# MoneyLang

**Uma Linguagem de Domínio Específico para Operações Bancárias**

---

## Índice



---

## Motivação

**Por que MoneyLang?**

O setor bancário e financeiro lida diariamente com operações complexas que envolvem transferências, cálculos de juros, validações de saldo e gestão de múltiplas contas. Desenvolver sistemas para essas operações em linguagens de propósito geral frequentemente resulta em código verboso e propenso a erros.

**MoneyLang foi criada para:**

1. **Simplificar operações financeiras**: Abstrair a complexidade de operações bancárias em comandos intuitivos
2. **Reduzir erros**: Sintaxe específica do domínio que torna o código mais legível e menos propenso a bugs
3. **Acelerar desenvolvimento**: Prototipagem rápida de simulações bancárias e sistemas financeiros
4. **Ensinar conceitos financeiros**: Sintaxe em português facilita o aprendizado de programação e conceitos bancários
5. **Testar cenários**: Simular rapidamente diferentes cenários de operações bancárias

**Casos de Uso**

- Simulações de investimentos e rendimentos
- Prototipagem de sistemas bancários
- Ensino de programação e educação financeira
- Testes de algoritmos financeiros
- Jogos educativos sobre finanças

---

## Características

### 1. Sintaxe em Português

MoneyLang utiliza palavras-chave em português, tornando-a acessível para falantes nativos e estudantes:

```
conta poupanca = 1000
depositar(poupanca, 500)
mostrar("Saldo:", poupanca)
```

### 2. Operações Bancárias Nativas

#### Declaração de Contas

```
conta corrente = 5000
conta poupanca = 10000
```

#### Depósitos e Saques

```
depositar(corrente, 1000)
sacar(poupanca, 500)
```

#### Transferências

```
transferir(origem, destino, 2000)
```

#### Aplicação de Juros

```
aplicar_juros(investimento, 0.08)  # 8% de rendimento
```

### 3. Estruturas de Controle

#### Condicionais

```
se (saldo >= 1000)
    mostrar("Cliente VIP")
senão
    mostrar("Cliente Regular")
```

#### Laços

```
enquanto (parcelas > 0)
    sacar(conta, valor_parcela)
    parcelas = parcelas - 1
```

### 4. Operadores Aritméticos Completos

```
total = (base + bonus) * taxa / 100
resto = valor % 30
negativo = -(saldo - divida)
```

Suporta: `+`, `-`, `*`, `/`, `%`, e operador unário `-`

**5. Operadores de Comparação**

```
se (saldo == 0)        # Igual
se (saldo != 0)        # Diferente
se (saldo < 1000)      # Menor
se (saldo > 5000)      # Maior
se (saldo <= 1000)     # Menor ou igual
se (saldo >= 5000)     # Maior ou igual
```

**6. Sensores do Sistema**

MoneyLang possui "sensores" que fornecem informações do ambiente:

```
# Sensor de tempo (segundos desde o início)
tempo_decorrido = tempo

# Sensor de taxa de juros do sistema
taxa_atual = juros
aplicar_juros(conta, juros)
```

**7. Sistema de Tipos Simples**

- **Números**: Ponto flutuante de 64 bits
- **Contas**: Variáveis especiais para armazenar valores monetários
- **Strings**: Apenas para impressão

**8. Entrada/Saída Intuitiva**

```
mostrar("Saldo:", saldo)
mostrar("Cliente:", nome, "Saldo:", valor)
```

---

## Curiosidades

**1. Arquitetura de Compilador Clássica**

MoneyLang utiliza **Flex** (análise léxica) e **Bison** (análise sintática), as mesmas ferramentas usadas em compiladores profissionais como GCC.

**2. Máquina Virtual Própria**

A **BankVM** é uma máquina virtual baseada em pilha, similar à JVM, mas otimizada para operações bancárias. Ela possui instruções específicas como `DEPOSIT`, `WITHDRAW`, `TRANSFER` e `APPLY_INTEREST`.

### 3. Geração de Assembly

O compilador não executa diretamente - ele **transpila** MoneyLang para assembly BankVM, permitindo: - Inspeção do código intermediário - Otimizações no nível de assembly - Portabilidade para diferentes backends

### 4. Indentação Significativa (Estilo Python)

MoneyLang usa indentação para delimitar blocos:

```
se (x > 0)
    # bloco indentado
    mostrar("positivo")
```

### 5. Sensores Temporais

O sensor `tempo` permite criar simulações que dependem do tempo real de execução - útil para benchmarks e demonstrações.

### 6. Precisão Financeira

Internamente, a BankVM usa ponto flutuante de 64 bits (IEEE 754), oferecendo precisão de ~15-17 dígitos decimais.

### 7. Zero Dependências Externas

O interpretador é um único arquivo Python sem dependências externas - pode ser executado em qualquer ambiente com Python 3.6+.

### 8. Modo Debug

A VM possui um modo debug que mostra a execução passo a passo:

```
python3 vm/bankvm.py programa.asm --debug
```

---

## Exemplos

### Exemplo 1: Operações Básicas

```
conta poupanca = 1000

depositar(poupanca, 500)
mostrar("Após depósito:", poupanca)

sacar(poupanca, 200)
mostrar("Após saque:", poupanca)
```

**Saída:**

```
Após depósito: 1500
Após saque: 1300
```

---

**Exemplo 2: Transferências Entre Contas**

```
conta origem = 5000
conta destino = 1000

mostrar("Antes - Origem:", origem, "Destino:", destino)

transferir(origem, destino, 1500)

mostrar("Depois - Origem:", origem, "Destino:", destino)
```

**Saída:**

```
Antes - Origem: 5000 Destino: 1000
Depois - Origem: 3500 Destino: 2500
```

---

**Exemplo 3: Simulação de Investimento**

```
conta investimento = 10000

mostrar("Capital inicial:", investimento)

# Aplicar 5% de juros por 3 meses
aplicar_juros(investimento, 0.05)
mostrar("Mês 1:", investimento)

aplicar_juros(investimento, 0.05)
mostrar("Mês 2:", investimento)

aplicar_juros(investimento, 0.05)
mostrar("Mês 3:", investimento)

mostrar("Rendimento:", investimento - 10000)
```

**Saída:**

```
Capital inicial: 10000
Mês 1: 10500.00
Mês 2: 11025.00
Mês 3: 11576.25
Rendimento: 1576.25
```

---

**Exemplo 4: Controle de Fluxo**

```
conta saldo = 750

se (saldo >= 1000)
    mostrar("Cliente VIP")
    aplicar_juros(saldo, 0.10)
senão
    mostrar("Cliente Regular")
    aplicar_juros(saldo, 0.05)

mostrar("Saldo final:", saldo)
```

**Saída:**

```
Cliente Regular
Saldo final: 787.50
```

---

**Exemplo 5: Pagamento em Parcelas**

```
conta saldo = 1000
conta parcelas = 5

mostrar("Saldo inicial:", saldo)
mostrar("Parcelas:", parcelas)

enquanto (parcelas > 0)
    sacar(saldo, 150)
    parcelas = parcelas - 1
    mostrar("Pago! Restam", parcelas, "parcelas | Saldo:", saldo)

mostrar("Saldo final:", saldo)
```

**Saída:**

```
Saldo inicial: 1000
Parcelas: 5
Pago! Restam 4 parcelas | Saldo: 850
Pago! Restam 3 parcelas | Saldo: 700
Pago! Restam 2 parcelas | Saldo: 550
Pago! Restam 1 parcelas | Saldo: 400
Pago! Restam 0 parcelas | Saldo: 250
Saldo final: 250
```

---

**Exemplo 6: Usando Sensores**

```
conta poupanca = 5000

mostrar("Taxa de juros do sistema:", juros)

# Aplicar juros baseado no sensor
aplicar_juros(poupanca, juros)

mostrar("Saldo atualizado:", poupanca)
mostrar("Tempo de execução:", tempo, "segundos")
```

**Saída:**

```
Taxa de juros do sistema: 0.05
Saldo atualizado: 5250.00
Tempo de execução: 0.001234 segundos
```

---

**Exemplo 7: Simulação Bancária Completa**

```
conta corrente = 2000
conta poupanca = 5000
conta investimento = 10000

mostrar("=== BANCO MONEYLANG ===")
mostrar("Corrente:", corrente)
mostrar("Poupança:", poupanca)
mostrar("Investimento:", investimento)

# Cliente faz saque
sacar(corrente, 300)
mostrar("\nApós saque de 300:")
mostrar("Corrente:", corrente)

# Transfere para investimento
transferir(corrente, investimento, 500)
mostrar("\nApós transferir 500:")
mostrar("Corrente:", corrente)
mostrar("Investimento:", investimento)

# Rendimentos mensais
aplicar_juros(poupanca, 0.03)
aplicar_juros(investimento, 0.08)

mostrar("\n=== APÓS RENDIMENTOS ===")
mostrar("Poupança (3%):", poupanca)
```

```
mostrar("Investimento (8%):", investimento)

# Verificar tarifas
se (corrente < 1000)
    mostrar("\nCobrar tarifa de manutenção")
    sacar(corrente, 15)

# Calcular patrimônio total
conta total = corrente + poupanca + investimento
mostrar("\nPATRIMÔNIO TOTAL:", total)
```

---

## Como Usar

### Pré-requisitos

- **GCC/Clang** com suporte a C11
- **Flex** (analisador léxico)
- **Bison** (gerador de parsers)
- **Python 3.6+** (para executar a VM)

### Instalação

```
# Clonar o repositório
git clone https://github.com/seu-usuario/moneylang.git
cd moneylang

# Compilar o compilador MoneyLang
make

# Verificar instalação
./bin/moneyc --help
```

### Compilando e Executando

```
# 1. Escrever um programa MoneyLang (exemplo.money)
# 2. Compilar para assembly BankVM
./bin/moneyc exemplo.money -o exemplo.asm

# 3. Executar na máquina virtual
python3 vm/bankvm.py exemplo.asm
```

### Modo Debug

```
# Ver execução passo a passo
python3 vm/bankvm.py exemplo.asm --debug
```

### Testando Exemplos

```
# Executar todos os exemplos de teste
./test_exemplos.sh
```

---

### Componentes

1. **Frontend (Compilador)**
   - `src/lexer.l`: Analisador léxico (Flex)
   - `src/parser.y`: Analisador sintático (Bison)
   - `src/ast.c`: Árvore Sintática Abstrata
   - `src/codegen.c`: Gerador de código assembly

2. **Backend (Máquina Virtual)**
   - `vm/bankvm.py`: Interpretador da BankVM
     - Pilha de execução
     - Armazenamento de contas
     - Sistema de labels e jumps
     - Primitivos bancários
     - Sistema de sensores

3. **Especificações**
   - `docs/VM_SPEC.md`: Especificação completa do assembly BankVM
   - `README.md`: Gramática EBNF da linguagem
   - `AGENTS.md`: Guia de contribuição

---

## Recursos Adicionais

### Estrutura de Diretórios

```
moneylang/
  src/                 # Código-fonte do compilador
     lexer.l          # Análise léxica
     parser.y         # Análise sintática
     ast.c            # AST
     codegen.c        # Geração de código
  include/             # Headers
  vm/                  # Máquina virtual
     bankvm.py       # Interpretador
  exemplos/            # Exemplos de código
     01_operacoes_basicas.money
     02_transferencias.money
```

```
   ...
docs/                # Documentação
   VM_SPEC.md      # Especificação da VM
bin/                 # Binários compilados
build/               # Artefatos de build
```

**Comandos Disponíveis**

| Comando | Descrição |
|---|---|
| `conta <nome> = <valor>` | Declara uma conta |
| `depositar(<conta>, <valor>)` | Adiciona valor à conta |
| `sacar(<conta>, <valor>)` | Remove valor da conta |
| `transferir(<origem>, <destino>, <valor>)` | Transfere entre contas |
| `aplicar_juros(<conta>, <taxa>)` | Aplica juros à conta |
| `mostrar(...)` | Imprime valores |
| `se (...) ... senão ...` | Estrutura condicional |
| `enquanto (...) ...` | Laço de repetição |

**Operadores**

| Categoria | Operadores |
|---|---|
| Aritméticos | `+`, `-`, `*`, `/`, `%` |
| Comparação | `==`, `!=`, `<`, `>`, `<=`, `>=` |
| Unários | `-` (negação) |

**Sensores**

| Sensor | Retorna |
|---|---|
| `tempo` | Segundos desde o início do programa |
| `juros` | Taxa de juros base do sistema (0.05) |

---

## Autor

Yuri Tabacof