

# C-ash

## 1. Apresentação:

A linguagem de programação *C-ash* nasceu da necessidade de unir simplicidade, segurança e clareza no desenvolvimento de sistemas bancários digitais. Com a crescente digitalização dos serviços financeiros, o setor exige soluções que reduzam riscos de falhas e aumentem a confiabilidade do software. Embora linguagens de uso geral, como Java, ainda sejam amplamente utilizadas, elas não foram projetadas especificamente para lidar com as particularidades do domínio financeiro, o que abre espaço para erros de implementação que podem resultar em prejuízos graves.

Diferente das linguagens tradicionais, *C-ash* é uma linguagem de domínio específico (DSL) criada para programar maquininhas de pagamento e caixas eletrônicos virtuais. Sua sintaxe foi desenhada para ser facilmente comprehensível e auditável, permitindo que até desenvolvedores juniores consigam implementar regras financeiras sem ambiguidades. O compilador exige a declaração explícita da moeda de cada banco virtual, obriga a presença de ao menos uma política antifraude e garante que toda operação de transferência esteja protegida dentro de blocos transacionais com rollback automático.

Entre seus diferenciais estão o tipo nativo `money` (que elimina os problemas de precisão numérica encontrados em linguagens comuns), os bancos virtuais como classes com moeda fixa e políticas próprias, o suporte a eventos reais de ambiente bancário (como `on card\_insert` ou `on net\_fail`) e a possibilidade de o próprio programador definir regras de antifraude sob medida. Com isso, *C-ash* entrega alta confiabilidade, auditabilidade e segurança por design, consolidando-se como uma base sólida para aplicações críticas no universo financeiro digital.

### 1.1 Filosofia de design

C-ash adota o princípio "*tudo é transação*", assegurando consistência e reversibilidade em qualquer operação que altere o estado.

Nenhum efeito colateral é permitido fora de blocos transacionais.

Além disso, o compilador prioriza a legibilidade e a auditabilidade.

Cada instrução de alto nível tem correspondência direta e explícita em PocketASM — eliminando a ambiguidade entre o código-fonte e a execução real.

O foco é segurança, previsibilidade e simplicidade: se um programa compila, ele é financeiramente seguro por construção.

---

## 2. Arquitetura

### 2.1 Componentes principais

Componente	Descrição
<b>Parser</b>	Analisa o código C-ash e gera a Árvore Sintática Abstrata (AST).
<b>Codegen</b>	Converte a AST em instruções PocketASM.
<b>PocketVM</b>	Máquina virtual que executa as instruções de PocketASM.

### 2.2 Fluxo de compilação

C-ash (.cash)

↓ Parser

↓ AST

↓ Codegen

PocketASM (.asm)

↓ PocketVM

Execução

---

## 3. Estrutura de um programa C-ash

Um programa C-ash é composto por **declarações, funções, transações, políticas bancárias e chamadas a built-ins**.

### 3.1 Exemplo básico

```
fn greet(name) {  
    display("Hello, ");  
    display(name);  
}  
  
reserve userName: string = "Ash";  
greet(userName);
```

---

## 4. Tipos de dados

Tipo	Descrição	Exemplo
int	Números inteiros	42
bool	Valores lógicos	true, false
string	Cadeias de texto	"C-ash"
money	Quantia monetária com moeda	100\$BRL
account	Referência a uma conta bancária	myBank:acct

---

## 5. Declarações e variáveis

### 5.1 Declaração simples

```
reserve saldo: money = 1000$BRL;
```

## 5.2 Inferência de tipo

```
reserve nome = "Ash";
```

## 5.3 Atribuição

```
saldo = saldo - 50$BRL;
```

---

# 6. Expressões e operadores

Categoria	Operadores	Exemplo
Aritméticos	+, -, *, /, %	a + b
Comparação	<, >, <=, >=, ==, !=	x == y
Lógicos	&&	
Unários	-, !	-x, !cond

---

# 7. Controle de fluxo

## 7.1 Estrutura condicional

```
if saldo > 0 {  
    display("Saldo positivo");  
} else {  
    display("Conta zerada");  
}
```

## 7.2 Loop while

```
while tentativas < 3 {  
    tentativas = tentativas + 1;  
}
```

## 7.3 Loop for

```
for i = 0; i < 10; i = i + 1 {  
    display(i);  
}
```

---

# 8. Funções

## 8.1 Declaração

```
fn soma(a, b) {  
    return a + b;  
}
```

## 8.2 Chamada

```
reserve x = soma(10, 5);
```

## 8.3 Retorno opcional

Funções sem `return` retornam `0` (ou tipo neutro equivalente).

---

## 9. Transações (transaction / onfail)

### 9.1 Estrutura

```
transaction {  
    xfer(from, to, 100$BRL);  
} onfail {  
    display("Falha na transação.");  
}
```

### 9.2 Regras

- Cada transação cria um **snapshot** do ambiente.
- Se um erro ou violação ocorrer, a VM executa **TX\_ROLLBACK**.
- O código dentro de **onfail** é executado automaticamente após um rollback.

### 9.3. Exemplo de rollback automático

```
bank PocketBR { currency=BRL; policy limit all 2000$BRL; }
```

```
open PocketBR("Ash") as br;
```

```
reserve valor: money = 3000$BRL;
```

```
transaction {  
    display("Iniciando transferência...");  
    enforce(br, valor);  
    xfer(br, br, valor); // Falha: excede limite  
    display("Essa linha não será exibida");  
} onfail {  
    display("Transação revertida com sucesso.");  
}
```

## Saída esperada:

[VM] TX\_BEGIN

Iniciando transferência...

[VM] LIMITE EXCEDIDO: 3000\$BRL > 2000\$BRL

[VM] TX\_ROLLBACK

Transação revertida com sucesso.

---

## 10. Funções built-in

Função	Descrição	Exemplo
display(x)	Exibe um valor na saída padrão.	display("ok")
print(x)	Igual a <code>display</code> .	
beep()	Emite um alerta.	
xfer(from, to, amount)	Transfere valores entre contas.	
enforce(cond, msg)	Garante uma condição antifraude.	
approve()	Aprova uma operação.	
deny()	Nega uma operação.	
now()	Retorna o timestamp atual.	
region()	Retorna a região da execução.	
fx_rate(a,b)	Retorna a taxa de câmbio entre duas moedas.	

---

## 11. Políticas e bancos

## 11.1 Declaração de banco

```
bank MyBank currency=BRL {  
    policy limit all 1000$BRL;  
}
```

## 11.2 Conexões

```
connect OtherBank using fx {  
    rule convert { ... }  
}
```

## 11.3 Abertura de conta

```
open MyBank as contaCliente;
```

---

## 12. Comentários

### Linha única:

; Este é um comentário

- **Multilinha:** não suportado (apenas ; por linha).
- 

## 13. Convenções e boas práticas

- Use **nomes descritivos** para variáveis (saldoCliente, valorTransferido).

- Sempre encapsule `xfer` dentro de `transaction + onfail`.
  - Agrupe lógicas financeiras em funções puras e chamadas de I/O em camadas externas.
  - Prefira funções pequenas e modulares.
- 

## 14. Exemplo completo

```
bank BancoCentral currency=BRL {  
    policy limit all 5000$BRL;  
}  
  
fn pagar(from, to, valor) {  
    transaction {  
        enforce(valor < 5000$BRL, "Valor excede limite");  
        xfer(from, to, valor);  
        display("Transferência concluída");  
    } onfail {  
        display("Erro na operação");  
    }  
}  
  
open BancoCentral as c1;  
open BancoCentral as c2;  
  
pagar(c1, c2, 200$BRL);
```

---

## 15. Licença

O projeto C-ash é distribuído sob a **licença MIT**, permitindo modificação, redistribuição e uso comercial com atribuição adequada.