Fundação Edson Queiroz
Universidade de Fortaleza
Centro de Ciências Tecnológicas

Aspectos Teóricos da Computação

Compilador entre Linguagens de Programação de Alto Nível (Transpilador)

Ana Beatriz Silva Ferreira – 2110915

Eygon Lucas Peixoto Saldanha – 2110880

Ana Laura Viana Geleilate -1910430

Isabele Silva Mesquita – 2110911

Gabriel Alves Matos – 2110945

Samuel Lincoln Magalhães Barrocas

Junho/2024

1. O Projeto:

O projeto em questão tem como objetivo principal a implementação de um transpilador, uma ferramenta capaz de traduzir programas escritos em uma linguagem de programação criada especificamente para este projeto para código equivalente em Python. A linguagem própria foi concebida com o intuito de oferecer uma sintaxe simplificada e intuitiva, permitindo aos desenvolvedores expressar lógicas de programação de alto nível de maneira eficiente e legível.

1.1 Sobre a Linguagem:

Para ilustrar a utilidade e a sintaxe da linguagem própria, apresentamos alguns exemplos comparativos entre código Python padrão e código na nossa linguagem:

- **Instruções de Entrada e Saída**

  Python:

```python
nome = input("Digite seu nome: ")
print("Olá ", nome)
```

  Nossa Linguagem:

```
get txt:nome by "Digite seu nome: ";
show "Olá " nome;
```

- **Lógicas de Laço**

  Python:

```python
contador = 0
while contador < 5:
    print(f"Contagem: {contador}")
    contador += 1

for i in range(1, 6):
    print(f"Valor: {i}")
```

  Nossa Linguagem:

```
int:contador = 0;
while contador < 5 do
    show "Contagem: " contador;
    contador += 1;
end

for i in range(1 to 6) do
    show "Valor: " i;
end
```

- **Estruturas de Controle**

  Python:

```python
if values.vigencia and values.corrente:
    ativa = True
else:
    ativa = False
```

  Nossa Linguagem:

```
having values.vigencia AND values.corrente not do
    ativa = True
either
    ativa = False
end
```

- **Declaração de Funções/Métodos**

  Python:

```python
def cpfValido(cpf, nome):
    return len(cpf) == 11

print(cpfValido(cpf, nome))
```

  Nossa Linguagem:

```
handler bool:cpfValido with txt:cpf and txt:nome doing
    return len(cpf) == 11
end

show cpfValido(values.cpf, values.nome)
```

- **Expressões Aritméticas e Booleanas**

  Python:

```python
x = 5
y = 3
maior = x > y
igual = x == y
print(f"x é maior que y? {maior}, x é igual a y? {igual}")
```

  Nossa Linguagem:

```
int:x = 5
int:y = 3
bool:maior = x > y
bool:igual = x == y
show "x é maior que y?" maior "x é igual a y?" igual;
```

1.2 Sobre a Gramática:

A gramática é fundamental em linguagens de programação e compiladores/transpiladores porque define as regras sintáticas que determinam como os programas devem ser estruturados e escritos para serem corretamente interpretados ou compilados.

A gramática da nossa linguagem própria utilizada no transpilador é definida da seguinte forma:

```
VariableDeclaration = Type ident "=" Expr ";"
FunctionDeclaration = Type ident Params Block
Params = Type ident {"," Type ident}
Block = "{" {Statement}"}"
Statement = Designator ("=" Expr | ActPars) ";"
       | "having" Condition "then" Statement ["else" Statement]
       | "as" Condition "do" Statement
       | "return" [Expr]";"
       | "read" Designator ";"
       | "print" Expr ["," number] ";"
       | Block
       | ";"
ActPars = [ Expr { "," Expr } ]
Condition  = Expr Relop Expr
Relop = "=="| "!=" |">" |">=" |"<" |"<="
Expr = ["-"] Term {Addop Term}
Term = Factor {Mullop Factor}
Factor = Designator [ActPars]
      | number
      | charConst
      | "new" ident [ Expr ]
      | Expr
Designator = ident { "." ident | Expr  }
Addop = "+" | "-"
Mullop = "*" | "/" | " % "
Type = "int" | "boolean" | "text" | "number"| ident
```

A gramática é crucial para o analisador sintático em nosso projeto de transpilador. Enquanto o analisador léxico identifica e classifica tokens individuais, o analisador sintático utiliza a gramática para estruturar esses tokens em uma árvore sintática que representa a estrutura hierárquica do código fonte. A gramática define as regras sintáticas que especificam como os tokens podem ser combinados para formar construções maiores, como declarações de variáveis, definições de funções, estruturas de controle e expressões matemáticas.

1.3 Sobre o PLY:

Nosso grupo decidiu utilizar a ferramenta PLY (Python Lex-Yacc) para construir analisadores léxicos e sintáticos em Python. Ele combina a funcionalidade de duas ferramentas clássicas, Lex e Yacc, que são amplamente usadas em compilação e linguagens formais. A importância do PLY para nosso projeto de transpilador pode ser descrita em várias frentes:

- **Definição da Gramática**: O PLY permite que definamos a gramática completa da nossa linguagem própria usando regras claras e concisas. Isso inclui a especificação de tokens como identificadores, números, strings, operadores, delimitadores e palavras-chave.

- **Análise Léxica**: O PLY facilita a criação de um analisador léxico robusto. Utilizando expressões regulares e funções Python para cada tipo de token, podemos eficientemente identificar e classificar diferentes partes do código fonte da nossa linguagem

- **Análise Sintática**: Com a gramática definida e os tokens identificados, o PLY nos permite implementar o analisador sintático que verifica a estrutura hierárquica do código fonte. Podemos definir regras gramaticais claras para cada construção sintática da nossa linguagem, como declarações de variáveis, expressões aritméticas, estruturas de controle e chamadas de funções. O PLY verifica se o código fonte segue corretamente essas regras gramaticais, garantindo assim que os programas escritos na nossa linguagem sejam interpretados corretamente e transpilados para Python de maneira adequada

- **Transformação para Python**: Uma vez que a análise léxica e sintática é concluída com sucesso, o PLY nos dá a capacidade de gerar código Python equivalente. Podemos definir ações semânticas associadas a cada regra gramatical para construir a representação em Python correspondente à estrutura analisada da nossa linguagem. Isso envolve a geração de código Python que capture a lógica e a funcionalidade originalmente expressas na linguagem própria.

2.Análise de Complexidade Assintótica do Lexer

A análise de complexidade se concentra em funções críticas do lexer, avaliando seu desempenho tanto em termos de tempo (complexidade temporal) quanto em termos de uso de memória (complexidade espacial). A seguir, descrevemos as funções principais e suas complexidades**.**

- **Funções Analisadas**

Função t_IDENTIFIER

```
def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = reserved.get(t.value, 'IDENTIFIER')
    return t
```

Complexidade Temporal: O(n), onde n é o comprimento do identificador.

Complexidade Espacial: O(n), para armazenar o identificador.

Função t_FLOAT

```
def t_FLOAT(t):
    r'\d+\.\d+'
    t.value = float(t.value)
    return t
```

Complexidade Temporal: A expressão regular `\d+\.\d+` tem complexidade O(n), onde n é o número de dígitos no número de ponto flutuante.

Complexidade Espacial: A função converte a string para um valor de ponto flutuante, usando O(1) espaço adicional.

Função `t_NUMBER`

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Complexidade Temporal: A expressão regular `\d+` tem complexidade O(n), onde n é o número de dígitos no número inteiro.

Complexidade Espacial: A função converte a string para um valor inteiro, usando O(1) espaço adicional.

Função `t_CHARCONST`

```
def t_CHARCONST(t):
    r'\'([^\\\n]|(\\.))*?\''
    t.value = t.value.strip("'")
    return t
```

Complexidade Temporal: A expressão regular `\'([^\\\n]|(\\.))*?\'` pode ser complexa, mas geralmente é processada linearmente O(n), onde n é o comprimento do caractere constante.

Complexidade Espacial: A função remove as aspas da string, resultando em O(n) espaço para armazenar a string resultante.

Função `t_error`

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

Complexidade Temporal: A função `t_error` é chamada para caracteres ilegais. Ela imprime uma mensagem de erro e avança um caractere, resultando em O(1) tempo.

Complexidade Espacial: A função usa O(1) espaço adicional.

Sugestões de Otimização:

- Reduzir o número de conversões de tipos (ex.: string para int/float) ao mínimo necessário.
- Compilar expressões regulares antes da análise pode melhorar a performance.
- Implementar uma estratégia de gestão de erros que minimize operações de I/O

3.Referências:

- ChatGPt: auxiliou na estrutura dos arquivos do projeto e como base para fase das compilações.
- Material Complementar do Professor: https://drive.google.com/drive/u/0/folders/1cBrwORfPEHvnxj3CH1w8KH_9kjC79eB4