecl(Id(ctx[2]), ctx[5], ctx[3])]

    def p_classExtends(self, ctx):
        '''
        classExtends : EXTENDS ID
                     | empty
        '''
        if (len(ctx) == 3): ctx[0] = Id(ctx[2])
        else: ctx[0] = None

    def p_classMemberList(self, ctx):
        '''
        classMemberList : classMember classMemberList
                        | empty
        '''
        if (len(ctx) == 3): ctx[0] = ctx[1] + ctx[2]
        else: ctx[0] = []
```

Một số lưu ý quan trọng cấu trúc hàm luật sinh `p_` :

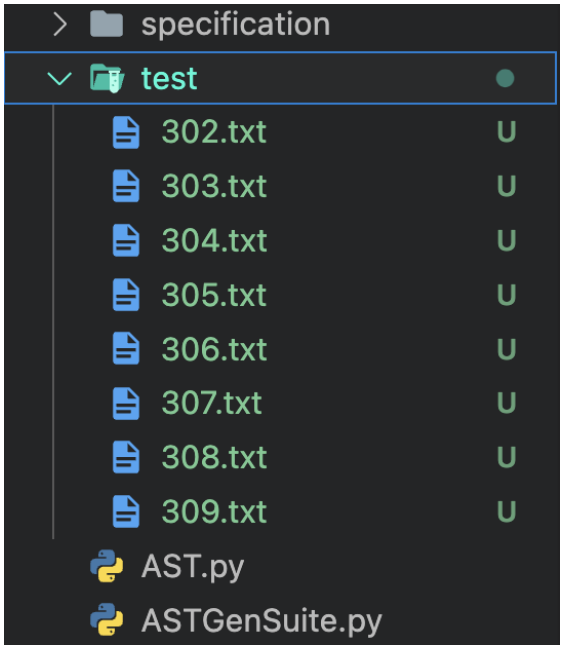
- Việc định nghĩa **docstring** (chuỗi nằm ngay bên dưới tên hàm) **là bắt buộc**, do PLY sử dụng các chuỗi này để xác định tất cả grammar rules của ngôn ngữ BKOOL và ánh xạ các thành phần của luật sinh đó đến hàm luật sinh `p_` tương ứng.
- Tham số `ctx` là một đối tượng có các thuộc tính và phương thức nhất định, có thể sử dụng chúng để truy cập thông tin về các ký tự đang được xử lý, giá trị của các token, và thậm chí là thực hiện các hành động ngữ nghĩa (semantic actions) tùy chỉnh. Sử dụng `ctx[i] (ctx[0], ctx[1],...)` để truy cập vào giá trị của thành phần (node) có chỉ số tương ứng trong luật sinh được định nghĩa trong docstring. `ctx[0` đại diện cho giá trị của node hiện tại, được tính toán dựa trên kết quả trả về từ các `ctx[1], ctx[2],...` . Nếu `ctx[i]` không phải terminal node, nó sẽ được ánh xạ đến hàm chứa luật sinh tương ứng với nó để tính toán giá trị và trả về kết quả.

2.3 User manual

1. Cài đặt framework `ply`: `pip3 install ply`
2. Lexer test: `python3 LexerSuite.py`
3. Parser and AST Generator test: `python3 ASTGenSuite.py`

2.4 Testing and Results

Sau khi chạy một trong 2 lệnh để kiểm thử ở trên, 1 folder `test` sẽ sinh ra, trong đó chứa các file là kết quả của từng testcases trong `LexerSuite.py` và `ASTGenSuite.py` . Tên của từng file chính là số thứ tự tương ứng với từng testcases.



Hình 3: Testcases Folder Result

```
Case:   #8
Result: True
Input:  /* Comment */ x := 1;
Expect: [('ID', 'x'), ('ASSIGN_OP', ':='), ('INTLIT', 1), ('SEMI', ';')]
Answer: [('ID', 'x'), ('ASSIGN_OP', ':='), ('INTLIT', 1), ('SEMI', ';')]
```

Hình 4: Lexer Testcase Result

```
Case:   #300
Result: True
Input:  class main {}
Expect: Program([ClassDecl(Id(main),[])])
Answer: Program([ClassDecl(Id(main),[])])
```

Hình 5: AST Generation Testcase Result

References

[1] David Beazley. Ply (python lex-yacc). https://ply.readthedocs.io/en/latest/, 2020. Accessed on 2023-12-17.

[2] David Beazley. ply. https://github.com/dabeaz/ply, 2023. Accessed on 2023-12-18.