UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE - UERN
FACULDADE DE CIÊNCIAS EXATAS E NATURAIS - FANAT
DOCENTE: Sebastião Emidio Alves Filho.
DISCENTES: Allany dos Santos Rodrigues, Fabio Bentes Tavares de Melo Junior, João Victor da Costa Gomes, Reinaldo Rogger Santos da Silva, Victor Manoel Soares da Silva.
DISCIPLINA: Compiladores e Paradigmas de Programação

TRABALHO TRANSPILADOR - 3° AVALIAÇÃO

15 de Dezembro de 2024
MOSSORÓ - RN

Sumário

1. INTRODUÇÃO

1.1. Introdução

Este trabalho propõe a implementação de um transpilador que converte programas escritos em Python para Ruby, com o objetivo de facilitar a migração e a integração de sistemas desenvolvidos nessas linguagens. A escolha dessas duas linguagens se justifica pela sua ampla utilização e características complementares, além do potencial de otimização e aprendizagem que esse processo oferece. A implementação do transpilador envolve o estudo de técnicas de análise sintática, semântica e otimização de código, além de proporcionar uma oportunidade para explorar as diferenças e semelhanças entre essas linguagens de alto nível. O trabalho busca, assim, contribuir para a compreensão e aplicação de conceitos fundamentais na área de compiladores e transpiladores, ao mesmo tempo que oferece uma ferramenta prática para desenvolvedores que precisam interagir com Python e Ruby em seus projetos.

1.2. JUSTIFICATIVA

A criação de um transpilador que converte programas escritos em Python para Ruby é relevante e importante por diversas razões. Primeiramente, ambas as linguagens são amplamente utilizadas em diferentes domínios, e facilitar a conversão entre elas pode aumentar a flexibilidade no desenvolvimento de software, permitindo que códigos escritos em Python sejam rapidamente adaptados para ambientes onde Ruby seja a linguagem preferida, como em muitas plataformas web e frameworks como Ruby on Rails.

Com um transpilador, é possível migrar ou integrar sistemas existentes de forma mais ágil, sem a necessidade de reescrever grandes trechos de código, economizando tempo e esforço. Isso proporciona uma maneira eficiente de adaptar softwares entre ambientes onde Python e Ruby são usados de forma intercambiável.

2. A LINGUAGEM DE ORIGEM

2.1. Descrição da Linguagem de Origem

Python é uma linguagem de programação de alto nível, criada por Guido van Rossum e lançada em 1991. Seu principal objetivo é ser simples e legível, com uma sintaxe clara que facilita a escrita e leitura do código. A linguagem é interpretada, o que permite um desenvolvimento mais rápido, e adota tipagem dinâmica, dispensando a declaração explícita de tipos de variáveis.

Python é multiparadigma, ou seja, suporta programação orientada a objetos, funcional e imperativa. Sua rica biblioteca padrão e um vasto ecossistema de bibliotecas externas cobrem áreas como desenvolvimento web, ciência de dados e automação. Além disso, é uma linguagem portátil, funcionando em diferentes sistemas operacionais sem a necessidade de ajustes no código.

A sintaxe de Python é organizada por indentação, o que elimina a necessidade de chaves ou palavras-chave para definir blocos de código, promovendo a clareza. Apesar de ser fácil de usar, Python pode ser mais lento que linguagens compiladas como C, e a tipagem dinâmica pode gerar problemas de desempenho em algumas situações. A implementação padrão, o CPython, também possui o GIL (Global Interpreter Lock), que limita a execução concorrente de threads.

3. A LINGUAGEM DE DESTINO

3.1. Descrição da Linguagem de Destino

Ruby é uma linguagem de programação de alto nível, dinâmica e orientada a objetos, criada por Yukihiro Matsumoto em 1995. Seu design foca na simplicidade e produtividade, com uma sintaxe natural que permite aos desenvolvedores escrever código de maneira concisa e elegante.

Em Ruby, quase tudo é um objeto, incluindo tipos primitivos como números e strings. A linguagem é dinâmica, com tipagem determinada em tempo de execução, o que proporciona flexibilidade, mas pode gerar erros difíceis de identificar antecipadamente.

Ruby é multiparadigma, suportando programação orientada a objetos, funcional e imperativa. Oferece recursos como blocos de código, lambdas e métodos integrados para manipulação eficiente de coleções como arrays, hashes e strings.

4. ANÁLISE LÉXICA

4.1. Tokens Suportados

```
# tokens.py
tokens = (
    'IDENTIFIER', 'EQUALS', 'NUMBER', 'FLOAT', 'STRING', 'COMMENT',
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN',
    'TRUE', 'FALSE', 'AND', 'OR', 'PRINT',
    'IF', 'ELSE', 'FOR', 'WHILE', 'DEF', 'COLON', 'IN',
    'COMMA',
    'GT', 'LT', 'GE', 'LE', 'EQ', 'NE',
    'PLUS_EQUALS', 'MINUS_EQUALS', 'TIMES_EQUALS', 'DIVIDE_EQUALS'

)
```

4.2. Literais e Tipos de Dados Suportados

Tipos de Dados Reconhecidos

Os tipos de dados básicos reconhecidos pelo lexer e parser são:

1. **Numéricos:**
   - NUMBER: números inteiros (e.g., 42).
   - FLOAT: números de ponto flutuante (e.g., 3.14).
2. **Booleanos:**
   - TRUE e FALSE: reconhecidos e convertidos para true e false no código Ruby.
3. **Strings:**
   - STRING: suporta strings com aspas simples ou duplas, incluindo caracteres escapados (definição detalhada no lexer.py).

Literais Suportados

Os operadores e palavras reservadas definidos em tokens.py e utilizados no lexer.py incluem:

- **Operadores aritméticos:** PLUS, MINUS, TIMES, DIVIDE.
- **Operadores de comparação:** GT (>), LT(<),GE(>=),LE(<=),EQ(==),NE` (!=).
- **Operadores compostos:** PLUS_EQUALS (+=), MINUS_EQUALS (-=), etc.
- **Delimitadores:** COLON (:), COMMA (,).

4.3. Palavras Reservadas

As palavras reservadas, listadas em tokens.py, não podem ser usadas como identificadores. Elas incluem:

- **Controle de fluxo:**
  - IF, ELSE, WHILE, FOR.
- **Funções:**
  - DEF, IN.
- **Operadores lógicos:**
  - AND, OR.
- **Exibição:**
  - PRINT.

Essas palavras são tratadas como tokens no lexer.py com funções específicas que garantem que sejam corretamente interpretadas.

5. GRAMÁTICA UTILIZADA NO RECONHECIMENTO DOS COMANDOS

5.1. Definição da Gramática

A gramática é definida no parser.py usando o módulo ply.yacc. Aqui estão as principais definições:

- **Regras Sintáticas Gerais:**
  - A gramática começa com a produção inicial program, que pode conter várias statements (declarações).
  - Cada statement pode ser uma atribuição, um laço, um condicional, ou mesmo um comentário.
- **Regras para Operadores:**
  - Precedência é definida no topo do arquivo (precedence), especificando a ordem de resolução de operadores, com suporte para operadores unários (UMINUS).
- **Exemplos de Produções:**

Atribuição:
statement : IDENTIFIER EQUALS expression

- **Exemplo: x = 42 → x = 42 (em Ruby).**

Operadores compostos:
statement : IDENTIFIER PLUS_EQUALS expression

- **Exemplo: x += 1 → x += 1.**

Estruturas de controle:
statement : WHILE expression COLON statements
Exemplo:

```
while x < 10:
   x += 1
```

Se torna:

```
while x < 10
  x += 1
end
```

Condicionais:
statement : IF expression COLON statements ELSE COLON statements
Exemplo:

```
if x > 10:
   print(x)
else:
   print(0)
```

Se torna:

```
if x > 10
  puts x
else
  puts 0
end
```

5.2. Exemplos de Regras Sintáticas

A gramática cobre uma ampla gama de construções Python, traduzindo-as para Ruby. Exemplos incluem:

Funções:
statement : DEF IDENTIFIER LPAREN params RPAREN COLON statements
Exemplo:

```
def soma(a, b):
   return a + b
```

Se torna:

```
def soma(a, b)
  return a + b
end
```

Laços for:
statement : FOR IDENTIFIER IN expression COLON statements

Exemplo:

```
for i in range(5):
    print(i)
```

Se torna:

```
for i in range(5)
  puts i
end
```

6. CÓDIGO DO LEXER.PY

```
# lexer.py

import ply.lex as lex
from tokens import tokens

# Operadores de comparação
t_GE = r'>='
t_LE = r'<='
t_EQ = r'=='
t_NE = r'!='

# Operadores compostos
t_PLUS_EQUALS = r'\+='
t_MINUS_EQUALS = r'-='
t_TIMES_EQUALS = r'\*='
t_DIVIDE_EQUALS = r'/='

# Operadores de um único caractere
t_GT = r'>'
t_LT = r'<'
t_EQUALS = r'='
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_COMMA = r','

# Delimitadores
t_COLON = r':'

# Definição de strings com suporte a caracteres escapados
t_STRING = r'\"([^\\\"]|\\.)*\"|\'([^\\\']|\\.)*\''
```

```
# Identificadores
t_IDENTIFIER = r'[a-zA-Z_][a-zA-Z_0-9]*'

# Literais numéricos
t_FLOAT = r'\d+\.\d+'
t_NUMBER = r'\d+'

# Comentários
t_COMMENT = r'\#.*'

# Palavras-chave definidas como funções para retornar o token correspondente
def t_PRINT(t):
    r'print'
    return t

def t_IF(t):
    r'if'
    return t

def t_ELSE(t):
    r'else'
    return t

def t_FOR(t):
    r'for'
    return t

def t_WHILE(t):
    r'while'
    return t

def t_DEF(t):
    r'def'
    return t

def t_IN(t):
    r'in'
    return t

def t_TRUE(t):
    r'True'
```

```
    t.value = 'true'
    return t

def t_FALSE(t):
    r'False'
    t.value = 'false'
    return t

def t_AND(t):
    r'and'
    t.value = '&&'
    return t

def t_OR(t):
    r'or'
    t.value = '||'
    return t

# Regra para quebras de linha
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# Caractere a serem ignorados (espaços e tabulações)
t_ignore = ' \t'

# Regra para tratamento de erros
def t_error(t):
    print(f"Caractere inválido '{t.value[0]}' na linha {t.lineno}")
    t.lexer.skip(1)

# Construção do lexer
lexer = lex.lex()
```

7. CÓDIGO DO PARSER.PY

```
# parser.py

import ply.yacc as yacc
from tokens import tokens
```

```
# Definição de precedência para operadores
precedence = (
    ('left', 'OR'),
    ('left', 'AND'),
    ('left', 'EQ', 'NE', 'GT', 'LT', 'GE', 'LE'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),  # Suporte para operador unário
)

# Tabela de símbolos para rastrear variáveis definidas
symbol_table = set()

# Regra inicial
def p_program(p):
    '''program : statements'''
    p[0] = p[1]

# Regras para múltiplas declarações
def p_statements_multiple(p):
    '''statements : statements statement'''
    p[0] = f"{p[1]}\n{p[2]}"

def p_statements_single(p):
    '''statements : statement'''
    p[0] = p[1]

# Regras para atribuição ou print
def p_statement_assign_or_print(p):
    '''statement : IDENTIFIER EQUALS expression
                 | PRINT LPAREN expression RPAREN'''
    if p[1] == 'print':
        p[0] = f"puts {p[3]}"
    else:
        symbol_table.add(p[1])  # Adiciona a variável à tabela de símbolos
        p[0] = f"{p[1]} = {p[3]}"

# Regras para operadores compostos (e.g., +=)
def p_statement_compound_assign(p):
    '''statement : IDENTIFIER PLUS_EQUALS expression
                 | IDENTIFIER MINUS_EQUALS expression
                 | IDENTIFIER TIMES_EQUALS expression
```

```
                | IDENTIFIER DIVIDE_EQUALS expression'''
    var = p[1]
    operador = p[2][0]  # '+' de '+=', '-' de '-=', etc.
    if var not in symbol_table:
        print(f"Warning: Variável '{var}' usada antes de ser definida na linha
{p.lineno(1)}")
    p[0] = f"{var} {operador}= {p[3]}"

# Regras para comentários
def p_statement_comment(p):
    '''statement : COMMENT'''
    # Transforma o comentário de Python (# ...) em comentário de Ruby (# ...)
    p[0] = p[1]

# Regras para expressões lógicas
def p_expression_or(p):
    '''expression : expression OR and_expression'''
    p[0] = f"{p[1]} || {p[3]}"

def p_expression_and(p):
    '''and_expression : and_expression AND comparison'''
    p[0] = f"{p[1]} && {p[3]}"

def p_expression_and_single(p):
    '''and_expression : comparison'''
    p[0] = p[1]

def p_expression_and_expression(p):
    '''expression : and_expression'''
    p[0] = p[1]

# Regras para comparações
def p_comparison_gt(p):
    '''comparison : arithmetic_expression GT arithmetic_expression'''
    p[0] = f"{p[1]} > {p[3]}"

def p_comparison_lt(p):
    '''comparison : arithmetic_expression LT arithmetic_expression'''
    p[0] = f"{p[1]} < {p[3]}"

def p_comparison_ge(p):
    '''comparison : arithmetic_expression GE arithmetic_expression'''
```

```
    p[0] = f"{p[1]} >= {p[3]}"

def p_comparison_le(p):
    '''comparison : arithmetic_expression LE arithmetic_expression'''
    p[0] = f"{p[1]} <= {p[3]}"

def p_comparison_eq(p):
    '''comparison : arithmetic_expression EQ arithmetic_expression'''
    p[0] = f"{p[1]} == {p[3]}"

def p_comparison_ne(p):
    '''comparison : arithmetic_expression NE arithmetic_expression'''
    p[0] = f"{p[1]} != {p[3]}"

def p_comparison_arithmetic(p):
    '''comparison : arithmetic_expression'''
    p[0] = p[1]

# Regras para expressões aritméticas
def p_arithmetic_expression_plus(p):
    '''arithmetic_expression : arithmetic_expression PLUS term'''
    p[0] = f"{p[1]} + {p[3]}"

def p_arithmetic_expression_minus(p):
    '''arithmetic_expression : arithmetic_expression MINUS term'''
    p[0] = f"{p[1]} - {p[3]}"

def p_arithmetic_expression_uminus(p):
    '''arithmetic_expression : MINUS arithmetic_expression %prec UMINUS'''
    p[0] = f"-{p[2]}"

def p_arithmetic_expression_term(p):
    '''arithmetic_expression : term'''
    p[0] = p[1]

# Regras para termos
def p_term_times(p):
    '''term : term TIMES factor'''
    p[0] = f"{p[1]} * {p[3]}"

def p_term_divide(p):
    '''term : term DIVIDE factor'''
```

```
    p[0] = f"{p[1]} / {p[3]}"

def p_term_factor(p):
    '''term : factor'''
    p[0] = p[1]

# Regras para fatores
def p_factor_number(p):
    '''factor : NUMBER
              | FLOAT
              | STRING
              | TRUE
              | FALSE'''
    p[0] = p[1]

def p_factor_identifier(p):
    '''factor : IDENTIFIER'''
    if p[1] not in symbol_table:
        print(f"Warning: Variável '{p[1]}' usada antes de ser definida na
linha {p.lineno(1)}")
    p[0] = p[1]

def p_factor_expr(p):
    '''factor : LPAREN expression RPAREN'''
    p[0] = f"({p[2]})"

def p_factor_call(p):
    '''factor : IDENTIFIER LPAREN args RPAREN'''
    p[0] = f"{p[1]}({p[3]})"

# Regras para argumentos de funções
def p_args_multiple(p):
    '''args : expression COMMA args'''
    p[0] = f"{p[1]}, {p[3]}"

def p_args_single(p):
    '''args : expression'''
    p[0] = p[1]

def p_args_empty(p):
    '''args : empty'''
    p[0] = ''
```

```
# Adicionar regra para tratar expressões como declarações
def p_statement_expression(p):
    '''statement : expression'''
    p[0] = p[1]

# Regras para condicionais
def p_statement_if_else(p):
    '''statement : IF expression COLON statements ELSE COLON statements'''
    p[0] = f"if {p[2]}\n  {indent(p[4])}\nelse\n  {indent(p[7])}\nend"

def p_statement_if(p):
    '''statement : IF expression COLON statements'''
    p[0] = f"if {p[2]}\n  {indent(p[4])}\nend"

# Regras para loops while
def p_statement_while(p):
    '''statement : WHILE expression COLON statements'''
    p[0] = f"while {p[2]}\n  {indent(p[4])}\nend"

# Regras para loops for
def p_statement_for(p):
    '''statement : FOR IDENTIFIER IN expression COLON statements'''
    p[0] = f"for {p[2]} in {p[4]}\n  {indent(p[6])}\nend"

# Regras para definição de funções
def p_statement_function(p):
    '''statement : DEF IDENTIFIER LPAREN params RPAREN COLON statements'''
    p[0] = f"def {p[2]}({p[4]})\n  {indent(p[7])}\nend"

# Regras para parâmetros de funções
def p_params_multiple(p):
    '''params : IDENTIFIER COMMA params'''
    p[0] = f"{p[1]}, {p[3]}"

def p_params_single(p):
    '''params : IDENTIFIER'''
    p[0] = p[1]

def p_params_empty(p):
    '''params : empty'''
    p[0] = ''
```

```python
# Função auxiliar para indentação
def indent(text, num_spaces=2):
    indented = ""
    for line in text.split('\n'):
        indented += ' ' * num_spaces + line + '\n'
    return indented.rstrip()

# Produção vazia
def p_empty(p):
    '''empty :'''
    p[0] = ''

# Regra de erro
def p_error(p):
    if p:
        print(f"Erro de sintaxe na linha {p.lineno} perto de '{p.value}'")
    else:
        print("Erro de sintaxe no EOF")

# Construção do parser
parser = yacc.yacc(debug=False, write_tables=False)
```