# MAB 471 - Compiladores I
# Introdução

http://www.dcc.ufrj.br/~fabiom/comp

# Compiladores

- O que é um compilador?
  - Um programa que traduz um programa executável em uma linguagem em um programa executável em outra linguagem
  - O compilador deve melhorar de alguma forma o programa
- O que é um interpretador?
  - Um programa que lê um programa executável e produz o resultado da execução desse programa
- C é tipicamente compilada, PHP é tipicamente interpretada
- Java é compilado para bytecodes (código para VM Java)
  - podem ser interpretados
  - ou compilados
    - → Compilação Just-in-time

Erro comum:
X é uma linguagem interpretada (ou compilada)

# Por que estudar Compiladores?

- Compiladores são importantes
  - Responsáveis por vários aspectos do desempenho de sistemas
  - Aproveitar o hardware tem ficado mais difícil
    - In 1980, conseguia-se 85% ou mais do desempenho máximo
    - Hoje esse número está mais para 5 a 10% do máximo
    - O compilador tem grande influência no desempenho
- Compiladores são interessantes
  - Incluem muitas aplicações práticas de aspectos teóricos
  - Expõem questões algorítmicas e de engenharia
- Compiladores estão em todo lugar
  - Muitas aplicações têm linguagens embutidas
    - Comandos, macros, formatação...
  - Muitos formatos de arquivo parecem linguagens

# Por que estudar Compiladores?

- Construção de compiladores usa ideias de muitas áreas da computação

| | |
|---|---|
| Inteligência Artificial | Algoritmos gulosos<br>Busca heurística |
| Algoritmos | Algoritmos de grafos, union-find<br>Programação dinâmica |
| Teoria | DFAs, PDAs, casamento de padrão<br>Algoritmos de ponto fixo |
| Sistemas | Alocação, nomes, sincronização,<br>localidade, concorrência |
| Arquitetura | Gerenciamento do pipeline<br>Uso do conjunto de instruções |

# Por que isso importa hoje?

Todo computador atualmente é multiprocessado

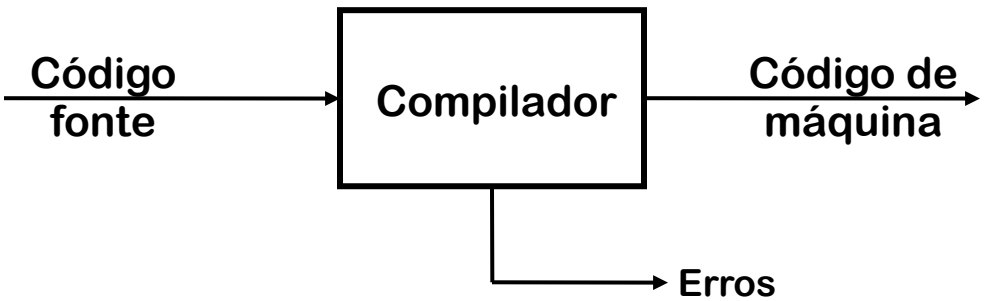
- A era dos ganhos de clock está acabando
  - Consumo de energia proibitivo (quadrático em relação ao clock)
  - Fios menores -> maior resistência -> maior consumo
- Melhor desempenho virá através de múltiplas cópias de um mesmo processador (núcleo) em um único chip
  - Programas em linguagens tradicionais não conseguem aproveitar bem esse nível de paralelismo
    - → Linguagens paralelas, alguns sistemas OO concorrentes, linguagens funcionais
  - Programas paralelos precisam de compiladores sofisticados

# Linguagens precisam de compiladores

It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

— John Backus sobre o primeiro compilador FORTRAN

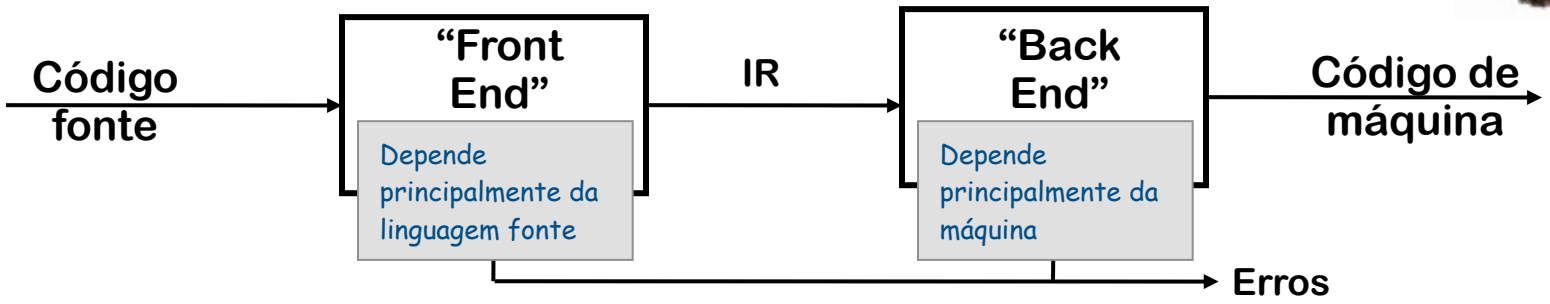
# Visão de alto nível de um compilador




## Implicações

- Devem reconhecer programas legais (e ilegais)
- Deve gerar código correto
- Deve gerenciar o armazenamento das variáveis (e código)
- Deve concordar com o SO e linker sobre o formato de código objeto
- Grande avanço em relação à linguagem de montagem—notação de alto nível

# Compilador de duas partes



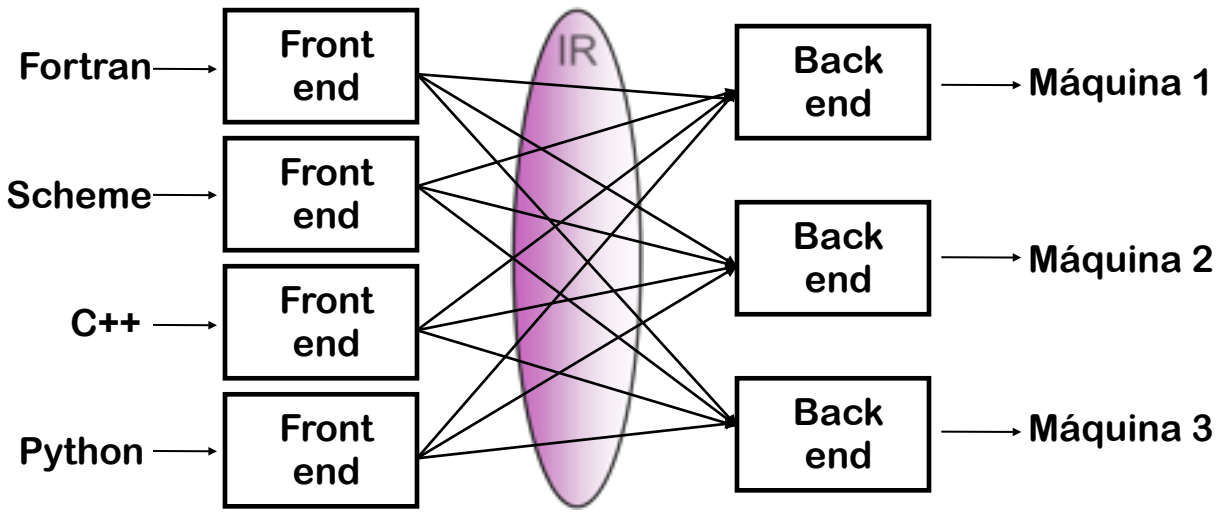

## Implicações

- Uso de uma representação intemerdiária (IR)
- "Front end" mapeia fonte em IR
- "Back end" mapeia IR em código de máquina
- Pode ter múltiplas passadas no front e back ends

Princípio clássico de Eng. de Software: Separação de Interesses

Tipicamente o front end é O(n) ou O(n log n), e o back end é NPC

# O Santo Graal



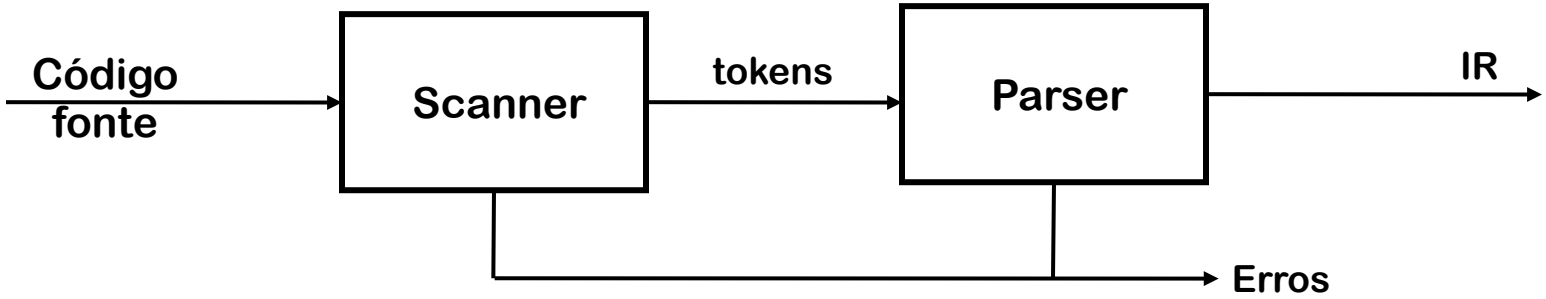

Podemos fazer n x m compiladores com n+m componentes?

- Deve codificar conhecimento específico de cada linguagem em cada front end
- Deve codificar todas as características em um único IR
- Deve codificar conhecimento específico das máquinas em cada back end

Bem sucedido em sistemas com IRs no nível de assembler

ex: rtl do gcc ou ir llvm

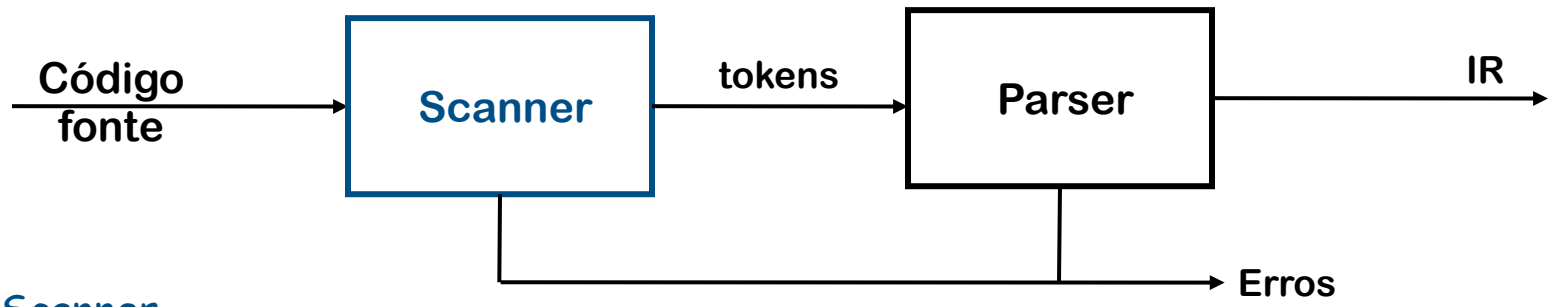
# O Front End




## Responsabilidades

- Reconhecer programas legais (e ilegais)
- Dar erros úteis ao usuário
- Produzir IR e mapa preliminar de alocação
- Formatar código para o resto do compilador
- Muito da construção do front end pode ser automatizada

# O Front End



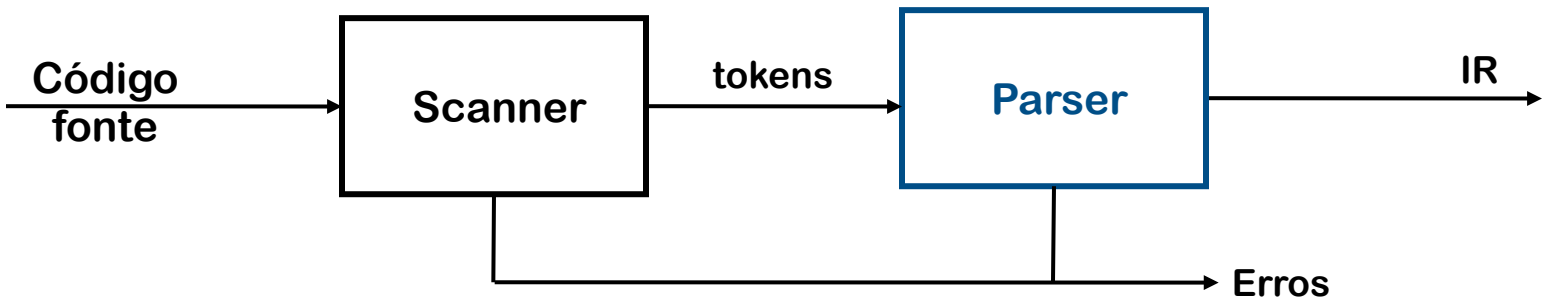

## Scanner

- Mapeia caracteres em palavras—a unidade básica da sintaxe
- Produz pares — uma palavra e sua categoria sintática
- x = x + y ; vira <id,x> = <id,x> + <id,y> ;
  - palavra ≅ lexeme, categoria sintática ≅ tipo do token, par ≅ um token
- Tokens típicos incluem números, identificadores, +, -, new, while, if
- Velocidade é importante

Livros texto advogam o uso de geradores de scanners

Vários compiladores reais usam scanners escritos à mão para maior desempenho e controle

# O Front End




## Parser

- Reconhece sintaxe livre de contexto e reporta erros
- Guia análise sensível ao contexto ("análise semântica"/checagem de tipos)
- Constrói IR para programa fonte

Relativamente fácil de escrever à mão (mais que o scanner)

A maioria dos livros advoga o uso de um gerador

# O Front End

Sintaxe livre de contexto é especificada com uma gramática

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid num$$

Essa gramática define o conjunto de expressões aritméticas simples

Escrita numa variante da Backus–Naur Form (BNF)

Formalmente, em uma gramática G = (S,N,T,P)

- S é o símbolo inicial
- N é um conjunto de símbolos não-terminais
- T é um conjunto de símbolos terminais (ou palavras)
- P é um conjunto de produções ou regras de reescrita
  - $(P : N \rightarrow N \cup T)$

# O Front End

## Outra gramática mais complexa

1. S → Expr
2. Expr → Expr Op Termo
3. | Term
4. Termo → num
5. | id
6. Op → +
7. | -

S = S

T = { num, id, +, - }

N = { S, Expr, Termo, Op }

P = { 1, 2, 3, 4, 5, 6, 7 }

- Expressões aditivas simples sobre "num" e "id"
- Essa gramática, como as outras que veremos nesse curso, é parte da classe das CFGs (gramáticas livres de contexto)

# O Front End

Dada uma CFG, podemos derivar frases por substituição

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

Produção Resultado

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

| Produção | Resultado |
|---|---|
| | S |

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

| Produção | Resultado |
|---|---|
| | S |
| 1 | Expr |

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

| Produção | Resultado |
|---|---|
| | S |
| 1 | Expr |
| 2 | Expr Op Termo |

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

| Produção | Resultado |
|---|---|
| | S |
| 1 | Expr |
| 2 | Expr Op Termo |
| 5 | Expr Op y |

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

| Produção | Resultado |
|---|---|
| | S |
| 1 | Expr |
| 2 | Expr Op Termo |
| 5 | Expr Op y |
| 7 | Expr - y |

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

| Produção | Resultado |
|---|---|
| | S |
| 1 | Expr |
| 2 | Expr Op Termo |
| 5 | Expr Op y |
| 7 | Expr - y |
| 2 | Expr Op Termo - y |

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

| Produção | Resultado |
|---|---|
| | S |
| 1 | Expr |
| 2 | Expr Op Termo |
| 5 | Expr Op y |
| 7 | Expr - y |
| 2 | Expr Op Termo - y |
| 4 | Expr Op 2 - y |

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

| Produção | Resultado |
|---|---|
| | S |
| 1 | Expr |
| 2 | Expr Op Termo |
| 5 | Expr Op y |
| 7 | Expr - y |
| 2 | Expr Op Termo - y |
| 4 | Expr Op 2 - y |
| 6 | Expr + 2 - y |

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

| Produção | Resultado |
|---|---|
| | S |
| 1 | Expr |
| 2 | Expr Op Termo |
| 5 | Expr Op y |
| 7 | Expr - y |
| 2 | Expr Op Termo - y |
| 4 | Expr Op 2 - y |
| 6 | Expr + 2 - y |
| 3 | Termo + 2 - y |

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

| Produção | Resultado |
|---|---|
| | S |
| 1 | Expr |
| 2 | Expr Op Termo |
| 5 | Expr Op y |
| 7 | Expr - y |
| 2 | Expr Op Termo - y |
| 4 | Expr Op 2 - y |
| 6 | Expr + 2 - y |
| 3 | Termo + 2 - y |
| 5 | x + 2 - y |

1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Dada uma CFG, podemos derivar frases por substituição

| Produção | Resultado |
|---|---|
| | S |
| 1 | Expr |
| 2 | Expr Op Termo |
| 5 | Expr Op y |
| 7 | Expr − y |
| 2 | Expr Op Termo - y |
| 4 | Expr Op 2 - y |
| 6 | Expr + 2 - y |
| 3 | Termo + 2 - y |
| 5 | x + 2 - y |

Uma derivação

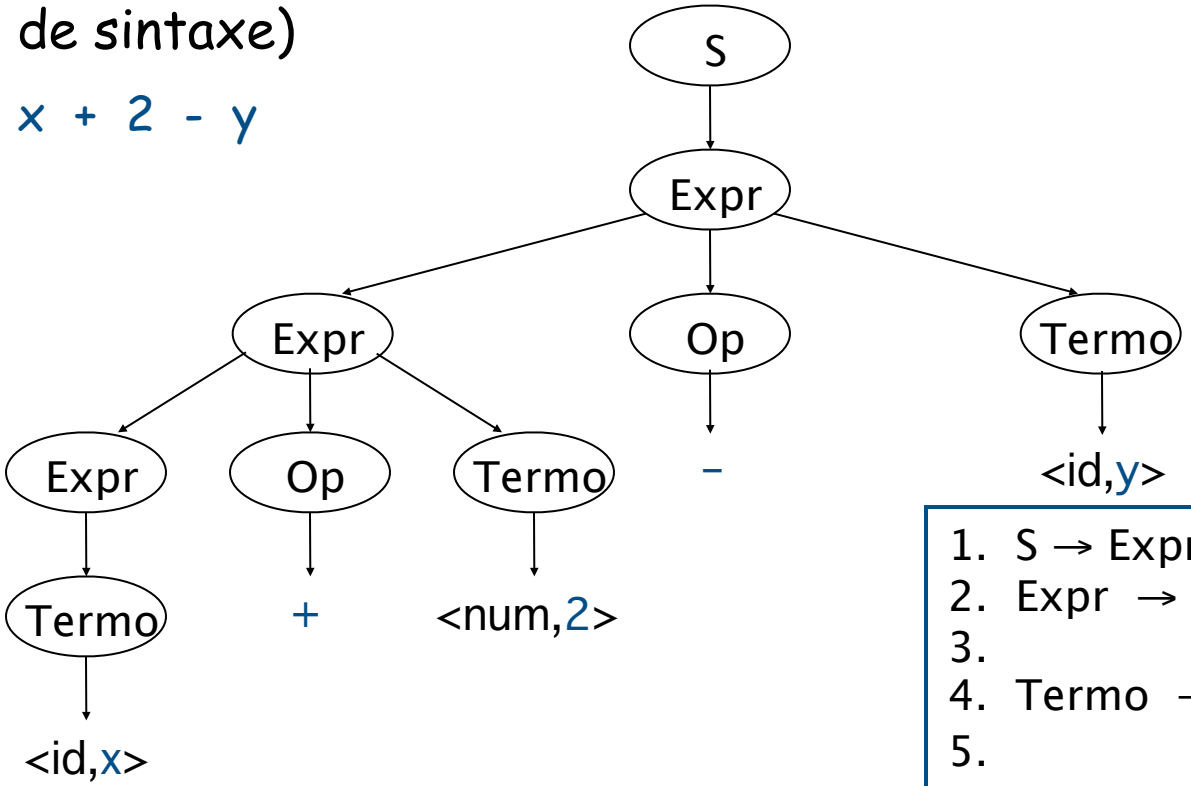
1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Term → num
5. | id
6. Op → +
7. | -

Para reconhecer uma frase válida em alguma CFG revertemos esse processo para construir um casamento

# O Front End

Um casamento pode ser representado por uma árvore (a árvore de sintaxe)

x + 2 - y

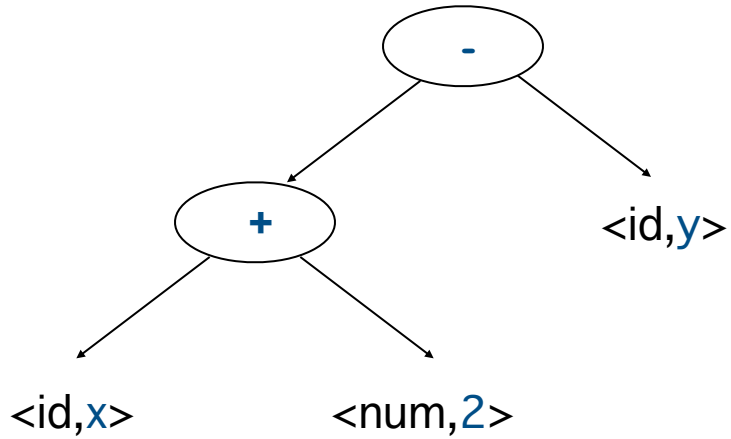



1. S → Expr
2. Expr → Expr Op Termo
3. | Termo
4. Termo → num
5. | id
6. Op → +
7. | -

Contém muita informação desnecessária

# O Front End

Compiladores normalmente usam uma "árvore sintática abstrata" (AST) ao invés de uma árvore de sintaxe



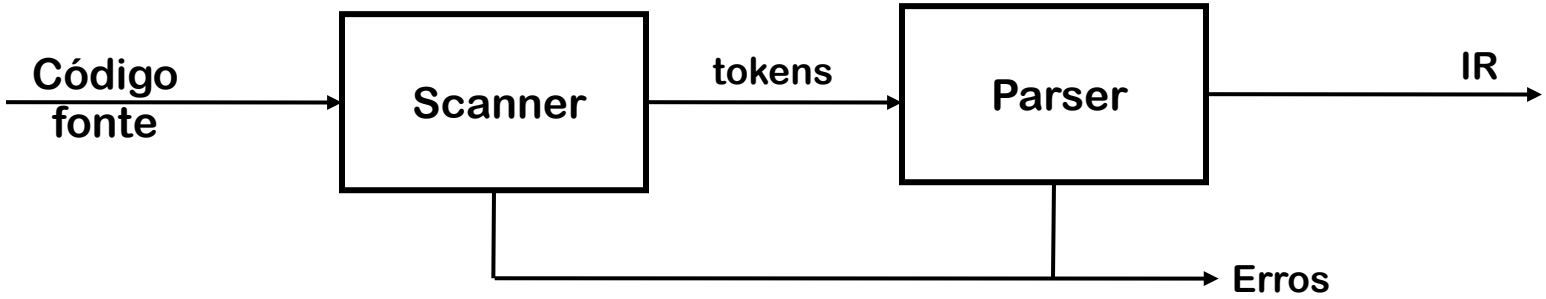

A AST resume a estrutura gramática, sem incluir os detalhes da derivação

Muito mais conciso

ASTs são um tipo de representação intermediária (IR)

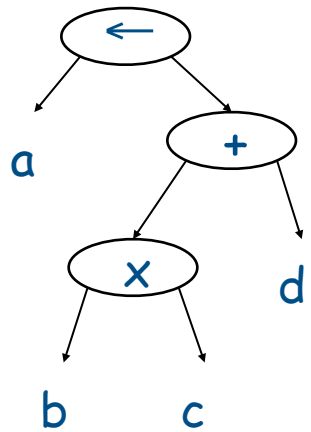
Alguns acham AST a IR "natural".

# O Front End



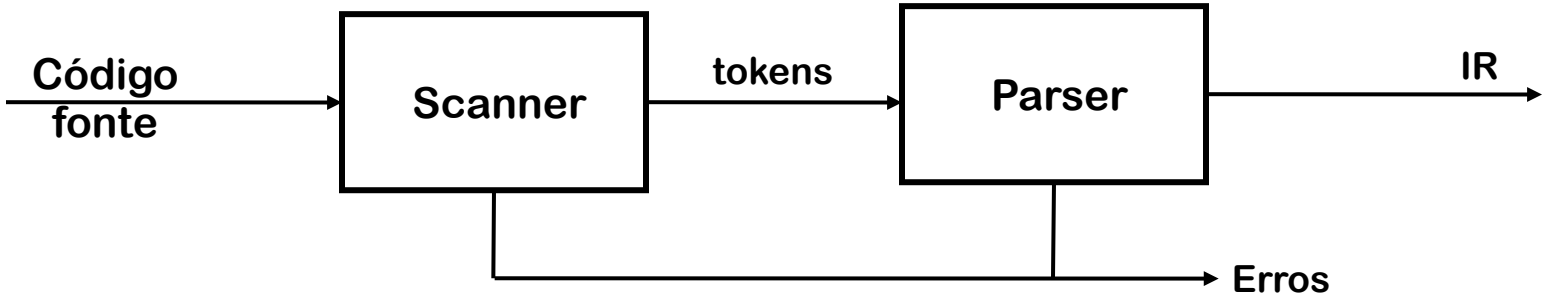

O Formato do Código determina muitas propriedades do programa resultante

a ← b x c + d


Relembre a diferença no uso dos registradores da aula passada!

# O Front End

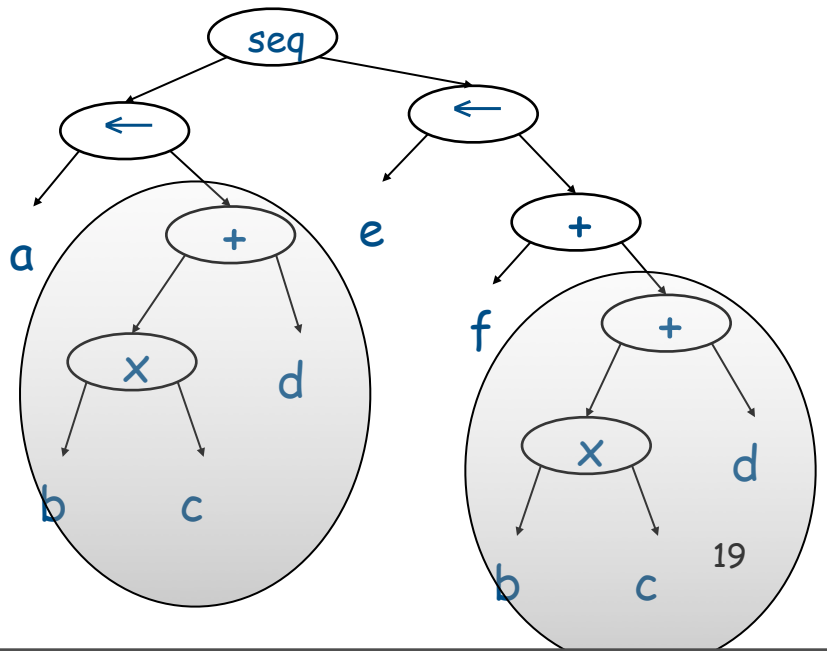



O Formato do Código determina muitas propriedades do programa resultante

a ← b x c + d

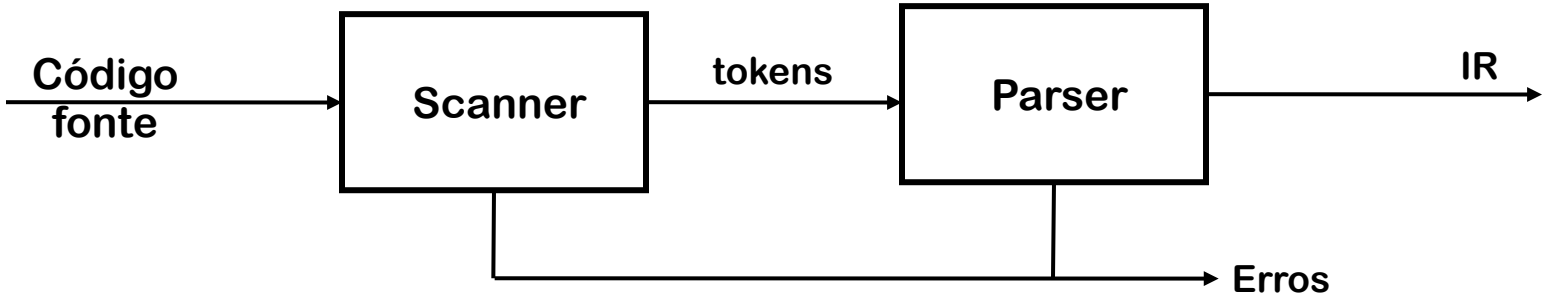
e ← f + b x c + d

vira




Se você transformar essa AST em código provavelmente vai ter duplicação.

# O Front End




O Formato do Código determina muitas propriedades do programa resultante

a ← b x c + d

e ← f + b x c + d

vira

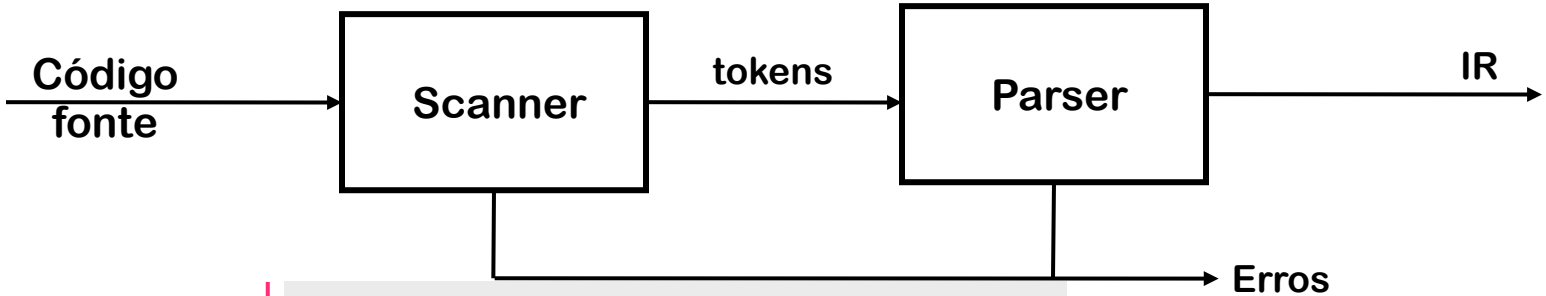
```
load @b ⇒ r1
load @c ⇒ r2
mult r1,r2 ⇒ r3
load @d ⇒ r4
add r3,r4 ⇒ r5
store r5 ⇒ @a
load @f ⇒ r6
add  r5,r6 ⇒ r7
store r7 ⇒ @e
```

calcula b x c + d

reusa b x c + d

Gostaríamos de produzir esse código, mas fazer isso corretamente requer bastante esforço!

# O Front End




"a" é distinto de "b", "c", e "d" ?

O Formato do Código determina muitas propriedades do programa resultante

a ← b x c + d

e ← f + b x c + d

vira

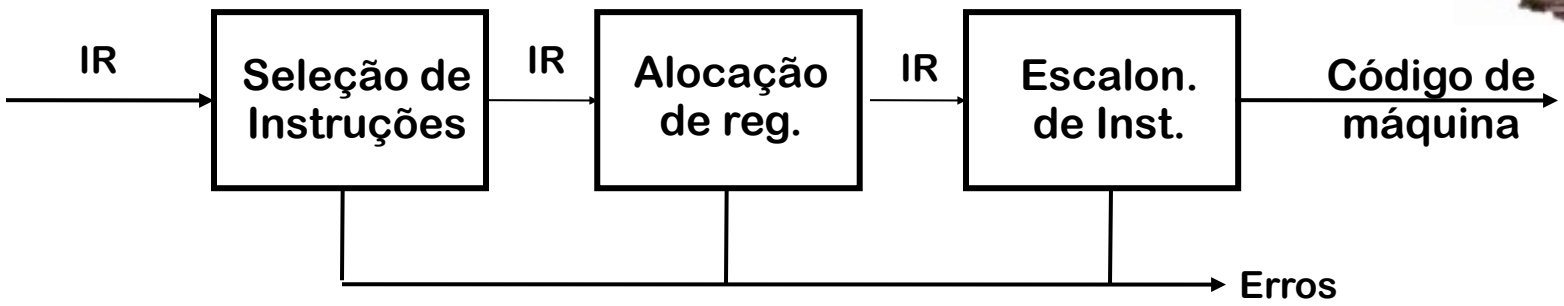
```
load @b ⇒ r1
load @c ⇒ r2
mult r1,r2 ⇒ r3
load @d ⇒ r4
add r3,r4 ⇒ r5
store r5 ⇒ @a
load @f ⇒ r6
add  r5,r6 ⇒ r7
store r7 ⇒ @e
```

calcula b x c + d

reusa b x c + d

Gostaríamos de produzir esse código, mas fazer isso corretamente requer bastante esforço!

# O Back End

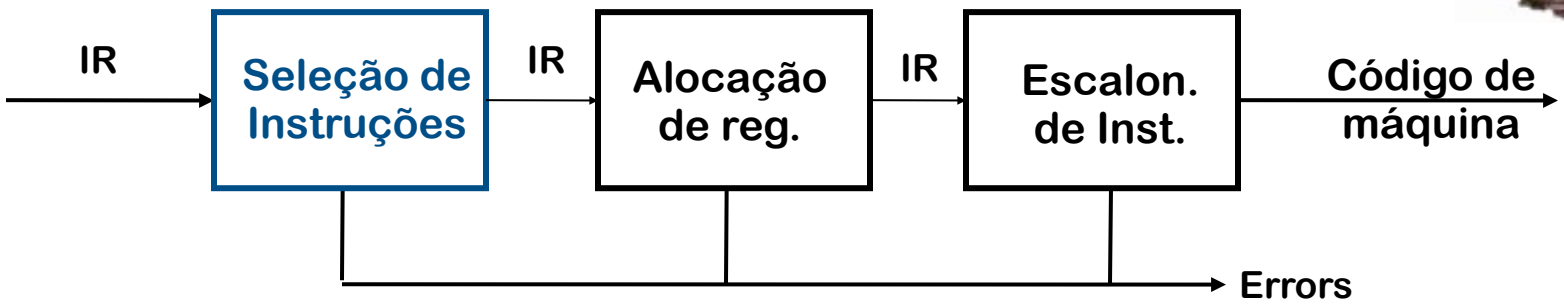



## Responsabilidades

- Traduz IR em código de máquina
- Escolhe instruções para implementar cada operação da IR
- Decide quais valores manter em registradores
- Garante conformidade com interfaces do SO

Alguma automação, mas bem menos que no front-end

# O Back End



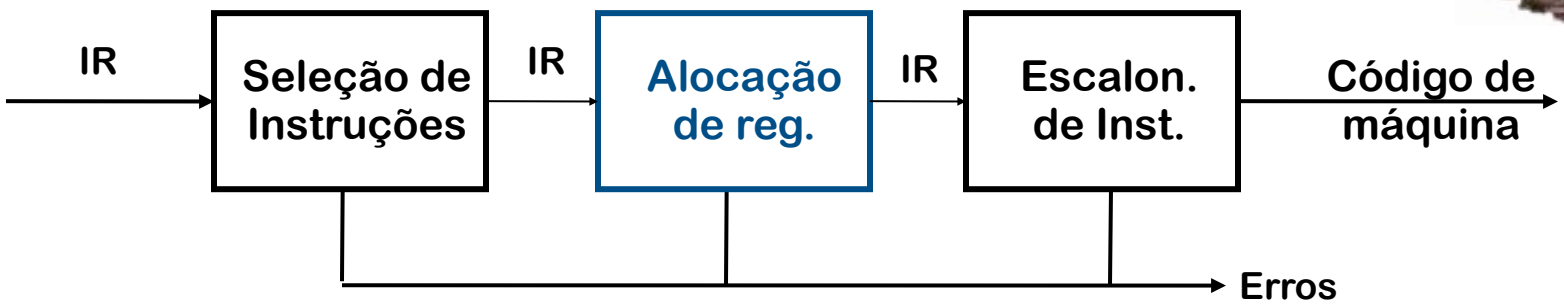

## Seleção de Instruções

- Produz código rápido e compacto
- Aproveita recursos da máquina como modos de endereçamento
- Normalmente visto como um problema de casamento de padrões
  - métodos ad hoc, casamento de padrões, programação dinâmica
  - Forma da IR influencia escolha da ténica

Perdeu importância com arquiteturas modernas

  - Processadores eram mais complicados
  - Ortogonalidade dos processadores RISC simplificou esse problema

# O Back End

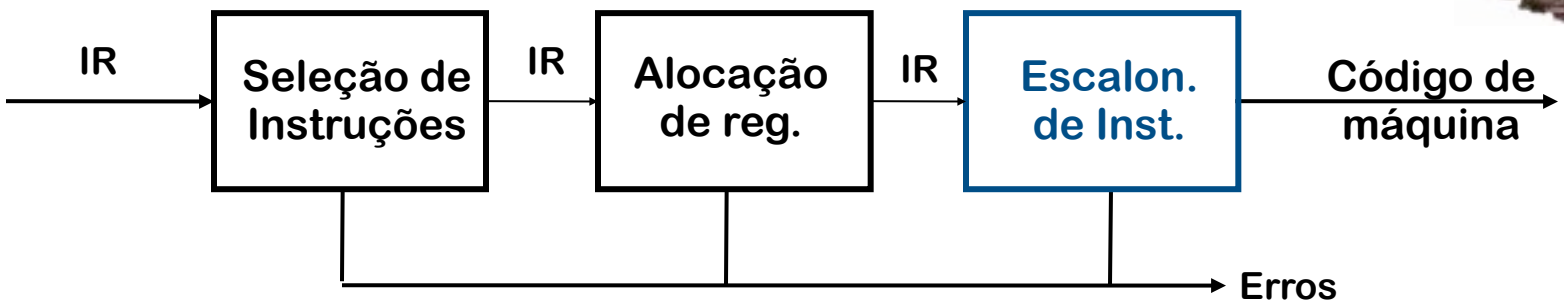



## Alocação de Registradores

- Ter cada valor em um registrador quando for usado
- Gerenciar um conjunto limitado de recursos
- Pode mudar a escolha de instruções e inserir LOADs e STOREs (afeta seleção e escalonamento)
- Alocação ótima é NP-completa na maioria dos casos

Compiladores usam soluções aproximadas

# O Back End



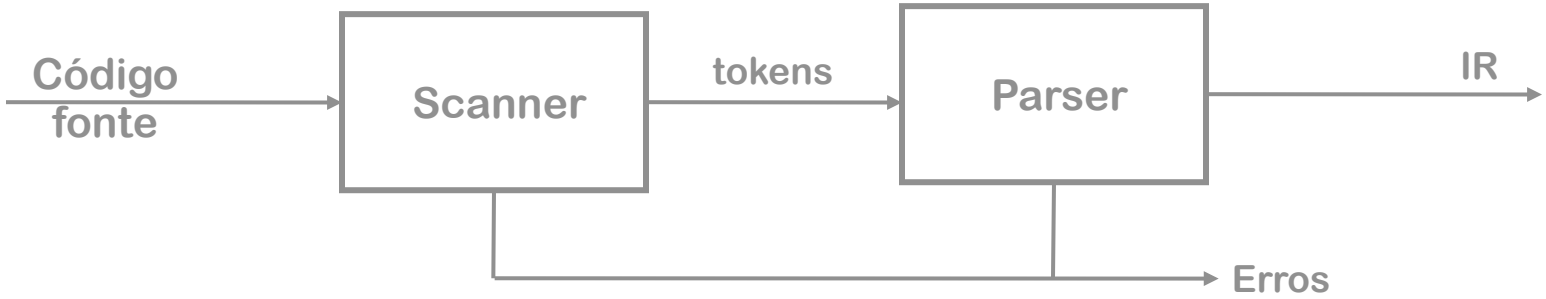

## Escalonamento de Instruções

- Evita paradas no pipeline do processador
- Usa todas as unidades do processador produtivamente
- Pode aumentar o tempo de vida de variáveis, afetando a alocação de registradores

Escalonamento ótimo é NP-Completo em quase todos os casos

Algumas heurísticas bem desenvolvidas

# O Front End




a ← b x c + d
e ← f + b x c + d

vira

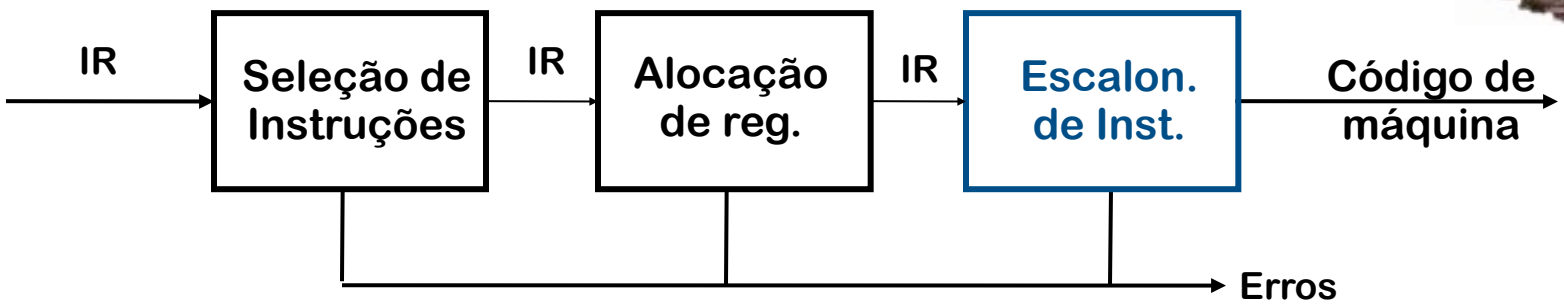
load @b ⇒ $r_1$
load @c ⇒ $r_2$
mult $r_1$,$r_2$ ⇒ $r_3$
load @d ⇒ $r_4$
add $r_3$,$r_4$ ⇒ $r_5$
store $r_5$ ⇒ @a
load @f ⇒ $r_6$
add $r_5$,$r_6$ ⇒ $r_7$
store $r_7$ ⇒ @e

calcula b x c + d

reusa b x c + d

Lembram desse exemplo de alguns slides atrás?

# O Back End



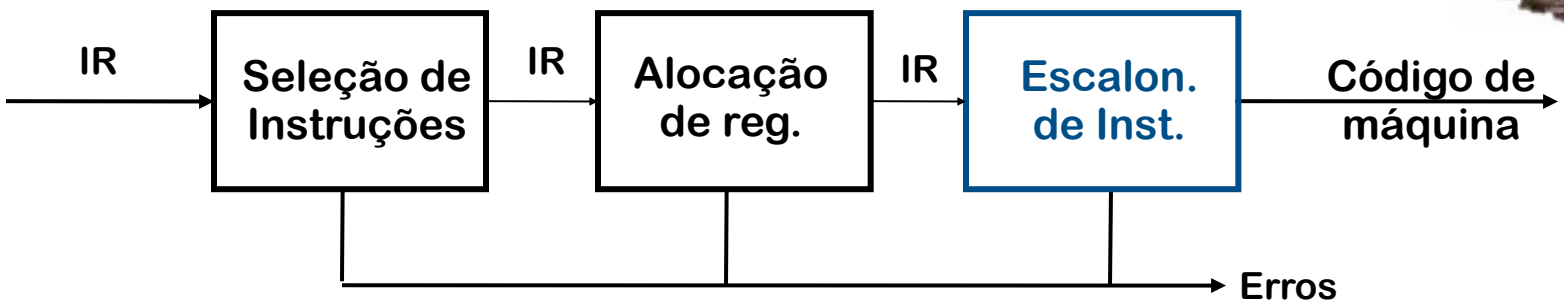

## Escalonamento de Instruções

| unidade 1 | unidade2 |
|---|---|
| load @b ⇒ $r_1$ | load @c ⇒ $r_2$ |
| load @d ⇒ $r_4$ | load @f ⇒ $r_6$ |
| mult $r_1$,$r_2$ ⇒ $r_3$ | nop |
| add $r_3$,$r_4$ ⇒ $r_5$ | nop |
| store $r_5$ ⇒ @a | nop |
| add $r_5$,$r_6$ ⇒ $r_7$ | nop |
| store $r_7$ ⇒ @e | nop |

Esse escalonamento carrega agressivamente valores em registradores pra esconder a latência da memória.

Termina a computação o mais cedo possível, assumindo 2 ciclos para load e store, e 1 ciclo pro resto.

# O Back End



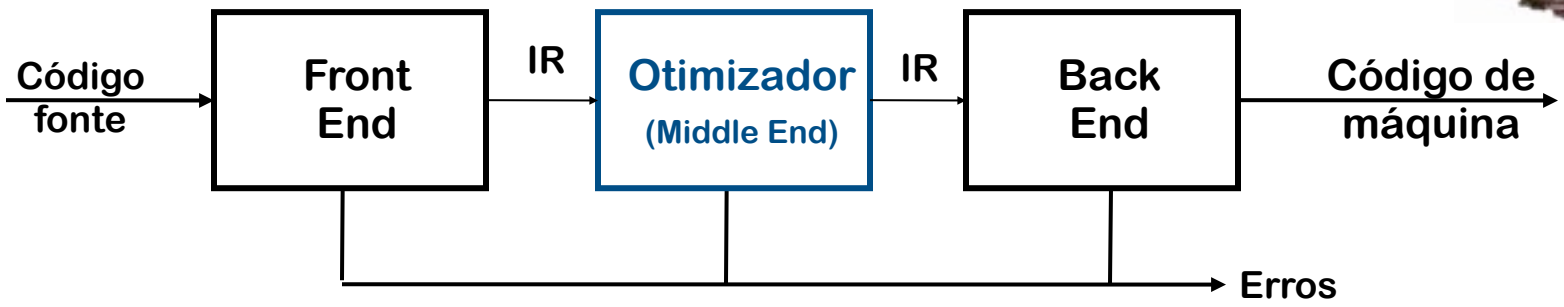

## Escalonamento de Instruções

Mesmo tempo, usa menos registradores

| unit 1 | unit 2 |
|---|---|
| load @b ⇒ $r_1$ | load @c ⇒ $r_2$ |
| load @d ⇒ $r_4$ | load @f ⇒ $r_6$ |
| mult $r_1$,$r_2$ ⇒ $r_3$ | nop |
| add $r_3$,$r_4$ ⇒ $r_5$ | nop |
| store $r_5$ ⇒ @a | nop |
| add $r_5$,$r_6$ ⇒ $r_7$ | nop |
| store $r_7$ ⇒ @e | nop |

| unit 1 | unit 2 |
|---|---|
| load @b ⇒ $r_1$ | load @c ⇒ $r_2$ |
| load @d ⇒ $r_4$ | nop |
| mult $r_1$,$r_2$ ⇒ $r_3$ | nop |
| add $r_3$,$r_4$ ⇒ $r_5$ | load @f ⇒ $r_6$ |
| store $r_5$ ⇒ @a | nop |
| add $r_5$,$r_6$ ⇒ $r_7$ | nop |
| store $r_7$ ⇒ @e | nop |

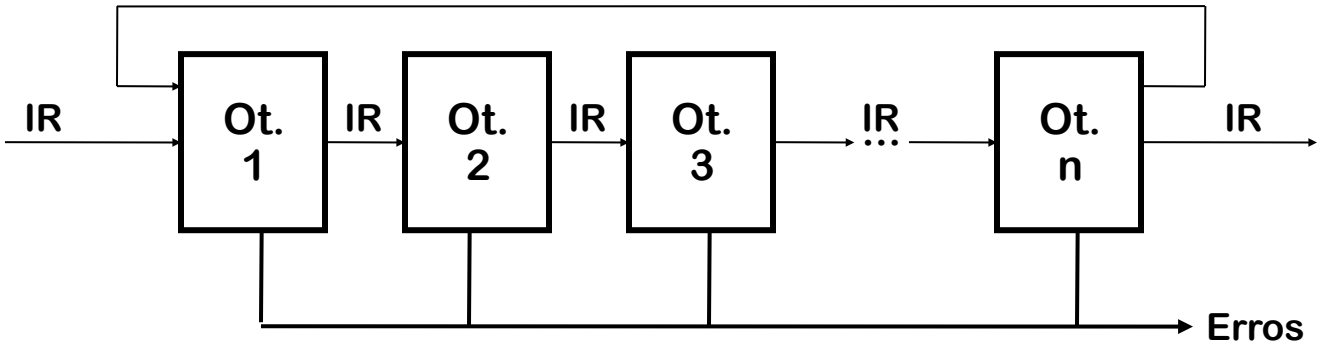
# Compilador de três partes




## Melhoria de Código (ou Otimização)

- Analisa IR e reescreve (ou transforma) IR
- Meta principal é reduzir tempo de execução do código compilado
  - Mas também pode melhorar tamanho, consumo de energia, …
- Deve preservar "semântica" do código
  - Medido pelos valores das variáveis

# O Otimizador (ou Middle End)



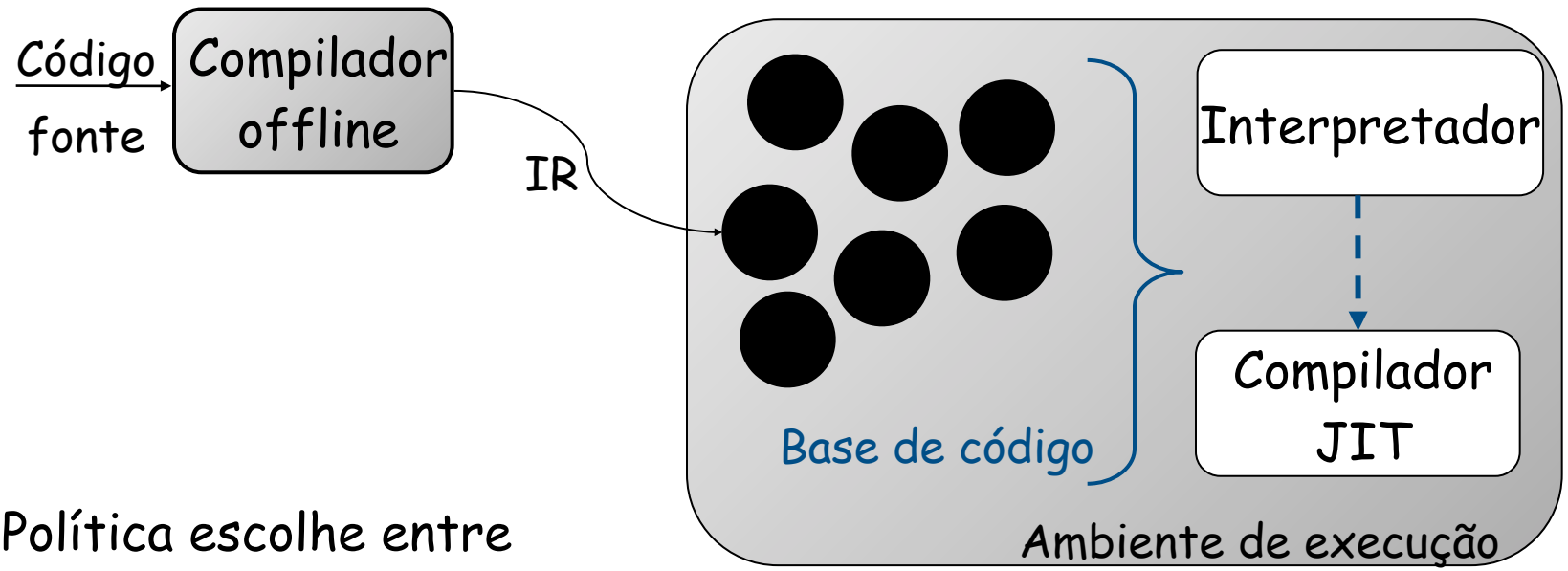

**Otimizadores modernos estruturados como uma série de passadas**

## Transformações Típicas

- Descobrir e propagar algum valor constante
- Mover uma computação para um lugar menos executado
- Especializar alguma computação baseada no contexto
- Descobrir e eliminar computação redundante
- Remover código inútil ou inalcançável
- Codificar um idioma em alguma forma particularmente eficiente

# Compilação em tempo de execução

Sistemas como HotSpot (Java) e V8 (JavaScript) usam de compilação e otimização em tempo de execução



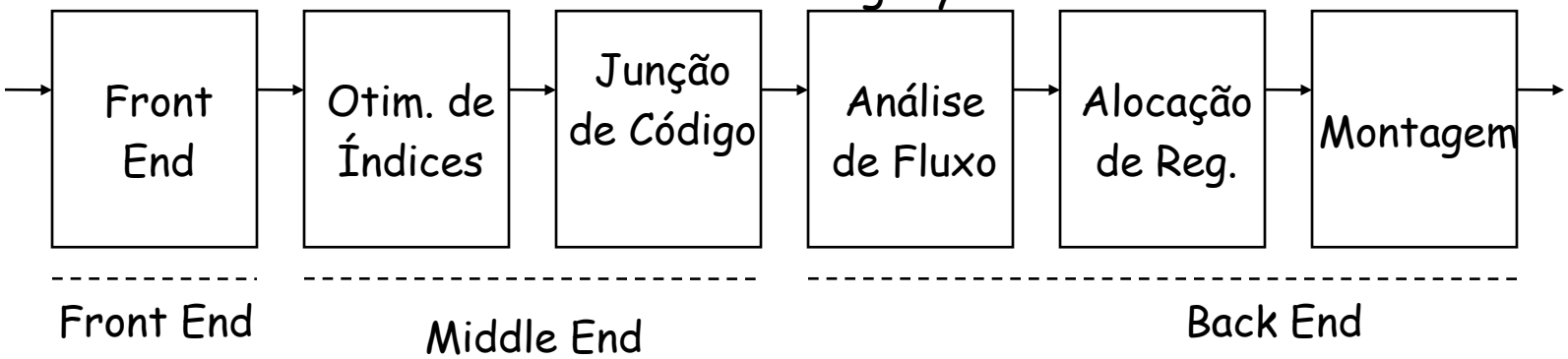

Política escolhe entre interpretador e compilador

# Ambiente de execução

- Serviços de gerenciamento de memória
  - Alocação (no "heap" ou em um registro de ativação na pilha)
  - Desalocação
  - Coleta de lixo
- Checagem de tipo em tempo de execução
- Processamento de erros (ex. exceções)
- Interface com sistema operacional
  - Entrada e saída
- Suporte a paralelismo
  - Inicialização de threads
  - Comunicação e sincronização
- Instrospecção Reflexão (Invoke de Java, RTTI de C++)

# Compiladores Clássicos

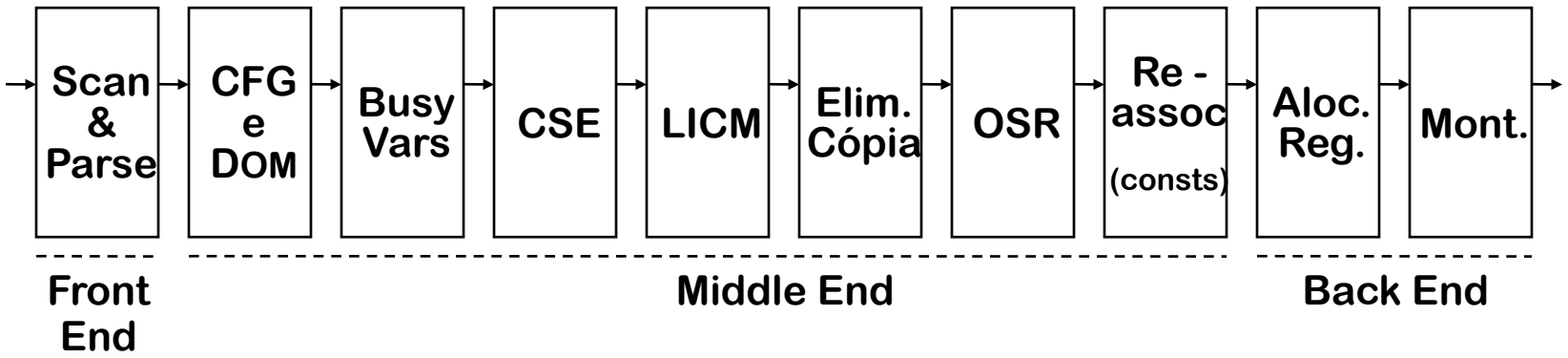
1957: The FORTRAN Automatic Coding System




- Seis passadas numa ordem fixa
- Gerava bom código
  - Assumia um número ilimitado de registradores pra índices
  - Extraía código pra fora de loops, incluindo ifs e gotos
  - Fazia análise de fluxo e alocação de registradores

# Compiladores Clássicos

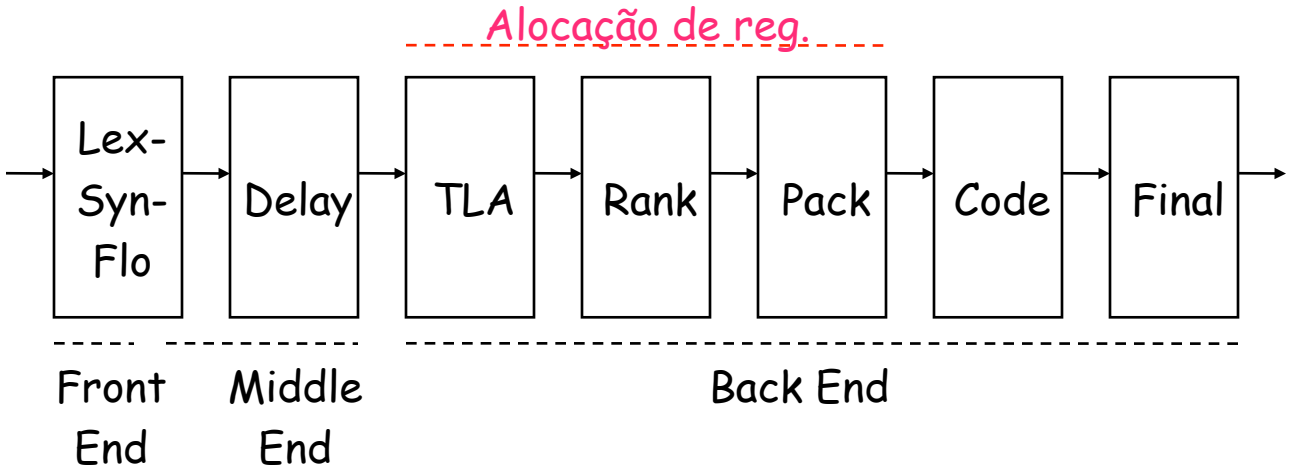
## 1969: Compilador FORTRAN H




- Usava IR de baixo nível, identificava loops usando dominadores
- Foco em otimização de loops
- Front end simples, back end simples para IBM 370

# Compiladores Clássicos

1975: BLISS-11 (Wulf et al., CMU)



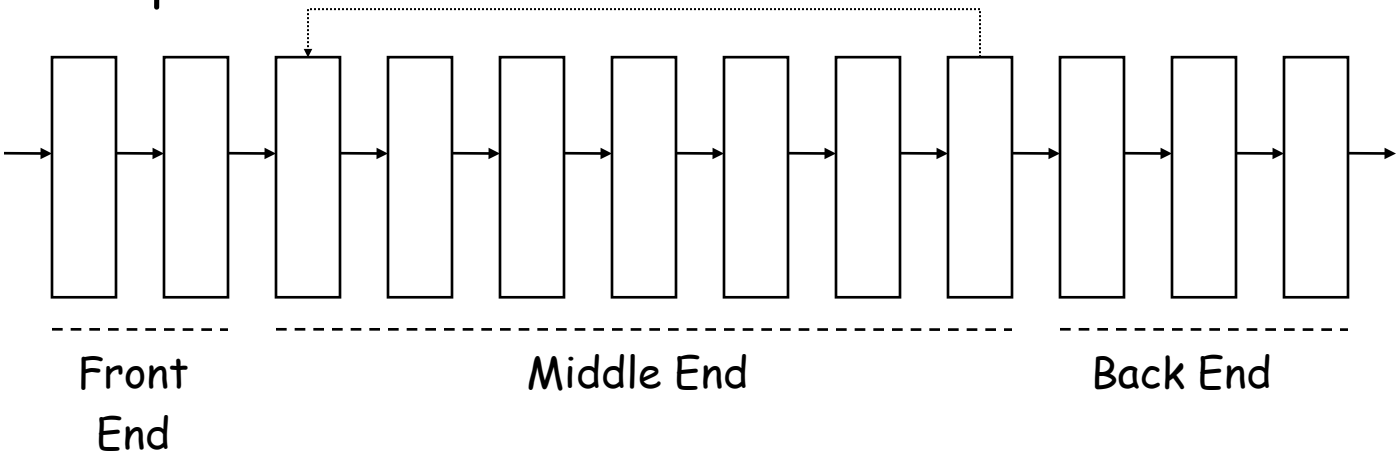

- Compilador para o PDP-11
- Sete passadas em uma ordem fixa
- Foco em formato do código e seleção de instruções

  LexSynFlo fazia uma análise preliminar de fluxo

  Final incluía várias otimizações "peephole"

# Compiladores Clássicos

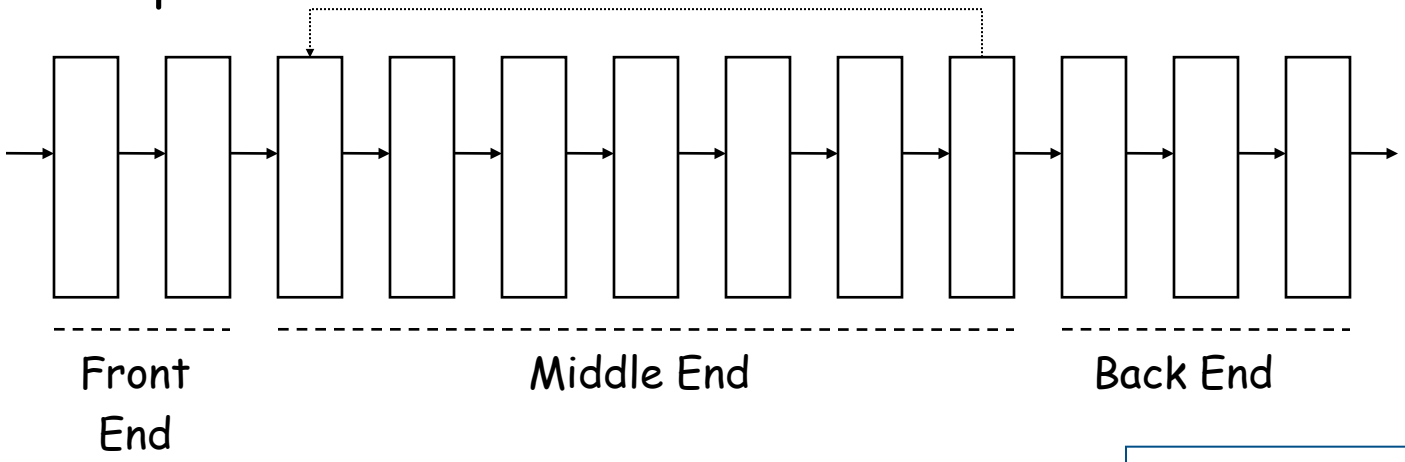
1980: Compilador PL.8 da IBM




- Um front end, vários back ends
- Coleção de 10 ou mais passadas
  - Repetia algumas passadas e análises
  - Representava ops complexas em 2 níveis
  - IR abaixo do nível da máquina

# Compiladores Clássicos

## 1980: Compilador PL.8 da IBM

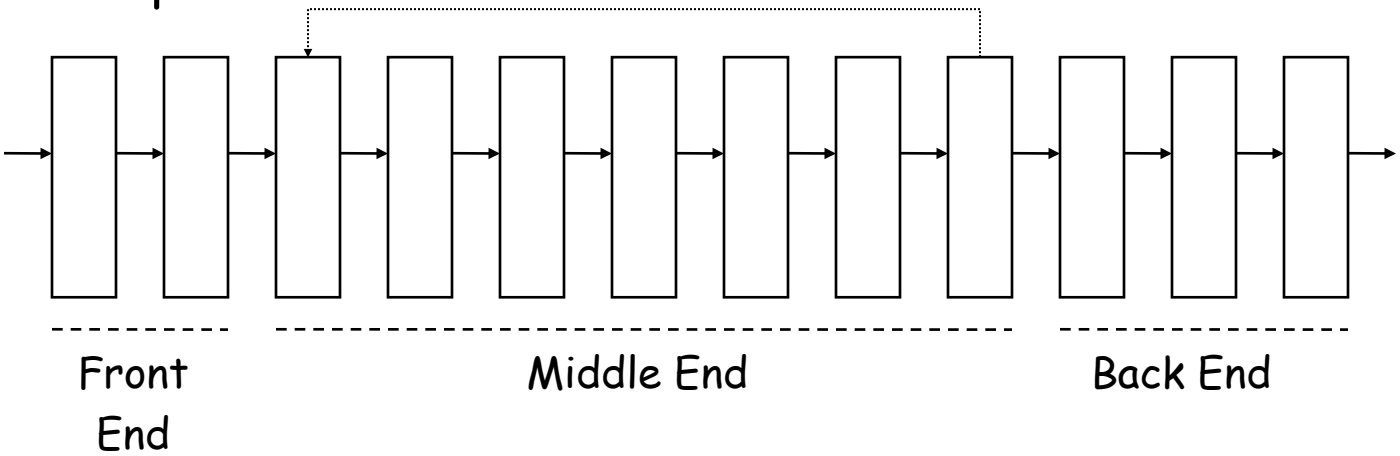



- Um front end, vários back ends
- Coleção de 10 ou mais passadas
  - Repetia algumas passadas e análises
  - Representava ops complexas em 2 níveis
  - IR abaixo do nível da máquina

*Dead code elimination*
*Global cse*
*Code motion*
*Constant folding*
*Strength reduction*
*Value numbering*
*Dead store elimination*
*Code straightening*
*Trap elimination*
*Algebraic reassociation*

# Compiladores Clássicos

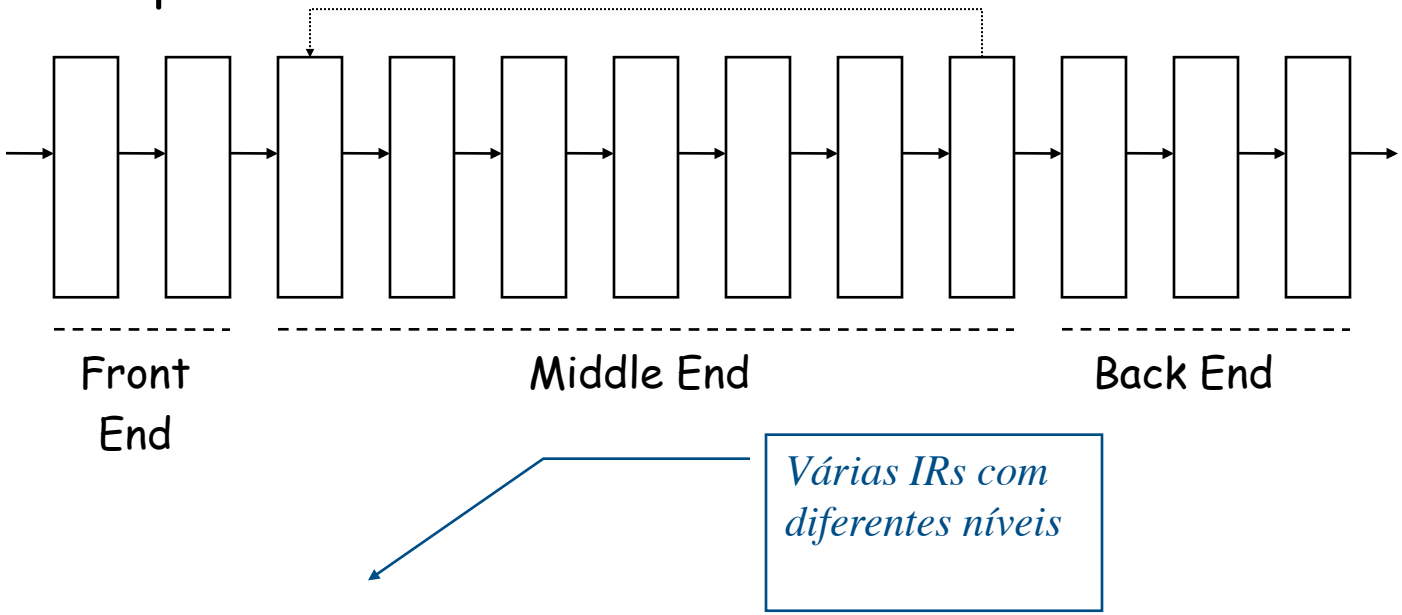
1980: Compilador PL.8 da IBM




- Um front end, vários back ends
- Coleção de 10 ou mais passadas
  - Repetia algumas passadas e análises
  - Representava ops complexas em 2 níveis
  - IR abaixo do nível da máquina

# Compiladores Clássicos

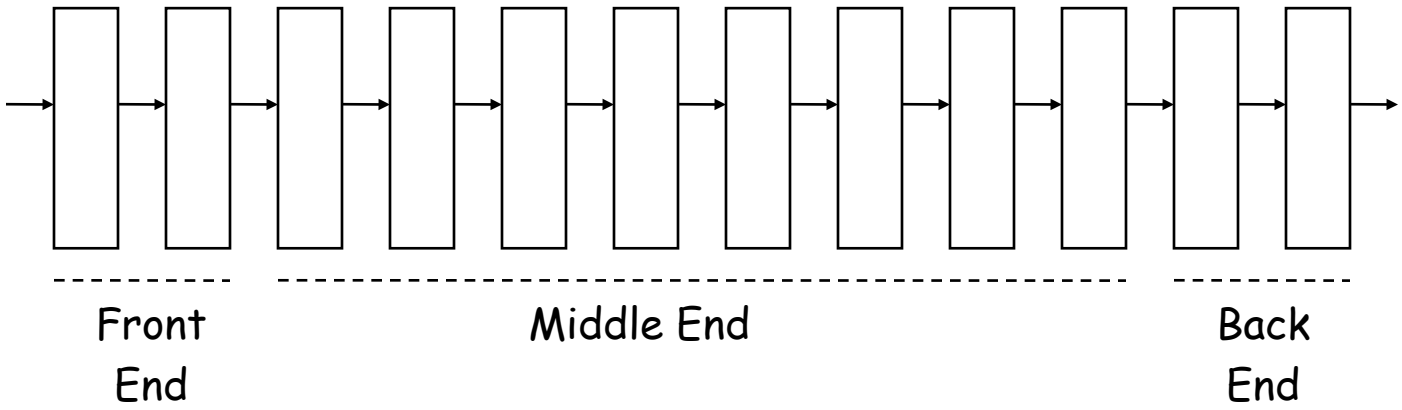
## 1980: Compilador PL.8 da IBM




- Um front end, vários back ends
- Coleção de 10 ou mais passadas
  - Repetia algumas passadas e análises
  - Representava ops complexas em 2 níveis
  - IR abaixo do nível da máquina

# Compiladores Clássicos

1986: Compilador PA-RISC HP




- Vários front ends, um optimizador, e um back end
- Quatro possíveis níveis de otimização, envolvendo 9 passadas
- Alocador por coloração de grafos, escalonador de instruções, otimizador "peephole"