 <p>ESCOLA SUPERIOR DE TECNOLOGIA IPCA</p>	<p>COMUNICAÇÕES DE DADOS</p>	<p>2023/2024</p>
--	------------------------------	------------------

Relatório de Processamento de Linguagens

LESI – PL



TRABALHO PRÁTICO – ENTREGA 2

Joaquim Peixoto Nº18459


João Pereira Nº20345

Flávio Costa Nº20349

Instituto Politécnico do Cávado e do Ave


2023/2024

Barcelos

 ESCOLA SUPERIOR DE TECNOLOGIA IPCA	COMUNICAÇÕES DE DADOS	2023/2024
--	------------------------------	------------------

Índice

Introdução	3
Introdução ao problema	4
Desenvolvimento	5
Especificação da Gramática concreta da linguagem de entrada	5
Criação da gramática	8
Leitura e mostrar dados	10
Problemas encontrados	13
Objetivos atingidos	13
Testes	13
Conclusão	14

 ESCOLA SUPERIOR DE TECNOLOGIA IPCA	COMUNICAÇÕES DE DADOS	2023/2024
--	------------------------------	------------------

INTRODUÇÃO

No âmbito da Unidade Curricular de Processamento de Linguagens, foi proposto a realização de um trabalho de grupo com base na matéria lecionada em sala de aula.

Este trabalho consiste na implementação de uma aplicação para o processamento de uma linguagem funcional, neste caso, chamada de Linguagem Funcional do Cávado e do Ave (FCA).

Esta aplicação deve ser capaz de ler ficheiros de texto com um programa escrito na linguagem mencionada e, após processar a informação, executar os comandos contidos nesse ficheiro.

O grupo, após a leitura do enunciado, procurou nos recursos fornecidos pelo professor o material necessário para facilitar o processo de realização do trabalho, obtendo assim uma base sólida para o desenvolvimento do trabalho. Sempre que surgiam dúvidas eram consultados alguns websites para esclarecer dúvidas.

INTRODUÇÃO AO PROBLEMA

Neste problema, o principal foco é que os alunos obtenham mais valências, quer académicas quer profissionais, na área de definição de analisadores léxicos e sintáticos e de ações semânticas.

Desta forma, dividimos o problema em 6 fases:

- Especificar a gramática concreta da linguagem de entrada;
- Desenvolver um reconhecedor léxico, com a biblioteca de *python lex* para reconhecer os símbolos terminais e iniciais da gramática;
- Desenvolver um reconhecedor sintático, com a biblioteca de *python yacc* para reconhecer a gramática concreta;
- Planear uma árvore de sintaxe abstrata para representar a linguagem de entrada e associar ações semânticas de tradução às produções da gramática de forma a construir a correspondente árvore de sintaxe abstrata;
- Desenvolver o gerador de código que se produza a resposta solicitada, através da avaliação da árvore de sintaxe abstrata.
- Testar o programa desenvolvido, através dos exemplos fornecidos.

DESENVOLVIMENTO

Depois de ler o enunciado que o professor nos deu, concluímos que a maior parte do que precisávamos aplicar na tarefa foi abordado em aula. Resolvemos então revisar alguns dos exercícios resolvidos pelo professor para fornecer o material e, portanto, já recebemos uma base para iniciar nosso trabalho.

Para iniciar o desenvolvimento, começamos pela leitura do enunciado, registrando quais seriam os pontos mais importantes para nós e buscando soluções prioritárias, portanto, com base nas regras estabelecidas pelo professor, começamos a especificar a gramática concreta da língua de entrada.

ESPECIFICAÇÃO DA GRAMÁTICA CONCRETA DA LINGUAGEM DE ENTRADA

A especificação da gramática é como se fosse uma receita que ensina ao computador como entender corretamente as palavras e símbolos num texto.

Desta forma, o computador consegue identificar números, palavras e símbolos especiais da linguagem que está a tentar interpretar.

Podemos dizer que é como ensinar ao computador as regras básicas de como a linguagem funciona. Abaixo, conseguimos consultar uma breve explicação da nossa especificação da gramática:

- Importação da Biblioteca *ply.lex* para criar analisadores léxicos em *Python*, importação da *Lang* (*tokens*, *literals* e mais algumas coisas necessárias)

```
✓ import ply.lex as lex  
from lang import Lang
```

- Ficheiro *Lang*, com a definição dos *tokens*, (o tipo de tokens que o *lexer* pode reconhecer), *literals* (caracteres que são reconhecidos como operadores aritméticos, pontos, vírgulas, etc.) e um *to_ignore* e *t_multilinecomment_ignore*,

```
class Lang:
    tokens = (
        "NUM",
        "VARIABEL",
        "STR",
        "FUNCAO",
        "FIM",
        "NAME",
        "ALEATORIO",
        "ESCREVER",
        "ENTRADA",
        "CONCAT",
    )
    literals = (
        "(",
        ")",
        "*",
        "+",
        "=",
        ",",
        ";",
        ":",
        "?",
        "!",
        "_",
        "-",
        "{",
        "}",
        "<",
        ">",
        "#",
        "[",
        "]",
    )
    t_ignore = " \n\t"
    t_multilinecomment_ignore = "\t"
```

- Definição das expressões regulares para os tokens, que são todos os métodos que começam por “t_” e que definem a forma como os tokens são reconhecidos e tratados para descrever padrões de texto.

```
def t_ComandosIniciais(self, t):
    r"ESCREVER"
    t.type = t.value
    return t

def t_ENTRADA(self, t):
    r"ENTRADA"
    t.type = t.value
    return t

def t_ALEATORIO(self, t):
    r"ALEATORIO"
    t.type = t.value
    return t

def t_NUM(self, t):
    r"[0-9]+"
    t.value = int(t.value)
    return t

def t_STR(self, t):
    r'"([^\"]|\\")*"'
    t.value = t.value[1:-1]
    t.value = t.value.replace('\\"', '"')
    return t

def t_VARIAVEL(self, t):
    r"[a-z_][a-zA-Z0-9_]*(?:[!]?)"
    if t.value in self.function_names:
        t.type = self.function_names[t.value]
    return t

def t_NAME(self, t):
    r"[a-z][a-z0-9]*"
    if t.value in self.function_names:
        t.type = self.function_names[t.value]
    return t

def t_FUNCAO(self, t):
    r"FUNCAO"
    t.type = t.value
    return t

def t_FIM(self, t):
    r"FIM"
    t.type = t.value
    return t

def t_newline(self, t):
    r"\n+"
    t.lexer.lineno += len(t.value)

def t_COMMENT(self, t):
    r"\-\.*"
    pass

def t_multilinecomment(self, t):
    r"\{\- "
    t.lexer.begin("multilinecomment")
    pass

def t_multilinecomment_content(self, t):
    r"^[^\-]+"
    pass

def t_multilinecomment_end(self, t):
    r"\-\}"
    t.lexer.begin("INITIAL")

def t_multilinecomment_newline(self, t):
    r"\n+"
    t.lexer.lineno += len(t.value)

def t_multilinecomment_error(self, t):
    t.lexer.skip(1)

def t_error(self, t):
    print(f"Unexpected token: [{t.value[0]}]")
    t.lexer.skip(1)
```

- Métodos “*Build*” (cria o lexer com as regras definidas), método “*input*” (fornece a entrada para o *lexer*) e “*token*” (obtem o próximo *token* de entrada)

```
def build(self) → None:
    self.lexer = lex.lex(module=self)

def input(self, string: str) → None:
    self.lexer.input(string)

def token(self):
    return self.lexer.token()
```

CRIAÇÃO DA GRAMÁTICA

O próximo passo é a criação de uma gramática para a linguagem em questão, usando a biblioteca ‘*ply.yacc*’. É fundamental existir a gramática para definir a estrutura válida da linguagem, permitindo assim a interpretação correta do código.

- Neste ficheiro importamos o *ArithLexer* que é o lexer que nós criamos, importamos o *PrettyPrinter* que é uma classe usada para mostrar estruturas de dados mais legíveis, importamos ainda o nosso *utils* que tem algumas funções para ler ficheiros e fazer algumas verificações e substituições, o *random* para gerar números aleatórios e por fim o *yacc*, usado para construir analisadores léxicos.

```
import re
import os

class Utils:
    @staticmethod
    def read_lang_file(file_path):
        if os.path.exists(file_path) == False:
            raise Exception(f"File {file_path} not found.")

        (method def replace_interpolation( tf-8") as file:
            text: Self@Utils,
            variables: Any
        def ) → (Self@Utils | Any)
            replace_interpolation
            ', text)

    def replace_interpolation(text, variables):
        for key in variables:
            text = text.replace(f"#{key}", str(variables[key]))

        return text
```

utils.py


```
from lexer import ArithLexer
import ply.yacc as yacc
from pprint import PrettyPrinter
import random
from utils import Utils

pp = PrettyPrinter()

class ArithGrammar:
    tokens = ArithLexer.tokens

    _variables = {}

    precedence = (
        ("left", "CONCAT"),
        ("left", "+", "-"),
        ("left", "*", "/"),
        ("right", "UMINUS"),
    )

    def p_Rec0(self, p):
        "Rec : Start"
        p[0] = [p[1]]

    def p_Rec1(self, p):
        "Rec : Rec Start"
        p[0] = p[1]
        p[0].append(p[2])

    def p_Start(self, p):
        """Start : ComandosIniciais"""
        p[0] = p[1]
```

```
def p_escrever(self, p):
    "escrever : '(' expression '"'"
    p[0] = p[2]

def p_variavel(self, p):
    """variavel : VARIABEL '=' expression"""
    if len(p) == 4:
        self._variaveis[p[1]] = p[3]
        p[0] = {"op": "declare", "var_name": p[1], "value": p[3]}

def p_expression(self, p):
    """
    expression : expression_ops
    | STR
    | VARIABEL
    | func_call
    | list
    """
    if isinstance(p[1], str) and p[1] in self._variaveis:
        p[0] = self._variaveis[p[1]]
    elif p[1] is None:
        p[0] = 0
    else:
        p[0] = p[1]
```

Alguns exemplos do grammar.py

- Aqui está definido o método Build para construir o analisador sintático:

```
def build(self) → None:  
    self.lexer: ArithLexer = ArithLexer()  
    self.lexer.build()  
    self.yacc: yacc.LRParser = yacc.yacc(module=self)
```

- Por fim é definido o parse para analisar a entrada de acordo com a gramática definida. Recebe os dados de entrada e as variáveis e retorna a estrutura de dados resultante da análise sintática

LEITURA E MOSTRAR DADOS

Para podermos ler os ficheiros, usar a gramática e interpretar o código inserido. Recebe o nome do ficheiro na linha de comandos, lê o ficheiro, da Build na grammar e dá parse no texto e variáveis.

Por fim mostra o resultado interpretado e gera o código em C, mostrando também o código em C.

```
import sys
from utils import Utils
from grammar import ArithGrammar
from codegen import CodeGen
from interpreter import Interpreter

def main():
    variaveis = {}
    if len(sys.argv) != 2:
        print("Usage: python main.py <input_file>")
        return

    input_file = sys.argv[1]
    data = Utils.read_lang_file(input_file)

    parser = ArithGrammar()
    parser.build()
    interpreter = Interpreter()
    codegen = CodeGen()

    ast = parser.parse(data, variaveis)

    if ast is not None:
        # Interpret the input
        print("AST:")
        print(ast)
        print("\nInterpreted Output:")
        interpreter = Interpreter()
        interpreter.eval(ast)

        # Generate Python code
        generated_code = codegen.generate(ast)
        print("\nGenerated C Code:")
        print(generated_code)
    else:
        print("Parsing failed.")

if __name__ == "__main__":
    main()
```

Aqui temos o codegen usado para transformar o código em C.


```
class CodeGen:
    def __init__(self):
        self.code = []
        self.variables = {}

    def generate(self, parsed_data):
        self.code.append("#include <stdio.h>")
        self.code.append("int main() {")
        for statement in parsed_data:
            self.code.append("    " + self._generate_statement(statement) + ";")
        self.code.append("    return 0;")
        self.code.append("}")
        return "\n".join(self.code)

    def _generate_statement(self, statement):
        if statement['op'] == 'print':
            return f'printf("d\\n", {self._generate_expression(statement["value"])})'
        elif statement['op'] == 'declare':
            if statement['var_name'] not in self.variables:
                self.variables[statement['var_name']] = "int"
            return f'int {statement["var_name"]} = {self._generate_expression(statement["value"])}'
        elif statement['op'] == 'func_declare':
            func_args = ", ".join(f'int {arg}' for arg in statement['args'])
            func_body = "\n    ".join(self._generate_statement(line) for line in statement['body'])
            return f'int {statement["func_name"]}({func_args}) {{\n        {func_body};\n    }}'
        elif statement['op'] == 'map':
            return f'map({statement["func"]}, {self._generate_expression(statement["list"])})'
        elif statement['op'] == 'fold':
            return f'fold({statement["func"]}, {self._generate_expression(statement["initial"])}, {self._generate_expression(statement["list"])})'
        return ""

    def _generate_expression(self, expression):
        if isinstance(expression, dict):
            if expression['op'] == 'UMINUS':
                return f'!-{self._generate_expression(expression["value"])}'
            return f'({self._generate_expression(expression["left"])} {expression["op"]} {self._generate_expression(expression["right"])})'
        return str(expression)
```

jhpixoto, 1 hour ago - Removed parser.out file from the repository

 <p>ESCOLA SUPERIOR DE TECNOLOGIA IPCA</p>	<p>COMUNICAÇÕES DE DADOS</p>	<p>2023/2024</p>
--	------------------------------	------------------

Problemas encontrados

Durante o desenvolvimento do trabalho, enfrentamos alguns problemas derivados do conhecimento limitado sobre a matéria do trabalho, principalmente na parte do map e fold, mas com alguma pesquisa e conseguimos ultrapassar.

Outro problema foi no codegen que também tivemos alguma dificuldade a desenvolver essa parte.


OBJETIVOS ATINGIDOS

No que toca aos objetivos atingidos, pensamos que conseguimos corresponder às expectativas do pedido do trabalho e até superar as nossas expectativas sendo que no início pensamos que seria muito complicada a conclusão do mesmo.

Como objetivo tínhamos a melhoria dos nossos conhecimentos, o que sem dúvida foi atingido.

TESTES


No que toca aos testes, optamos por fazer testes automatizados para não termos de escrever sempre o nome dos ficheiros todos um de cada vez. Para isso basta correr o código “python3 tests.py” e na pasta output aparecem os ficheiros com a resposta a cada um deles.

 <p>ESCOLA SUPERIOR DE TECNOLOGIA IPCA</p>	<p>COMUNICAÇÕES DE DADOS</p>	<p>2023/2024</p>
--	------------------------------	------------------

CONCLUSÃO

Com a realização deste trabalho, concluímos que possuíamos poucos conhecimentos em python e esta foi uma forma enriquecedora para a aquisição de experiência nesta linguagem e neste tema.

Sentimos alguma dificuldade, mas, com algum esforço e pesquisa, sem nunca desistir, conseguimos ultrapassar e ganhar mais conhecimentos, transformando estes aprendizados em ensinamentos uteis no nosso futuro.

 <p>ESCOLA SUPERIOR DE TECNOLOGIA IPCA</p>	<p>COMUNICAÇÕES DE DADOS</p>	<p>2023/2024</p>
--	------------------------------	------------------

Webgrafia

- Dabeaz.com: <https://www.dabeaz.com/ply/ply.html>
- Ply.readthedocs: <https://ply.readthedocs.io/en/latest/>
- StackOverflow: <https://stackoverflow.com/>