Material de Aula: PLY (Python Lex-Yacc) em Python

Introdução

PLY (Python Lex-Yacc) é uma biblioteca Python que implementa ferramentas para análise léxica e sintática, inspirada nas clássicas ferramentas Lex e Yacc, amplamente usadas na construção de compiladores e interpretadores. O PLY é projetado para processar linguagens definidas por gramáticas livres de contexto, sendo ideal para aplicações como compiladores, interpretadores, analisadores de formatos de dados (ex.: JSON, XML, CSV), ou até mesmo ferramentas de validação de sintaxe para linguagens personalizadas. Ele é leve, não possui dependências externas e é totalmente implementado em Python, garantindo fácil integração e portabilidade. Além disso, o PLY é altamente personalizável, permitindo que desenvolvedores definam gramáticas complexas, lidem com erros de forma robusta e implementem ações semânticas avançadas, como a construção de árvores sintáticas ou a execução de cálculos em tempo real.

Contexto e Relevância

A análise léxica e sintática é fundamental no desenvolvimento de software que lida com linguagens estruturadas. Por exemplo, um compilador de uma linguagem de programação precisa primeiro tokenizar o código-fonte (análise léxica) e depois verificar se a estrutura do código segue as regras gramaticais da linguagem (análise sintática). O PLY simplifica esse processo ao fornecer uma interface intuitiva para definir tokens e gramáticas, além de suportar funcionalidades avançadas como estados, precedência e recuperação de erros. Comparado a outras ferramentas, como ANTLR ou PyParsing, o PLY se destaca por sua simplicidade e foco em Python puro, sem necessidade de gerar arquivos intermediários ou usar linguagens externas.

Objetivos

- Compreender os conceitos fundamentais de análise léxica e sintática e sua importância em sistemas computacionais.
- Explorar detalhadamente as funcionalidades do PLY, incluindo definição de tokens, gramáticas, precedência, estados e tratamento de erros.
- Entender como implementar um analisador para expressões aritméticas e como expandi-lo para cenários mais complexos, como linguagens com variáveis ou estruturas condicionais.
- Aprender a aplicar o PLY em casos práticos, com exemplos detalhados e cenários avançados.

Principais Funcionalidades do PLY

O PLY é composto por dois módulos principais que trabalham em conjunto para transformar uma entrada de texto em ações semânticas significativas:

1. **Lex**: Responsável pela análise léxica, que converte uma sequência de caracteres (como um código-fonte ou uma expressão) em uma sequência de tokens. Um token é uma unidade básica da linguagem, como um número, operador, identificador ou palavra-chave.
2. **Yacc**: Responsável pela análise sintática, que processa os tokens gerados pelo Lex para verificar se eles seguem as regras de uma gramática definida e executa ações associadas, como construir uma árvore sintática ou calcular um resultado.

Funcionalidades do PLY Lex

O módulo Lex do PLY é usado para definir e reconhecer tokens com base em expressões regulares. Ele é altamente configurável e suporta várias funcionalidades avançadas.

- **Definição de Tokens**: Tokens são definidos usando expressões regulares no formato `t_NOME = r'regex'` para tokens simples ou como funções `def t_NOME(t)` para tokens que requerem processamento adicional.
  Exemplo 1: Definindo tokens para operadores aritméticos:

  ```
  t_PLUS = r'\+'
  t_MINUS = r'-'
  t_TIMES = r'\*'
  t_DIVIDE = r'/'
  ```

  Esses tokens reconhecem os operadores `+`, `-`, `*` e `/`. Note que caracteres especiais como `+` e `*` precisam ser escapados (`\+`, `\*`) porque têm significado especial em expressões regulares.
  Exemplo 2: Definindo um token para números inteiros ou decimais:

  ```
  def t_NUMBER(t):
      r'\d+(\.\d+)?'
      t.value = float(t.value)  # Converte a string para float
      return t
  ```

  A expressão regular `\d+(\.\d+)?` reconhece números como `123` (inteiro) ou `123.45` (decimal). A função converte a string capturada (`t.value`) em um valor `float` e retorna o token.
  Exemplo 3: Definindo um token para identificadores (ex.: nomes de variáveis):

  ```
  def t_ID(t):
      r'[a-zA-Z_][a-zA-Z0-9_]*'
  ```

```
    t.value = str(t.value)  # Garante que o valor seja uma string
    return t
```

Aqui, `[a-zA-Z_][a-zA-Z0-9_]*` reconhece identificadores como `x`, `variable_1`, ou `sumTotal`, comumente usados em linguagens de programação.

- **Ignorar Caracteres**: Caracteres que não são relevantes para a análise, como espaços, tabulações ou quebras de linha, podem ser ignorados usando `t_ignore`.
  Exemplo 1: Ignorando espaços e tabulações:

  ```
  t_ignore = ' \t'
  ```

  Isso faz com que entradas como `3 + 5` e `3+5` sejam tratadas da mesma forma.
  Exemplo 2: Ignorando quebras de linha:

  ```
  t_ignore = ' \t\n'
  ```

  Útil para linguagens onde quebras de linha não têm significado sintático.
  Exemplo 3: Ignorando comentários:

  ```
  def t_comment(t):
      r'\#.*'
      pass  # Ignora o comentário
  ```

  A regex `\#.*` ignora tudo após um `#` até o fim da linha, como em `# este é um comentário`.

- **Tratamento de Erros**: A função `t_error` lida com caracteres que não correspondem a nenhum token definido, permitindo personalizar mensagens de erro ou implementar recuperação de erros.
  Exemplo 1: Exibindo mensagem de erro básica:

  ```
  def t_error(t):
      print(f"Caractere ilegal: '{t.value[0]}' na posição
  {t.lexer.lexpos}")
      t.lexer.skip(1)
  ```

  Para a entrada `3 @ 5`, o lexer exibe “Caractere ilegal: ‘@’ na posição 2” e avança para o próximo caractere. A propriedade `t.lexer.lexpos` indica a posição atual no texto de entrada.
  Exemplo 2: Contando erros:

  ```
  errors = 0
  def t_error(t):
      global errors
      errors += 1
  ```

```
    print(f"Caractere ilegal: '{t.value[0]}'")
    t.lexer.skip(1)
```

Aqui, cada caractere inválido incrementa a variável `errors`, útil para relatórios de análise.

- **Estados**: O PLY suporta estados para análise léxica condicional, permitindo que o lexer mude de comportamento com base no contexto (ex.: strings, comentários, modos especiais).
  Exemplo 1: Reconhecendo strings entre aspas:

```
states = (('string', 'exclusive'),)

def t_string(t):
    r'"'
    t.lexer.begin('string')

def t_string_content(t):
    r'[^"]+'
    t.value = t.value
    return t

def t_string_end(t):
    r'"'
    t.lexer.begin('INITIAL')

def t_string_error(t):
    print(f"Erro: string não fechada")
    t.lexer.skip(1)
```

Ao encontrar ", o lexer entra no estado `string`. Ele captura o conteúdo até encontrar outra " e retorna ao estado inicial (`INITIAL`). Se o conteúdo não for válido, `t_string_error` é chamado.
Exemplo 2: Comentários aninhados (ex.: `/* comentario */`):

```
states = (('comment', 'exclusive'),)

def t_comment_start(t):
    r'/\*'
    t.lexer.comment_count = 1
    t.lexer.begin('comment')

def t_comment_end(t):
    r'\*/'
    t.lexer.comment_count -= 1
    if t.lexer.comment_count == 0:
        t.lexer.begin('INITIAL')

def t_comment_start_nested(t):
```

```
    r'/\*'
    t.lexer.comment_count += 1

def t_comment_any(t):
    r'[^*/]+'
    pass  # Ignora conteúdo
```

Isso suporta comentários aninhados como `/* comentario /* aninhado */ fim */`, contando os níveis de aninhamento.

Funcionalidades do PLY Yacc

O módulo Yacc do PLY processa os tokens gerados pelo Lex, verificando se eles seguem uma gramática definida e executando ações associadas, como cálculos ou construção de estruturas de dados.

- **Gramáticas**: As gramáticas são definidas em funções com docstrings no formato BNF (Backus-Naur Form), especificando como os tokens podem ser combinados para formar construções válidas.
  Exemplo 1: Gramática para expressões aritméticas básicas:

```
def p_expression(p):
    '''expression : NUMBER
                  | expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression'''
    if len(p) == 2:  # expression : NUMBER
        p[0] = p[1]
    else:  # Operações binárias
        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]
```

Essa gramática define que uma `expression` pode ser um `NUMBER` (ex.: `5`) ou uma operação binária como `3 + 5`. O valor resultante é atribuído a `p[0]`.
Exemplo 2: Gramática para instruções com variáveis:

```
def p_statement(p):
    '''statement : ID EQUALS expression'''
    p[0] = ('assign', p[1], p[3])
```

Para uma entrada como `x = 5 + 3`, isso gera uma tupla `('assign', 'x', 8)`, representando a atribuição.

- **Ações Semânticas**: Cada regra sintática pode executar uma ação para processar os dados, como calcular valores, construir árvores sintáticas ou executar comandos.
  Exemplo 1: Calculando valores (já mostrado acima).
  Exemplo 2: Construindo uma árvore sintática:

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

  Para `3 + 5`, isso gera `('+', 3, 5)`. Para `3 + 5 * 2`, a árvore seria `('+', 3, ('*', 5, 2))`.
  Exemplo 3: Executando comandos em tempo real:

```
def p_statement_print(p):
    '''statement : PRINT expression'''
    print(f"Saída: {p[2]}")
    p[0] = None
```

  Para `print 5 + 3`, isso imprime "Saída: 8" durante o parsing.

- **Precedência**: Regras de precedência e associatividade resolvem ambiguidades em gramáticas.
  Exemplo 1: Definindo precedência para operadores aritméticos:

```
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),  # Menos unário
)
```

  - `'left'` para `PLUS` e `MINUS` faz com que `3 + 5 - 2` seja `(3 + 5) - 2`.
  - `TIMES` e `DIVIDE` têm maior prioridade, então `3 + 5 * 2` é `3 + (5 * 2)`.
  - `'right'` para `UMINUS` faz com que `- - 3` seja interpretado como `-(-3) = 3`.

    Exemplo 2: Adicionando uma operação de potência:

```
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'POWER'),
    ('right', 'UMINUS'),
)
```

  Aqui, `POWER` (ex.: ^) tem maior precedência, então `2 ^ 3 * 4` é `(2 ^ 3) * 4 = 8 * 4 = 32`.

- **Tratamento de Erros**: A função `p_error` gerencia erros sintáticos, e o PLY suporta recuperação de erros para continuar o parsing.
  Exemplo 1: Mensagem de erro básica:

  ```
  def p_error(p):
      if p:
          print(f"Erro sintático no token '{p.value}' na posição
  {p.lexpos}")
      else:
          print("Erro sintático: entrada incompleta")
  ```

  Para `3 + + 5`, exibe "Erro sintático no token '+'". Para `3 +`, exibe "Erro sintático: entrada incompleta".
  Exemplo 2: Recuperação de erros:

  ```
  def p_error(p):
      if not p:
          print("Erro: entrada incompleta")
          return
      # Pula tokens até encontrar um ponto de sincronização
      while True:
          tok = parser.token()
          if not tok or tok.type == 'SEMICOLON':
              break
          parser.errok()
  ```

  Para uma linguagem que usa `;` como separador, isso pula tokens até encontrar um `;` ou o fim da entrada, permitindo continuar o parsing.

Integração entre Lex e Yacc

O Lex e o Yacc trabalham juntos: o Lex tokeniza a entrada, e o Yacc consome esses tokens para construir a análise sintática. O PLY gerencia automaticamente a comunicação entre os dois, mas é possível personalizar o processo.
Exemplo: Para passar informações do lexer para o parser:

```
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.value = (t.value, t.lexer.lineno)  # Armazena o número da linha
    return t
```

No parser:

```
def p_statement(p):
    '''statement : ID EQUALS expression'''
    var_name, line = p[1]  # Extrai nome e linha
    p[0] = ('assign', var_name, p[3], line)
```

Isso permite que o parser acesse informações adicionais, como a linha onde um token foi encontrado, útil para mensagens de erro detalhadas.

Instalação

Instale o PLY usando o comando:

```
pip install ply
```

Certifique-se de ter o Python instalado (versão 3.6 ou superior é recomendada). O PLY não possui dependências adicionais, mas requer permissões para instalação de pacotes via pip.

Exemplo Prático: Analisador de Expressões Aritméticas

Vamos detalhar a construção de um interpretador para expressões aritméticas que suporta adição (+), subtração (-), multiplicação (*), divisão (/) e parênteses (( )), com suporte a menos unário (ex.: `-3`). Este exemplo é um caso prático comum para aprender PLY e pode ser expandido para cenários mais complexos.

Explicação do Processo

1. **Análise Léxica**:
    - **Tokens Definidos**:
        - `NUMBER`: Números inteiros ou decimais (ex.: `123`, `123.45`).
        - `PLUS`, `MINUS`, `TIMES`, `DIVIDE`: Operadores `+`, `-`, `*`, `/`.
        - `LPAREN`, `RPAREN`: Parênteses `(` e `)`.
    - **Conversão de Valores**: O token `NUMBER` converte strings numéricas em valores `float`. Ex.: `"123.45"` vira `123.45`.
    - **Ignorar Caracteres**: Espaços, tabulações e quebras de linha são ignorados, permitindo entradas como `3 + 5` ou `3+5`.
    - **Erros Léxicos**: Caracteres inválidos, como `@` em `3 @ 5`, geram uma mensagem como “Caractere ilegal: ‘@’” e o lexer avança.
2. **Análise Sintática**:
    - **Precedência**:
        - `TIMES` e `DIVIDE` têm maior prioridade que `PLUS` e `MINUS`.
        - `UMINUS` (menos unário) tem a maior prioridade.
        - Associatividade à esquerda para operadores binários, à direita para menos unário.
        - Ex.: `3 + 5 * 2` é `3 + (5 * 2) = 13`.
        - Ex.: `- - 3` é `-(-3) = 3`.
    - **Regras Gramaticais**:
        - Operações binárias: `expression : expression PLUS expression` (ex.: `3 + 5`).
        - Menos unário: `expression : MINUS expression` (ex.: `-3`).
        - Parênteses: `expression : LPAREN expression RPAREN` (ex.: `(3 + 5)`).

        - Números: `expression : NUMBER` (ex.: `123`).
    - **Ações**: Cada regra calcula o valor da expressão. Ex.: Para `expression PLUS expression`, somamos os valores das subexpressões (`p[0] = p[1] + p[3]`).

    - **Erros Sintáticos**: Entradas inválidas como `3 + + 5` geram uma mensagem de erro sintático, e o parser pode ser configurado para recuperação.
3. **Execução**:
    - O programa oferece um prompt interativo (`calc >`) onde o usuário insere expressões.
    - O lexer tokeniza a entrada, o parser aplica as regras gramaticais e retorna o resultado.
    - Exemplo de fluxo para `3 + 5 * 2`:
        - Tokenização: `NUMBER(3)`, `PLUS`, `NUMBER(5)`, `TIMES`, `NUMBER(2)`.
        - Parsing: Resolve `5 * 2 = 10` (devido à precedência de *), depois `3 + 10 = 13`.
        - Saída: `13.0`.

Como Testar

1. Execute o programa.
2. No prompt `calc >`, teste expressões como:
    - `3 + 5` (deve retornar `8.0`)
    - `3 + 5 * 2` (deve retornar `13.0`)
    - `(3 + 5) * 2` (deve retornar `16.0`)
    - `-3 + 2` (deve retornar `-1.0`)
    - `10 / 2 + 3` (deve retornar `8.0`)
    - `2 + 3 * 4 - 1` (deve retornar `13.0`, pois `2 + (3 * 4) - 1 = 2 + 12 - 1`)
    - `- (3 + 2)` (deve retornar `-5.0`)
    - Digite `exit` para sair.
3. Teste entradas inválidas:
    - `3 + + 5` (deve exibir um erro sintático).
    - `3 @ 5` (deve exibir um erro léxico para `@`).
    - `(3 + 5` (deve exibir um erro sintático por parêntese não fechado).

Expansão do Exemplo

Podemos expandir o analisador para suportar: - **Variáveis**: Adicione um token `ID` para identificadores e uma regra para atribuição: `python def p_statement_assign(p): '''statement : ID EQUALS expression''' p[0] = ('assign', p[1], p[3])` Isso permite entradas como `x = 5 + 3`, que podem ser armazenadas em um

dicionário para uso posterior.
- **Funções Matemáticas**: Adicione suporte para funções como `sin` ou `sqrt`: python `def p_expression_func(p): '''expression : SIN LPAREN expression RPAREN''' import math p[0] = math.sin(p[3])` Para `sin(0)`, isso retorna `0.0`.

Exercícios

1. Adicione suporte para diferenciar números inteiros e decimais, tratando-os como tipos diferentes (`INTEGER` e `FLOAT`), e implemente conversões específicas no parser (ex.: `INTEGER` para `int`, `FLOAT` para `float`).
2. Implemente a operação de potência (`^`), com precedência maior que multiplicação (ex.: `2 ^ 3 * 4` deve ser `(2 ^ 3) * 4 = 32`).
3. Crie uma função que gere uma árvore sintática (ex.: `3 + 5` vira `('+', 3, 5)`) e implemente uma função para visualizá-la em formato texto (ex.: `(+ 3 5)`).
4. Adicione suporte para comentários na entrada, ignorando texto após `//` até o fim da linha, e permita comentários aninhados com `/* */`.
5. Implemente recuperação de erros no parser para sugerir correções (ex.: para `3 + + 5`, sugerir remover o `+` extra; para `(3 + 5`, sugerir adicionar um `)`).
6. Adicione suporte para variáveis e um ambiente de execução, permitindo entradas como `x = 5` e depois `x + 3` (deve retornar `8`).
7. Implemente funções matemáticas como `sin`, `cos` e `sqrt`, permitindo expressões como `sqrt(16) + sin(0)`.

Casos de Uso Avançados

- **Compilador Simples**: Use o PLY para criar um compilador para uma linguagem de programação mínima, com suporte a variáveis, condicionais (`if`), loops (`while`) e funções.
- **Validador de Formato**: Construa um validador de JSON ou XML, verificando se a entrada segue as regras sintáticas do formato.
- **Interpretador de Scripts**: Crie um interpretador para uma linguagem de script personalizada, com suporte a comandos como `print`, `set` e `if`.
- **Análise de Logs**: Use o PLY para analisar logs estruturados, extraindo informações como timestamps, níveis de log e mensagens.

Dicas e Boas Práticas

- **Debugging**: Use a opção `debug=True` no `yacc.yacc(debug=True)` para gerar logs detalhados do parsing, úteis para depurar gramáticas complexas.
- **Organização**: Divida o código em seções claras (análise léxica, sintática, semântica) para facilitar manutenção.
- **Testes**: Crie casos de teste abrangentes, incluindo entradas válidas e inválidas, para garantir robustez.

- **Documentação**: Documente cada regra gramatical e token com comentários explicativos, especialmente em projetos grandes.
- **Performance**: Para entradas grandes, otimize expressões regulares e minimize o número de estados para melhorar o desempenho do lexer.

Conclusão

O PLY é uma ferramenta excepcionalmente poderosa e versátil para análise léxica e sintática em Python. Este material detalhou cada aspecto do PLY, desde a definição de tokens e gramáticas até funcionalidades avançadas como estados, precedência, tratamento de erros e recuperação. Com exemplos práticos, como o analisador de expressões aritméticas, e sugestões de expansão, o PLY pode ser aplicado a uma ampla gama de projetos, desde compiladores e interpretadores até ferramentas de validação e análise de dados. Os exercícios propostos incentivam a exploração de cenários mais complexos, enquanto os casos de uso avançados demonstram o potencial do PLY em aplicações reais. Com uma abordagem estruturada e boas práticas, o PLY pode ser uma peça central em sistemas que lidam com linguagens estruturadas.