INSTITUTO POLITÉCNICO
DO CÁVADO E DO AVE

Escola superior de Tecnologia

Processamento de linguagens

Autores:
Diogo Bernardo ***21144***
João Ribeiro ***23795***
José Fangueiro ***23515***

Docentes:
Prof. Óscar Ribeiro

Submissão do projeto para
o curso de Engenharia de Sistemas Informáticos

ESI

April 14, 2023

IPCA

Resumo

Engenharia de Sistemas Informáticos

Processamento de linguagens

por Diogo Bernardo **21144**
João Ribeiro **23795**
José Fangueiro **23515**

Trabalho em cargo da disciplina de Processamento de linguagens.

Contents

Chapter 1

Exercício 2:

1.1 Descrição do problema

O problema em questão envolve a implementação de um analisador léxico para a simulação da interação de um utilizador com uma máquina de venda automática com comandos em linguagem natural. A máquina (algoritmo) é capaz de gerir a venda automática com base nos comandos disponíveis e na sintaxe definida.

1.2 Descrição da solução implementada

Como sugerido foram implementadas várias formas de interação com a máquina de venda, bem como um lógica de funcionamento favorável á gestão da mesma.

- Venda automática do produto, com base nas regras definidas (ex: possibilidade de troco..)
- É efetuada a análise léxica aos comandos principais, alguns identificam o ínicio de um estado. (INSERT, PRODUCT, REFILL, RESET).., é descrita uma lógica de funcionamento com base nos tokens e estados.
- Análise léxica á inserção de moedas ('e1', 'c20'...), a mesma é feita também para o ato de troco.
- É possível efetuar reposição de stock com um comando em linguagem natural "ex:REFILL=Twix 8".
- São guardadas e otidas informações das moedas produtos e vendas da máquina em ficheiros json.
- Com base nas mooedas existentes é verificado a possibilidade de efeturar troco, foi utilizado um algoritmo para devolver as moedas maiores sem que possível.
- É possível ler uma lista de comandos a efetuar através de um ficheiro devidamente encodificado.
- É possivel retirar as moedas, verificar o valor do stock e o valor em moedas presente na máquina.

1.3 Descrição dos Tokens e estados

1.3.1 Tokens

INSERT Token para entrar no estado de inserção.

CANCEL Token para cancelar uma ação ou estado e voltar ao estado inicial.

PRODUCT Token para entrar no estado de compra.

REFILL Token para entrar no estado de reposição de stock.

ITEM Token para identificar um item.

COIN Token para indentificar uma moeda.

RESET Token para concluir um comando '.'.

OTHER Token para identificar entradas não reconhecidas.

1.3.2 Estados

ALL identifica todos os estados.

INITIAL estado normal do programa, pronto a encontrar identificadores...

inserting estado de inserção.

sell estado de venda.

refill estado de reposição de stock.

1.4 Código implementado

```
""" Ex2 vending maching using python ply.lex."""

import re
import json
from typing import Dict
import ply.lex as plex
import atexit
import math
import datetime
import os


class VendingMachine:
   """A simple vending machine class."""

   products: Dict[str, Dict[str, float]] = {}
   coins: Dict[str, int] = {}
   sales: Dict[str, int] = {}

   states = (
      ("inserting", "exclusive"),
      ("sell", "exclusive"),
      ("refill", "exclusive"),
   )
```

```
tokens = (
    "INSERT",
    "CANCEL",
    "PRODUCT",
    "REFILL",
    "ITEM",
    "COIN",
    "OTHER",
    "RESET",
)

@staticmethod
@plex.TOKEN(r"INSERT")
def t_INSERT(t):
    """
    Transition to the 'inserting' state when in the INITIAL state.

    Args:
        t (Token): The token object.
    """

    t.lexer.begin("inserting")

@staticmethod
@plex.TOKEN(r"PRODUCT")
def t_PRODUCT(t):
    """
    Transition to the 'sell' state when in the INITIAL state.

    Args:
        t (Token): The token object.
    """

    t.lexer.begin("sell")

@staticmethod
@plex.TOKEN(r"REFILL")
def t_REFILL(t):
    """
    Transition to the 'refill' state when in the INITIAL state.

    Args:
        t (Token): The token object.
    """

    t.lexer.begin("refill")

@staticmethod
@plex.TOKEN(r"\.")
def t_ANY_RESET(t):
    """
    Transition to the 'INITIAL' state and print a new line when in ANY state.

    Args:
        t (Token): The token object.
    """
```

```
        print()
        t.lexer.begin("INITIAL")

    @plex.TOKEN(r"[ce][0-9]+")
    def t_inserting_COIN(self, t):
        """
        Process the coin input and update the client balance when in the inserting state.

        Args:
            t (Token): The token object with a value containing the coin
            identifier.
        """

        if t.value[0:] not in self.coins:
            print("Coin not accepted by the vending machine!")
            return

        multiplier = 1
        self.coins[t.value] += 1

        if t.value[0] == "e":
            multiplier = 100

        value = int(t.value[1:]) * multiplier * 0.01
        self.client_balance += value
        print(f"valor inserido: {value:.2f} (saldo: {self.client_balance:.2f})")

    @plex.TOKEN(r"=\w+")
    def t_sell_ITEM(self, t):
        """
        Process the product input for selling when in the sell state.

        Args:
            t (Token): The token object with a value containing the product
            identifier.
        """

        t.value = t.value[1:].rstrip(".")

        product = t.value[0:]
        if product not in self.products:
            print("Invalid product item!")
            return

        self.sell_product(product)
        t.lexer.begin("INITIAL")

    @plex.TOKEN(r"=(\w+)\s(\d+)")
    def t_refill_ITEM(self, t):
        """
        Process the product input for refilling when in the refill state.

        Args:
            t (Token): The token object with a value containing a product name
            and quantity.
        """
```

```
        pattern = re.compile(r"=(\w+)( +)(\d+)")
        match = pattern.match(t.value)

        if match:
            product_name = match.group(1)
            quantity = int(match.group(3))

            self.refill_product(product_name, quantity)

    @plex.TOKEN(r"CANCEL")
    def t_ANY_CANCEL(self, t):
        """
        Process the cancellation and return the money to the client when
        in ANY state

        Args:
            t (Token): The token object.
        """

        print(f"valor devolvido: {self.client_balance:.2f}")

        change = self.calculate_change(self.client_balance)
        self.update_coins(change)
        self.client_balance = 0

        t.lexer.begin("INITIAL")

    @staticmethod
    @plex.TOKEN(r"\n+")
    def t_ANY_newline(t):
        """
        Update the lexer line number.

        Args:
            t (Token): The token object containing newline characters when in ANY state.
        """

        t.lexer.lineno += len(t.value)

    @staticmethod
    @plex.TOKEN(r".")
    def t_ANY_OTHER(t):
        """
        Ignore any other character in the input when in ANY state.

        Args:
            t (Token): The token object containing unrecognized characters.
        """

        t.value = None

    @staticmethod
    def t_ANY_error(t):
        """
        Handle errors for illegal characters in the input when in ANY state.
```

```
        Args:
            t (Token): The token object containing illegal characters.
        """

        print(f"Illegal character '{t.value[0]}'")
        t.lexer.skip(1)

    def __init__(self, **kwargs):
        """
        Initialize the vending machine.

        Attributes:
            lexer: An instance of the lexer created using the plex library.
            client_balance (float): The current balance of the client.
            products (Dict[str, ...]): A dictionary containing product information.
            coins (Dict[str, int]): A dictionary containing coin information.
        """

        self.lexer = plex.lex(module=self, **kwargs)
        self.client_balance = 0

        with open("products.json", "r") as file:
            self.products = json.load(file)

        with open("coins.json", "r") as file:
            self.coins = json.load(file)

        for product in self.products.keys():
            self.sales[product] = 0

        atexit.register(self._save_state)

    def _save_state(self):
        """
        Save the current state of the vending machine to JSON files.
        This inclues products info, coins and the sales reports.
        """

        with open("products.json", "w") as file:
            json.dump(self.products, file, indent=4)

        with open("coins.json", "w") as file:
            json.dump(self.coins, file, indent=4)

        if not os.path.exists("reports"):
            os.makedirs("reports")

        current_time = datetime.datetime.utcnow()
        filename = f"{current_time.strftime('%Y%m%d_%H%M%S')}.json"
        file_path = os.path.join("reports", filename)
        with open(file_path, "w") as file:
            json.dump(self.sales, file, indent=4)

    def reset_coins(self):
        """
        Reset the available coins to their initial quantity
        simulated (20 for each coin).
```

```
        """

        for coin in self.coins:
            self.coins[coin] = 20

    def sell_product(self, product):
        """
        Attempt to sell a product if the balance is sufficient.

        Args:
            product (str): The name of the product to sell.
        """

        price = self.products[product]["price"]
        if not price or price <= 0:
            print(f"Invalid product, aborting item purchase! ({product})")
            return

        stock = self.products[product]["stock"]
        if not stock or stock <= 0:
            print(f"There are no products to sell, aborting! ({product})")
            return

        if self.client_balance >= price:
            change_needed = self.client_balance - price
            change = self.calculate_change(change_needed)

            if change is not None or math.isclose(self.client_balance, price):
                print(
                    f"Purchased: '{product}', {price:.2f} "
                    f"(change = {(self.client_balance - price):.2f})"
                )
                self.client_balance = 0.0
                if change_needed > 0.0095:
                    self.update_coins(change)
                self.products[product]["stock"] -= 1
                self.sales[product] += 1
            else:
                print(
                    "Cannot provide change for the purchase "
                    "(Aborting the item purchase!)"
                )
        else:
            print(
                f"Price: {price:.2f} (insufficient funds) "
                f"(client balance: {self.client_balance:.2f})"
            )

    def calculate_change(self, change_needed):
        """
        Calculate the change to be given using the available coins.

        Args:
            change_needed (float): The amount of change needed.

        Returns:
            Optional[List[str]]: A list of coin names representing the change,
```

```
            or None if exact change cannot be given.
        """

        coin_values = {
            "c5": 0.05,
            "c10": 0.10,
            "c20": 0.20,
            "c50": 0.50,
            "e1": 1.00,
            "e2": 2.00,
        }
        coins_copy = self.coins.copy()
        change = []

        for coin_name, coin_value in sorted(
            coin_values.items(), key=lambda x: x[1], reverse=True
        ):
            while coins_copy[coin_name] > 0 and change_needed >= coin_value:
                change_needed -= coin_value
                change_needed = round(change_needed, 2)
                change.append(coin_name)
                coins_copy[coin_name] -= 1

        if change_needed > 0:
            return None

        print(f"devolvido {change}")
        return change

    def update_coins(self, change):
        """
        Update the available coins based on the given change.

        Args:
            change (List[str]): A list of coin names representing the change.
        """

        for coin in change:
            self.coins[coin] -= 1

    def coins_value(self):
        """
        Calculate the total value of all coins in the vending machine.

        Returns:
            float: The total value of all coins in the vending machine.
        """

        total = 0.0
        for coin, qty in self.coins.items():
            multiplier = 100 if coin[0] == "e" else 1
            coin_value = int(coin[1:]) * multiplier * 0.01
            total += coin_value * qty

        return total

    def stock_value(self):
```

```
        """
        Calculate the total value of all products in the vending machine.

        Returns:
            float: The total value of all products in the vending machine.
        """

        total = 0.0
        for product_info in self.products.values():
            total += product_info["stock"] * product_info["price"]

        return total

    def refill_product(self, product_name, quantity):
        """
        Update the stock of a product in the products dictionary by adding
        the specified quantity.

        Args:
            product_name (str): The name of the product to be updated.
            quantity (int): The quantity to be added to the stock.
        """

        if product_name in self.products:
            self.products[product_name]["stock"] += quantity
            print(
                f"Refilled {product_name} with {quantity} units. New stock: "
                f"{self.products[product_name]['stock']}"
            )
        else:
            print(f"Product to refill not found: {product_name}")

    def _print_sale(self):
        """Print the sales report for the vending machine."""

        print("_________________________________________________")
        print("Sales report:")
        for product, count in self.sales.items():
            if count > 0:
                print(f"\t{product}: {count}")
        print()

    def process(self, data):
        """
        Process the input data to simulate the vending machine.

        Args:
            data (str): The input data containing commands for the vending
            machine.
        """

        if self.lexer is None:
            return

        self.lexer.input(data)

        for _ in self.lexer:
```

```
            continue

        self._print_sale()


def main():
    with open("input.dat", "r") as file:
        data = file.read()

    vending_machine = VendingMachine()
    vending_machine.process(data)

    print(
        f"The monetary value in the vending machine {vending_machine.coins_value():.2f}"
    )
    print(f"The vending machine stock value {vending_machine.stock_value():.2f}")


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

1.4.1 produtos.json

Neste ficheiro é guardada a informação dos produtos com a respetiva quantidade e preço por unidade. É automaticamente gerado e atualizado pelo programa.
Exemplo:

```
{
    "KitKat": {
        "stock": 10,
        "price": 0.5
    },
    "KinderBueno": {
        "stock": 7,
        "price": 1.3
    },
    "Cola": {
        "stock": 19,
        "price": 1.3
    },
    "RedBull": {
        "stock": 14,
        "price": 1.9
    },
    "Monster": {
        "stock": 10,
        "price": 1.9
    },
    "SuperBock": {
        "stock": 4,
        "price": 1.2
    },
    "Agua": {
        "stock": 15,
        "price": 0.4
    },
    "Bolachas": {
```

```
      "stock": 8,
      "price": 0.9
   },
   "BarraCereais": {
      "stock": 5,
      "price": 0.5
   },
   "KinderBuenoWhite": {
      "stock": 12,
      "price": 1.3
   },
   "Twix": {
      "stock": 44,
      "price": 2.3
   },
   "Mars": {
      "stock": 8,
      "price": 0.7
   },
   "Snickers": {
      "stock": 9,
      "price": 0.9
   }
}
```

1.4.2 coins.json

Neste ficheiro é guardada a informação das moedas com a respetiva quantidade. É automaticamente gerado e atualizado pelo programa.
Exemplo:

```
{
   "c5": 429,
   "c10": 494,
   "c20": 159,
   "c50": 571,
   "e1": 47,
   "e2": 52
}
```

1.4.3 Exemplo do ficheiro input.dat

É neste ficheiro onde são definidos todos os comandos a ser interpretados pelo analizador léxico. É automaticamente lido pelo programa, a escrita não é permitida!

```
INSERT c10, e1, c50, c50.
PRODUCT=Twix.
INSERT c20, c70.
PRODUCT=Twix.
INSERT c20, c10, c5, c50, c10, c5.
CANCEL.
REFILL=Twix 1.
```

1.4.4 report/executionTime.json

Neste ficheiro é guardado o histórico de vendas durante a execução do programa. É automaticamente gerado e atualizado pelo programa.
Exemplo: reports/20230414_171358.json


```
{
   "KitKat": 0,
   "KinderBueno": 0,
   "Cola": 0,
   "RedBull": 0,
   "Monster": 0,
   "SuperBock": 0,
   "Agua": 0,
   "Bolachas": 0,
   "BarraCereais": 0,
   "KinderBuenoWhite": 0,
   "Twix": 1,
   "Mars": 0,
   "Snickers": 0
}
```


1.5 Teste do código

Com a instalação de todas as dependências é possível executar o algoritmo através de um interpretador de python3. Este foi o ouput gerado pelos comandos definidos em input.dat:

```
inserted value: 0.10€ (balance: 0.10€)
inserted value: 1.00€ (balance: 1.10€)
inserted value: 0.50€ (balance: 1.60€)
inserted value: 0.50€ (balance: 2.10€)

Price: 2.30€ (insufficient funds) (client balance: 2.10€)

inserted value: 0.20€ (balance: 2.30€)
Coin not accepted by the vending machine!

Purchased: 'Twix', 2.30€ (change = 0.00€)

inserted value: 0.20€ (balance: 0.20€)
inserted value: 0.10€ (balance: 0.30€)
inserted value: 0.05€ (balance: 0.35€)
inserted value: 0.50€ (balance: 0.85€)
inserted value: 0.10€ (balance: 0.95€)
inserted value: 0.05€ (balance: 1.00€)

returned value: 1.00€
devolvido ['e1']

Refilled Twix with 1 units. New stock: 45
Refilled KitKat with 3 units. New stock: 19
```

```
_________________________________________________
Sales report:
        Twix: 1

The monetary value in the vending machine 587.45€
The vending machine stock value 242.20€
```