TRABALHO DE CONCLUSÃO DE CURSO

Micro C

Um Compilador Acadêmico

Autor: Pedro Henrique Ferreira Carvalho
Orientador: Prof. Brivaldo Alves da Silva Junior

Faculdade de Computação (FACOM)- UFMS
Dezembro de 2025

O Problema

- Compiladores são a base da computação, mas ferramentas modernas (GCC, LLVM) são massivas e complexas, contendo milhões de linhas de código.
- Existe uma lacuna clara entre a teoria do *"Livro do Dragão"* e a prática real de construção de um compilador.

Questão de Trabalho

Como construir uma ferramenta que transforme essa complexidade que é um compilador em algo acessível para o aprendizado?

Objetivos do Trabalho

Objetivo Geral

Projetar, implementar e validar um compilador funcional para a linguagem Micro C, gerando código binário executável de forma didática.

Objetivos Específicos

- Definir a linguagem **Micro C** (subconjunto do C).
- Implementar o **Front-End** (Análise Léxica, Sintática e Semântica).
- Implementar o **Back-End** (Geração de IR e Assembly x86-64).
- Validar com algoritmos (Bubble Sort, Fibonacci).

Metodologia: A Pipeline

Arquitetura baseada em fases distintas, desenvolvida em C.

Scanner (Tokens) → **Parser** (AST) → **Semantic** (Tabelas) → **IR Gen** (TAC) → **Assembly** (x86-64)

Front-End (Análise)

Entende o que o código faz e valida regras.

Back-End (Síntese)

Geração de código de máquina para o processador.

Metodologia: O Ciclo de Vida Completo

Comparação entre a implementação interna do Micro C e as ferramentas externas.

COMPILADOR MICRO C

1. Análise Léxica — Micro C
2. Análise Sintática — Micro C
3. Análise Semântica — Micro C
4. Geração de IR — Micro C
5. Geração de Assembly — Micro C

INFRAESTRUTURA & FUTURO

(Otimização de Código) — Futuro

6. Montagem (Assembler) — GCC
7. Linkedição (Linker) — GCC

Estratégia: O projeto concentra esforços nas **5 fases principais** da compilação (Esquerda), traduzindo código - fonte Micro C até Assembly x86-64.
A fase de *Otimização* foi abstraída por fins de simplicidade, e a transformação final em binário é delegada ao **GCC**.

A Linguagem Fonte: Micro C

⚠ C PADRÃO: MANUAL

```
printf( "%d", a);
```

O compilador resolve o formato para você!

MICRO C: AUTOMÁTICO

```
print(a);
```

Um Subconjunto do C

Mantém a sintaxe familiar (`int`, `if`, `for`), mas remove ponteiros e structs complexas para reduzir a curva de aprendizado.

Abstração Didática

Foco na lógica, não na burocracia. Enquanto o C exige gerenciamento manual de formatos (`%d`, `%s`), o Micro C abstrai isso.

COMO FUNCIONA

1. Frontend: Identifica tipo na Tabela de Símbolos.
2. Backend: Gera assembly com "%d" automaticamente.

Exemplo Base: `soma.mcc`

soma.mcc

```
1 int main() {
2 int a;
3 int b;
4 a = 10;
5 b = 20;
6 print(a + b);
7 return 0;
8 }
```

Propósito

Validar o ciclo de compilação completo do *Micro C* na sua forma mais fundamental.

Funcionalidades Testadas:

- Declaração e alocação de variáveis locais.
- Operação aritmética (+) e atribuição.
- Chamada de função externa (`print`) via ABI.

Este código simples será nosso guia. Vamos acompanhar como esses dados (a, b) viajam por todas as 5 fases do compilador.

Fase 1- Análise Léxica

```
total = 10;
```

LEXEMA	"10"
PADRÃO	[0-9]+
TOKEN	INTEGERCONST

FLUXO

```
int a ; //var
```

[INT] [ID] [SEMI]

Espaços
//var

A funcionalidade

Ler o código-fonte caractere por caractere e agrupá-los em unidades de significado.

A Tríade Léxica

Lexema: O texto real (`"10"`).

Padrão: A regra (`[0-9]+`).

Token: A categoria abstrata (`INTEGERCONST`).

Limpeza de Ruído

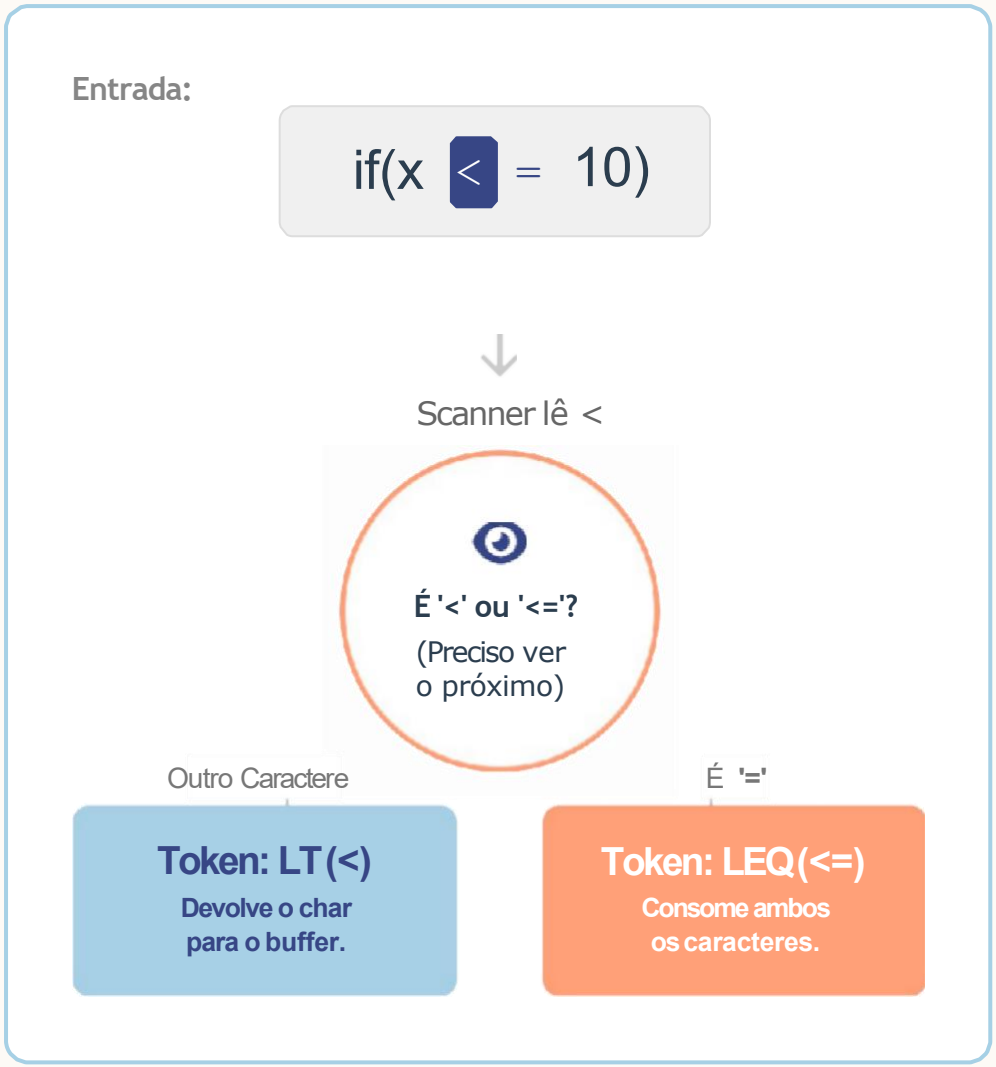
O Scanner descarta tudo que não é essencial para a lógica: espaços em branco, quebras de linhas e comentários (`//`).

Identificação de Erros

Padrão Desconhecido: Caracteres inválidos (ex: `@`) geram token `UNDEF`.

String Aberta: Encontrar `EOF` antes de fechar aspas indica string malformada.

Fase 1- Análise Léxica: Lookahead



⚠ O Desafio

- O caractere < é **ambíguo.**
- Ele pode ser um operador simples (menor que) ou o início de um operador composto (menor ou igual).

A Solução: Lookahead

Implementamos uma função que permite ao scanner espiar o futuro sem se comprometer.

Como Funciona:

1. O scanner consome o <.
2. Ele verifica o próximo caractere.
3. Se for =, consome e cria **LEQ**.
4. Caso contrário, cria **LT** e devolve o próximo.

Fase 1: Produto Final - A Cadeia de Tokens

TIPO	LEXEMA	LINHA
INT	int	1
MAIN	main	1
ID	a	3
SEMICOLON	;	3
[... 20 Tokens Processados ...]		
ID	a	9
PLUS	+	9
ID	b	9
RPAREN	)	9
RETURN	return	11
EOF	EOF	13

Sucesso

O scanner processou o código fonte e gerou com sucesso uma cadeia linear de **31 tokens**.

Separação Lógica

Crucial para as próximas fases:

- Tipo: Para o Parser (ex: PLUS)
- Lexema: Para a Semântica (ex: valor '10' ou nome 'a')

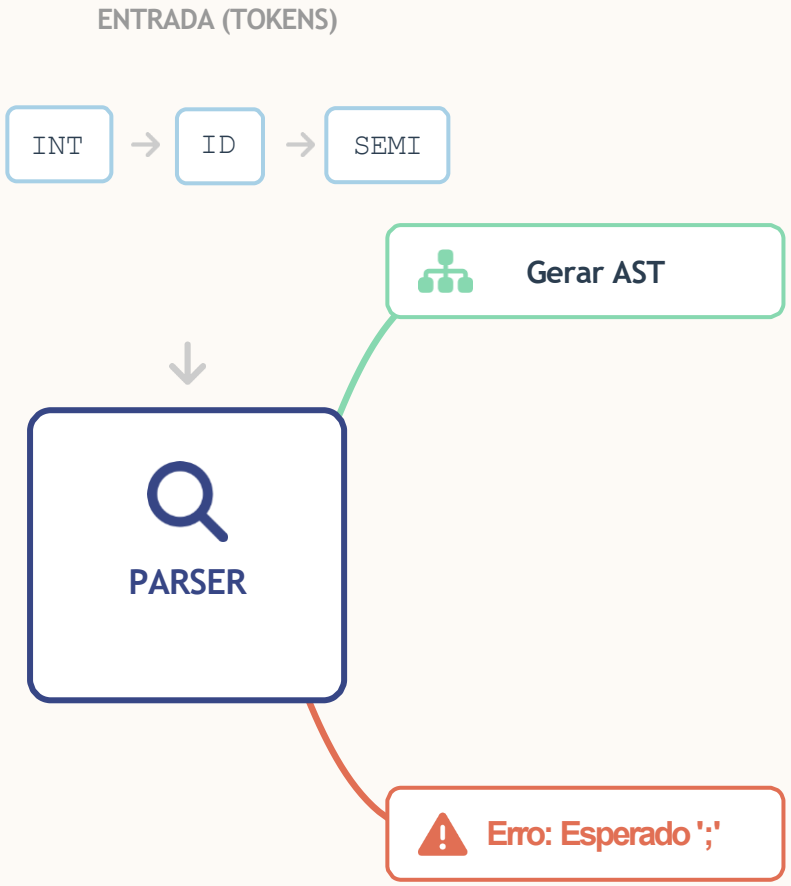
Precisão do DFA

O autômato provou sua capacidade de diferenciar palavras reservadas de identificadores e capturar operadores.

Rastreabilidade

O número da linha é preservado em cada token, garantindo mensagens de erro precisas no futuro.

Fase 2 - Análise Sintática



O Objetivo

Validar a **Gramática**. O parser verifica se o código obedece rigorosamente às regras estruturais da linguagem Micro C.

A Regra

A **ordem** dos tokens importa. O parser verifica se a fila faz sentido.

✓ Válido: `int a;`

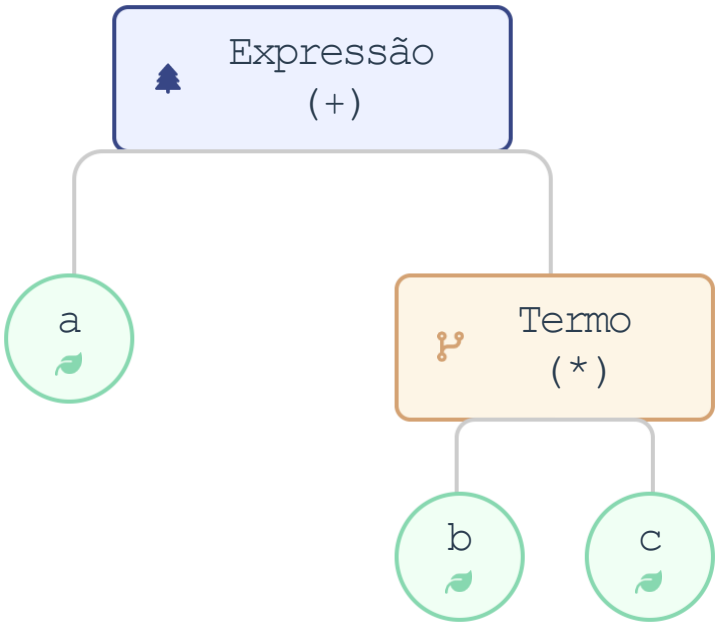
✗ Inválido: ~~`a int;`~~ (Ordem errada)

Resultado

Se a sequência for válida, avançamos para a construção da estrutura (AST). Se não, um erro sintático é reportado imediatamente.

Fase 2 - A Hierarquia de Precedência

Resolvendo `a + b * c` usando a estrutura da gramática.



O Conceito

Para garantir a precedência correta, o Parser organiza o código como uma árvore viva.

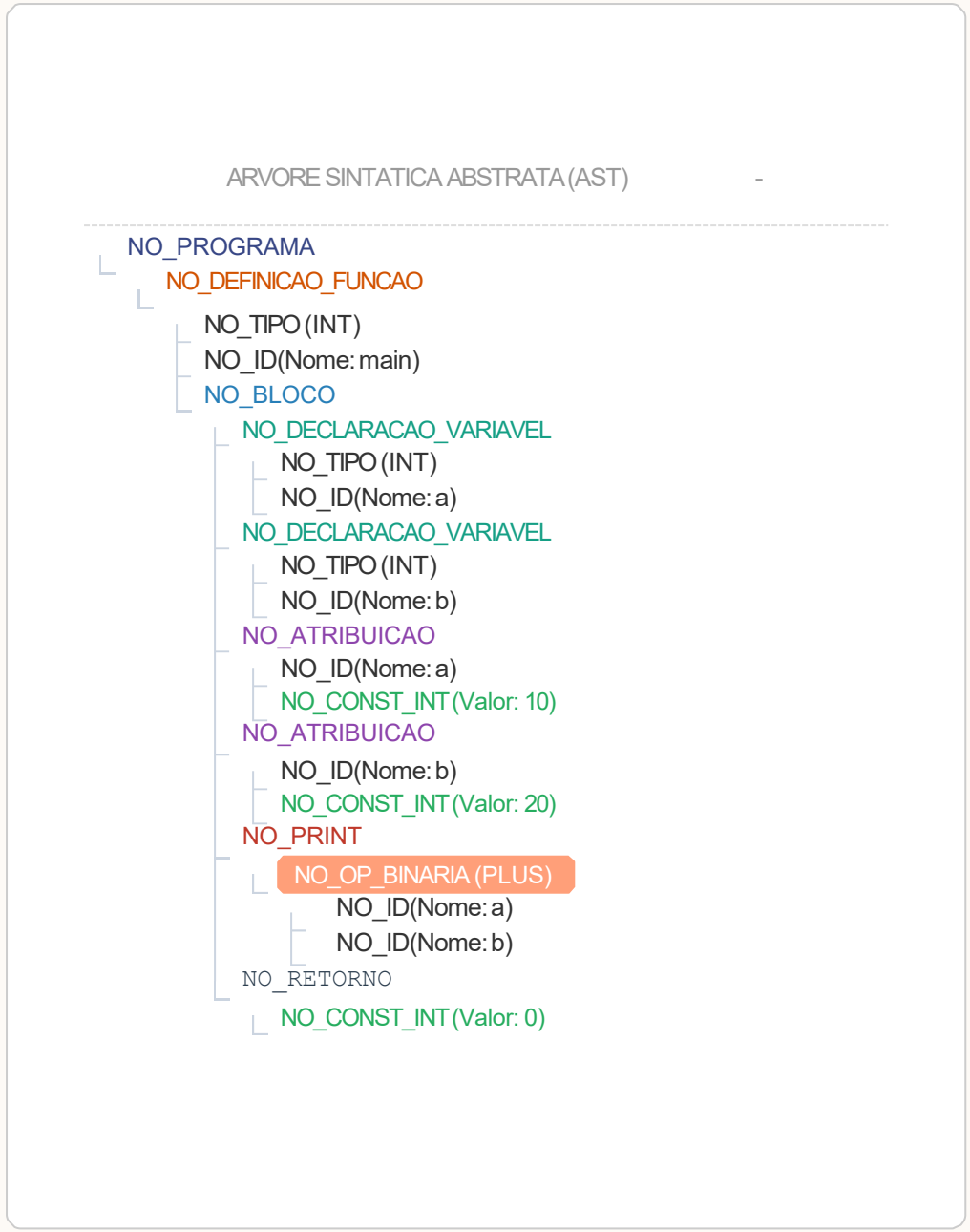
Fator (A Folha)
A unidade básica e final. Números e variáveis (`a`, `b`). Não se divide mais.

Termo (O Ramo)
As conexões internas que seguram as folhas. Resolve operações fortes (`*`, `/`) e se conecta à raiz.

Expressão (A Raiz)
A base de tudo. Agrupa os ramos e define o resultado final da soma (`+`, `-`).

Visualização: Note como o ramo da multiplicação (`b * c`) precisa ser resolvido antes de chegar na raiz da soma.

Fase 2 - Produto Final: A Estrutura Hierárquica



O Produto Limpo

A AST é a representação fiel do programa. cada declaração, atribuição e constante está mapeada como um nó específico.

Hierarquia de Escopo

Visualmente, vemos que NO_DECLARACAO e NO_ATRIBUICAO são irmãos e filhos do mesmo NO_BLOCO, definindo a ordem de execução.

Validação da Precedência

O Parser agrupou corretamente a operação: a soma (PLUS) é a raiz da sub-árvore, e os operandos a e b são seus filhos diretos.

O Mapa

Esta estrutura completa é o mapa que guiará a próxima fase (Análise Semântica) para validar tipos e escopos.

Fase 3 - Análise Semântica

O Objetivo

Validar se a estrutura gramatical (AST) possui coerência lógica e significado dentro das regras do Micro C.

O Cérebro do Compilador

Esta fase gera a **Tabela de Símbolos**, armazenando informações sobre cada identificador (como escopo, tipo e categoria).

Preparação para o Back-End

Além de validar, o analisador calcula e armazena o **Offset de Memória**. Sem isso, o gerador de Assembly não saberia onde guardar os dados na pilha.

Fase 3 - Regras de Integridade: Tipos e Limites

SITUAÇÃO A: JUIZ DE TIPOS

```
int x = 'A';
```

⚠ Erro: Tipos Incompatíveis

SITUAÇÃO B: INSPETOR DE VETORES

```
int v[10];    v[15] = 0;
```

⦸ Erro: Índice 15 excede tamanho 10

Tipagem Estrita

O compilador proíbe operações mistas (ex: somar `INT` com `STRING`) e atribuições inválidas, garantindo a consistência matemática dos dados.

Validação de Vetores

- **Categoria:** Impede o uso de vetor como variável simples (`v = 10` é proibido).
- **Checagem:** O analisador verifica se constantes de índice (`v[15]`) estouram o limite declarado, prevenindo corrupção de memória.

Fase 3 - Produto Final: O Contexto Semântico

ESC	NOME	TIPO	CATEGORIA	OFFSET (MEM)
1	a	INT	Var Local	-4 (%rbp)
1	b	INT	Var Local	-8 (%rbp)
0	main	INT	Função	LABEL

Mapeamento de Memória

A coluna **Offset** prova que o compilador já alocou espaço na pilha. a está em -4 e b logo abaixo em -8.

Gerenciamento de Escopo

A hierarquia está clara: main é 0, enquanto as variáveis vivem no escopo Local (1), protegidas de acesso externo.

Validação de Tipos

A operação a + b foi aprovada porque a tabela garante que ambos os nomes se referem a dados do tipo **INT**.

Fase 4 - Geração de IR

A Representação Intermediária como decisão de arquitetura.

Micro C (Alto Nível)

```
if (a < b)
```


IR (TAC - Baixo Nível)

```
t0    = a < b
if_false t0 goto L1
```

Linearização e Simplificação da lógica

Decisão de Arquitetura

A IR atua como uma **representação de baixo nível**. Ela recebe o Front-End (que valida o Micro C) e começa trabalhar no Back-End (que gera a IR e assembly).

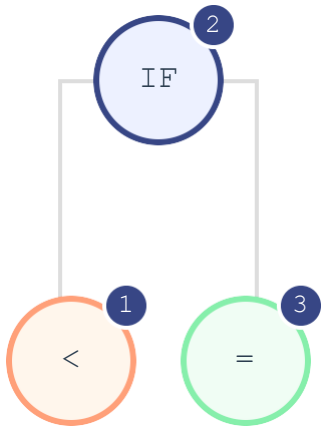
Independência de Máquina

O Código de Três Endereços (TAC) gerado é abstrato. Ele **não possui registradores físicos** (como %eax) nem instruções complexas de CPU, apenas lógica pura.

O Grande Diferencial: Portabilidade

Esta arquitetura permite que o mesmo Front-End gere código para **ARM** ou **RISC-V** no futuro, bastando apenas trocar o tradutor final, sem reescrever o compilador.

Fase 4 - Do Hierárquico ao Linear: A Tradução




```
t0  = a < b          (1)
if_false t0 goto L1  (2)
x  = 1               (3)
L1: (Ponto de Saída)
```

O Mecanismo: Tree Walker

O compilador visita os nós na ordem exata da execução: primeiro resolve a condição **(1)**, sobe para o IF **(2)** tomar a decisão, e se verdadeiro, desce para o bloco **(3)**.

Decomposição

Note que a árvore hierárquica é desmontada em passos simples. Cada nó visitado gera uma instrução correspondente na lista linear à direita.

Fase 4 - Produto Final: Código de Três Endereços

FONTE (soma.mcc)

```
int main()
  { int a,b;
  a = 10;
  b = 20;
   print(a + b);
  return 0;
}
```

SAÍDA (TAC)

```
main:
  a   = 10
  b   = 20
   t0    = a + b
   print_int t0
  return 0
```

Linearização

Note que cada linha contém no máximo **uma operação**, simplificando a vida do processador.

Variável Temporária (t0)

O compilador detectou a expressão a + b dentro do print e a extraiu.
Criou t0 para segurar o resultado, garantindo que a soma ocorra **antes** da impressão.

Resolução de I/O

A função genérica print foi substituída pelo opcode específico print_int, instruindo explicitamente o Back-End sobre qual rotina de sistema chamar.

Fase 5 - Geração de Assembly

Objetivo

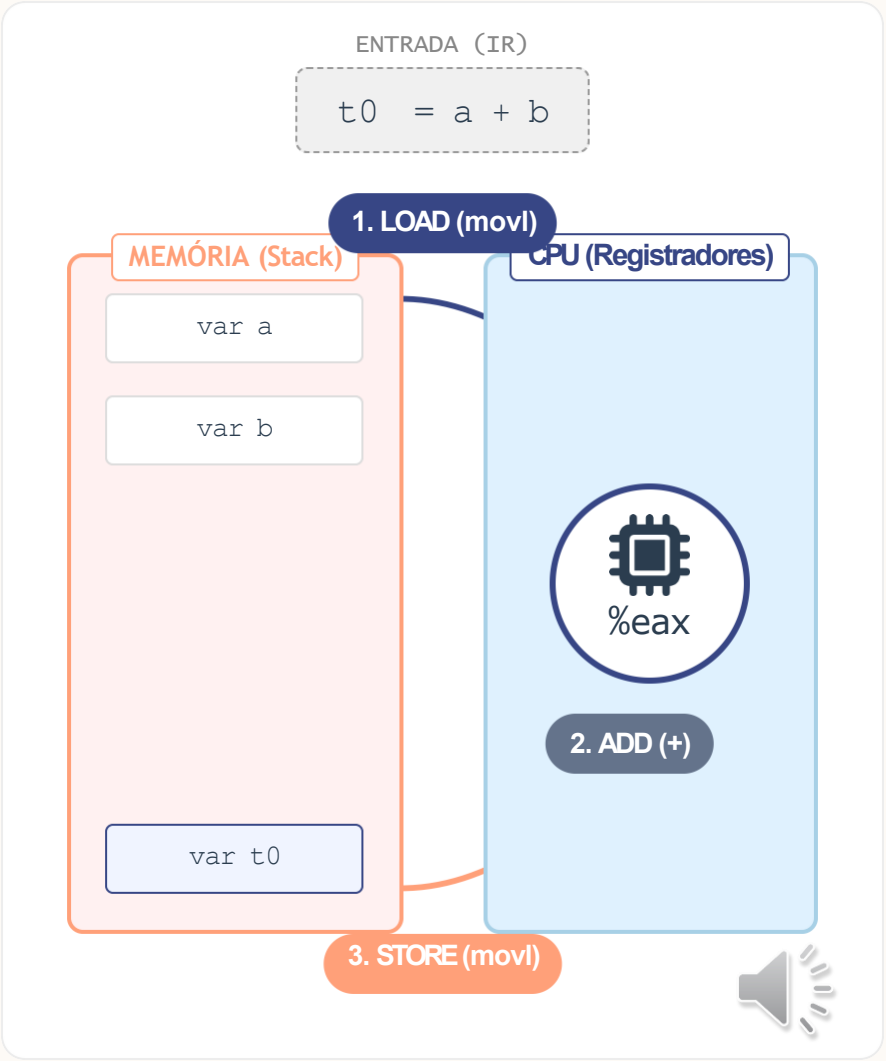
Traduzir a IR abstrata para instruções reais **x86-64**.

Sintaxe e Estratégia

Sintaxe AT&T :
A ordem correta é:
Instrução Origem, Destino

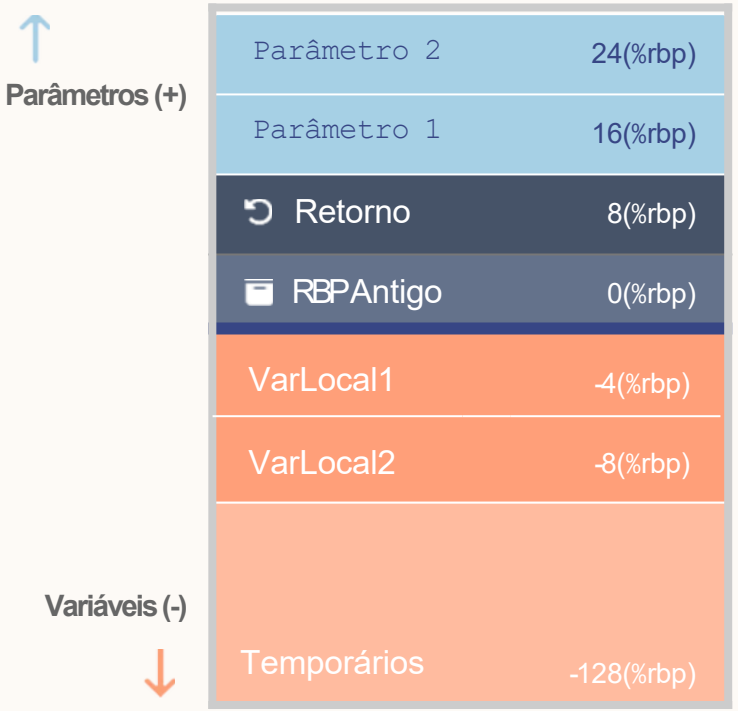
```
movl $10, %eax
```

Load-Operate-Store:
A CPU não soma memória diretamente: 1. Carregar (Load) 2. Calcular (Operate) 3. Salvar (Store)



Fase 5 - Anatomia do Stack Frame

Como o compilador organiza dados e fluxo de execução na memória.



O Ponto de Referência (%rbp)

O *Base Pointer* é o marco zero. Todo acesso é relativo a ele, evitando confusão quando a pilha cresce ou diminui.

A Zona Reservada

Existe uma barreira vital de 16 bytes que não usamos para dados:

- 8(%rbp): Guarda o endereço para onde a CPU deve pular ao fim da função.
- 0(%rbp): Salva o contexto anterior, permitindo que a função continue de onde parou.

Geografia dos Dados

↑ **Acima (+):** Parâmetros empilhados por quem chamou.

↓ **Abaixo (-):** Variáveis criadas pela função atual.

Fase 5 - Produto Final: Código Assembly

Saída Completa (soma.s)

```
.section .rodata
.L.str.int: .string "%d\n"
.L.str.char: .string "%c\n"
.L.str.str: .string "%s\n"

.text .globl main
main:
  pushq %rbp
  movq %rsp, %rbp
  subq $256, %rsp

  movl $10, %eax
  movl %eax, -4(%rbp)
  movl $20, %eax
  movl %eax, -8(%rbp)
  movl -4(%rbp), %eax
  movl -8(%rbp), %ebx
  addl %ebx, %eax
  movl %eax, -128(%rbp)

  leaq .L.str.int(%rip), %rdi
  movl -128(%rbp), %esi
  xorl %eax, %eax
  subq $8, %rsp
  call printf
  addq $8, %rsp

  movl $0, %eax
  movq %rbp, %rsp
  popq %rbp
  ret
```

Dados Estáticos
Strings de formatação (`%d`, `%c`, `%s`) geradas automaticamente na seção `.rodata`.

Prólogo
Configuração do Stack Frame com 256 bytes para variáveis locais e temporários.

Lógica
Operação `Load-Operate-Store` completa: carrega de `-4` e `-8`, soma em `%eax` e salva em `-128`.

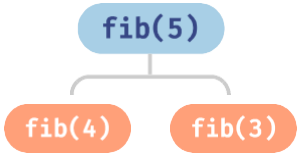
ABI System V
Argumentos em `%rdi` e `%esi`. Alinhamento de pilha (`subq $8`) antes da chamada `call printf`.

Resultados e Validação

Validação de Sucesso

Bubble Sort

Valida Laços Aninhados e acesso a Arrays.



Fibonacci

Valida Recursão Múltipla e Stack Frame.

Testes de Falha

Fase 1 (Léxico)

```
int valor = 10 @;
```

```
Erro Lexico: Token indefinido '@' encontrado na linha 11
```

Fase 2 (Sintático)

```
int a = 5
```

```
Erro Sintatico na linha 9: Token inesperado. Esperado: SEMICOLON
```

Fase 3 (Semântico)

```
int x = "texto";
```

```
Erro Semantico na linha 7: Atribuicao de tipos incompativeis.
```

> O compilador Micro C é funcional e robusto na detecção de erros, cumprindo seu papel em ser didático.

Trabalhos Futuros

Evoluindo de ferramenta didática para um compilador robusto.

Otimização de Código

Implementação de algoritmos na IR (ex: eliminação de código morto) e **Alocação de Registradores**, superando a estratégia ingênua de Load-Operate-Store.

Poder da Linguagem (Ponteiros)

Suporte completo a aritmética de ponteiros e alocação dinâmica , habilitando estruturas de dados complexas como Listas e Árvores.

Expansão de Tipos

Adição de tipos de números de ponto flutuante como float e double.

Independência Total

Desenvolvimento de um **Montador** e **Linker** próprios, eliminando a dependência do GCC para gerar o binário final.

Aarquitetura modular da IR facilita a expansão futura para outras arquiteturas de hardware, como ARM e RISC-V.

Micro C: Uma Ferramenta para o Ensino

```
int a = "texto";
```

PADRÃO (GCC)

```
warning:
initialization makes
integer from pointer
without a cast
```

Norma Técnica

MICRO C (TUTOR)

```
Erro: Tipos
incompatíveis.
Esperado: INT
Encontrado:
STRING
```

Foco no Aluno

Mensagens Humanizadas

Diferente de compiladores industriais que priorizam velocidade, o Micro C prioriza a **clareza**. Ele explica *o que* deu errado e *por que*, ajudando o iniciante a corrigir a lógica.

Análise com Feedback

A fases do front-end atua como um analisador rigoroso, ensinando conceitos de **Tipagem** e **Escopo** antes mesmo do programa rodar.

Futuro: Otimização Educativa

A eliminação de **Código Morto** na IR não apenas limpará o binário, mas gerará alertas pedagógicos:
"Atenção: O código após o 'return' nunca será executado."

Transformar o compilador de uma ferramenta que apenas cospe binários em uma ferramenta que ensina boas práticas.

Considerações Finais

Democratização do Conhecimento

Quebrar a barreira de entrada de temas complexos como Compiladores, tornando o impossível acessível e compreensível para qualquer estudante.

Educação Acessível

O Micro C cumpre seu propósito ao transformar a teoria abstrata em prática tangível. Aprendemos a programar não apenas usando, mas **construindo** a ferramenta.

Contribuição Open Source

Um compilador completo com código-fonte aberto (GPLv3), disponível para estudo, modificação e evolução por toda a comunidade.

> **Deixamos de ser apenas usuários de ferramentas para nos tornarmos criadores delas.**

Acessibilidade do Código

Comparativo de Linhas de Código & Curva de Aprendizado

~4k

MICRO C

Leve o suficiente para ser lido, entendido e modificado em **um único semestre**.

LLVM

~35.5 Milhões

GCC

~15 Milhões

Inviável para ser didático

A simplicidade do Micro C não é uma falta de recurso, é o seu maior trunfo pedagógico.

Obrigado!

Perguntas?