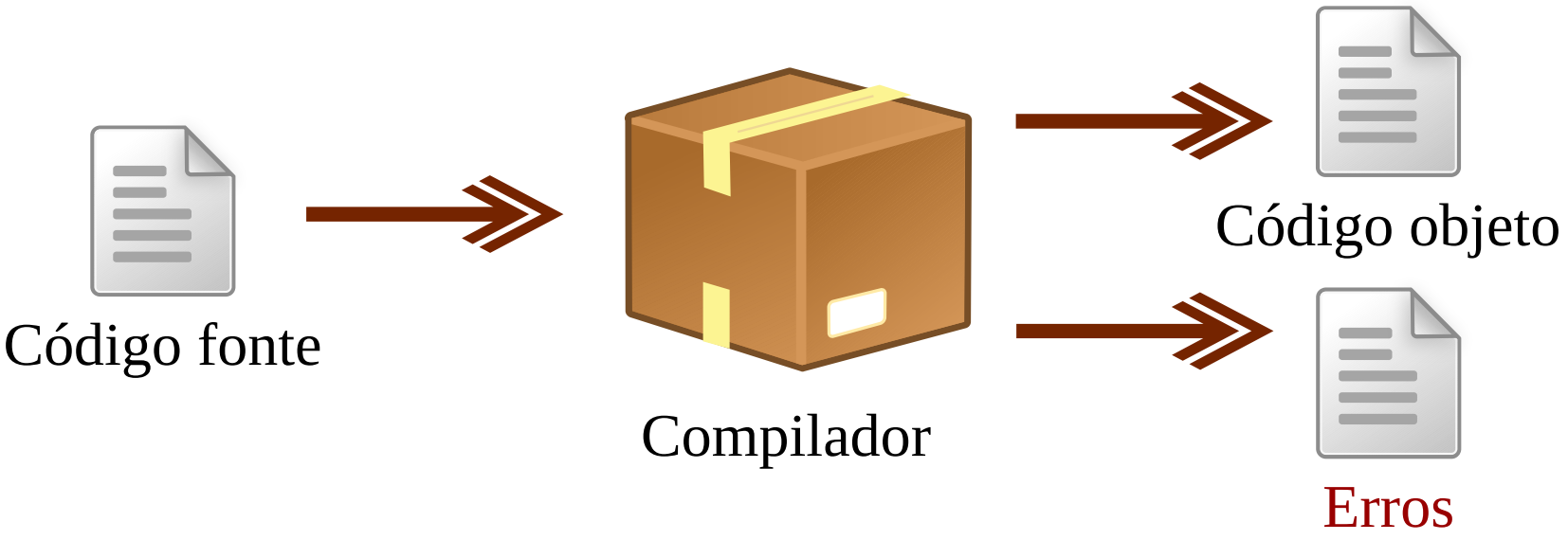
Universidade Federal de Uberlândia
Faculdade de Computação

Estrutura de um Compilador

Curso de Bacharelado em Ciência da Computação
GBC071 - Construção de Compiladores
Prof. Luiz Gustavo Almeida Martins

Visão geral do compilador

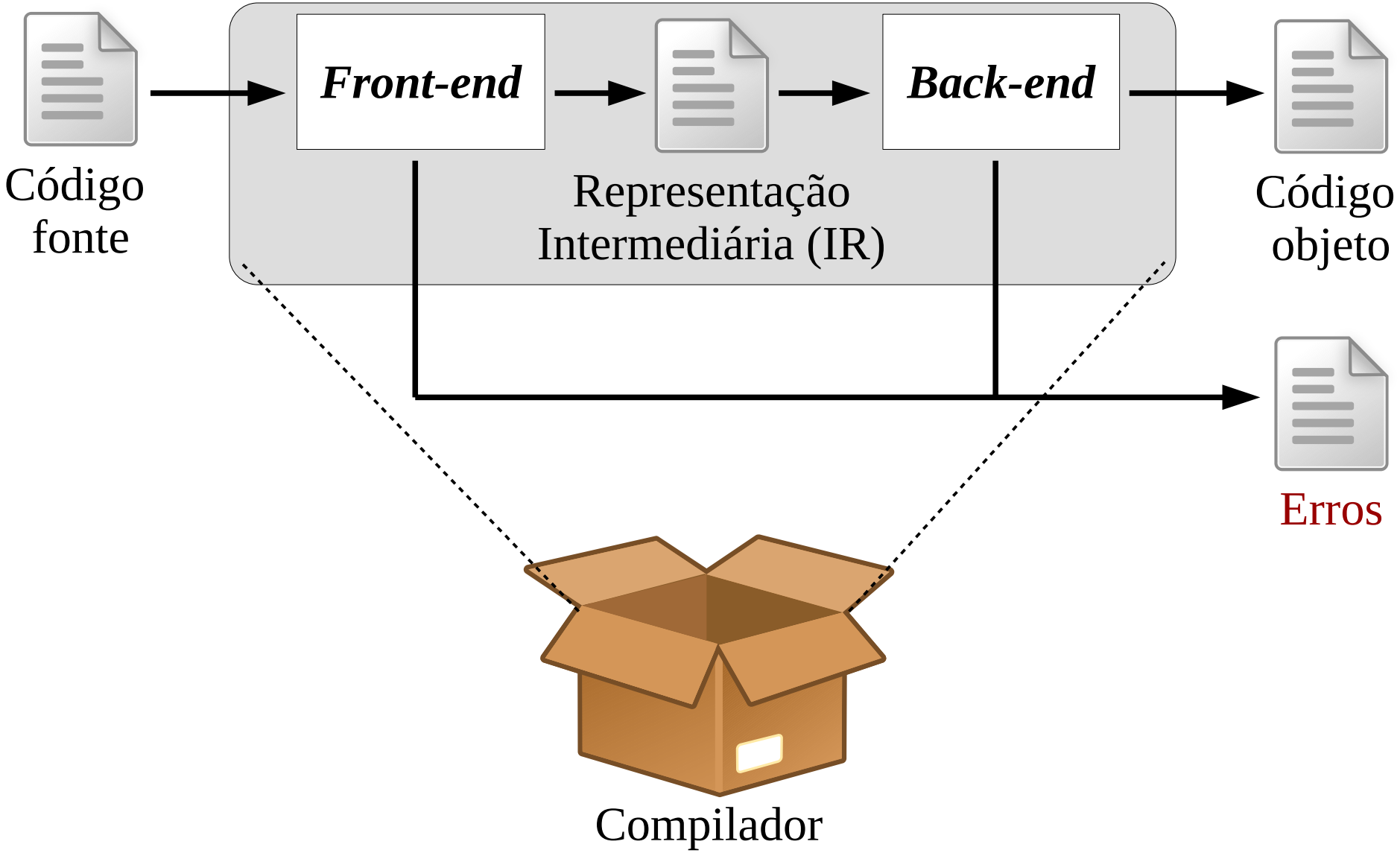
- Um compilador deve:
  - **Reconhecer programas** válidos e inválidos
  - **Gerar código CORRETO**
  - **Gerenciar o armazenamento** de todas as variáveis e do código
  - Gerar um formato do **código objeto compatível** com o S.O. e o ligador/carregador
  - Tratar ou, pelo menos, **reportar erros** encontrados



Estrutura básica do compilador

- Um compilador possui 2 partes essenciais:
  - ***Front-end*** é responsável pela etapa de **análise do código fonte**
    - Mapeia um código fonte pertencente a linguagem em uma IR
  - ***Back-end*** é responsável pela etapa de **síntese do codigo alvo**
    - Mapeia a IR em um código escrito na linguagem de máquina da arquitetura alvo
  - Usa uma **Representação Intermediária (IR)** para a **comunicação entre as partes**
    - Independente de linguagem e arquitetura

Estrutura básica do compilador



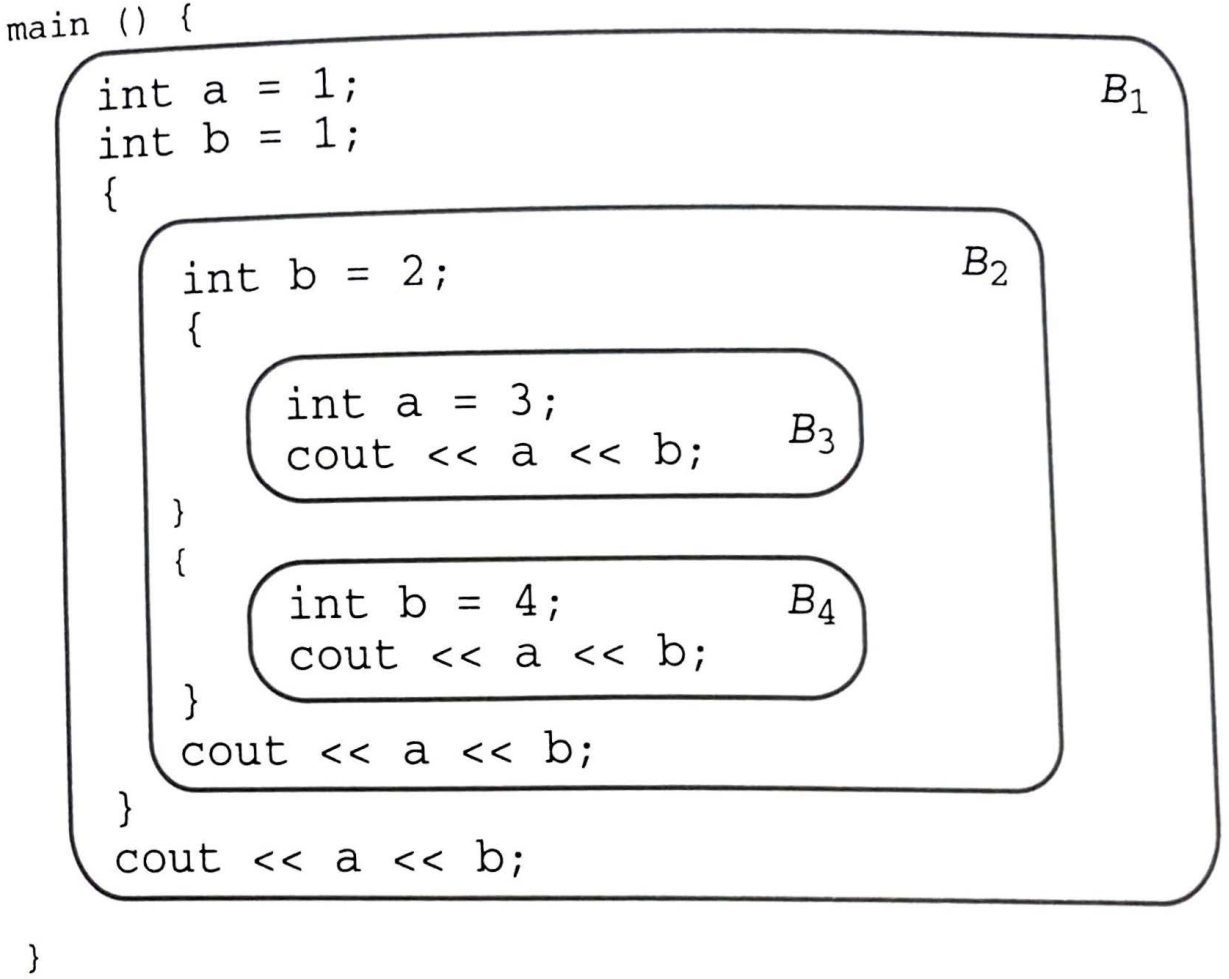
Etapa de análise (*front-end*)

- Determina se a estrutura do código fonte é válida sintática (**forma**) e semanticamente (**significado**)
  - **Sintaxe:** conjunto de regras que determinam a estrutura da linguagem (**construções corretas**)
  - **Semântica:** define como as construções devem ser **interpretadas e/ou executadas**

- **Ex:** *x = y;* /* linguagem C */
  - **Sintaxe:** estrutura correta de uma atribuição
  - **Semântica:** substituir o valor de *x* pelo valor de *y*
    - Definir qual valor usar pode não ser uma tarefa trivial (**análise de escopo**)

Etapa de análise (*front-end*)

- Exemplo de **análise de escopo**:



Declaração	Escopo
int a = 1;	B_1 - B_3
int b = 1;	B_1 - B_2
int b = 2;	B_2 - B_4
int a = 3;	B_3
int b = 4;	B_4

Fonte: (Aho, 2008)

Etapa de análise (*front-end*)

- Quebra o código fonte em pedaços (***tokens***)
- Verifica se o código atende a estrutura gramatical da linguagem
  - Reconhece programas aceitos (ou não) pela linguagem
- Verifica a consistência semântica do programa com as definições da linguagem
  - **Ex:** verificação de tipos, compatibilidade de operandos, coerção, etc.
- Gera uma representação intermediária (IR) do código
  - IR é moldada de acordo com as necessidades do *back-end*
- Coleta informações sobre o código e as armazena em uma estrutura de dados chamada **tabela de símbolos**
  - Usada em todas as fases do compilador
- Reporta erros de uma forma útil
- Boa parte da construção do *front-end* pode ser automatizada

Etapa de análise (*front-end*)

- **Entrada:** código fonte

- **Etapas:**
  - Analisador léxico
  - Analisador sintático
  - Analisador semântico
  - Gerador de código intermediário

- **Saída:**
  - Representação intermediária (IR)
  - Tabela de símbolos
  - Mensagens de erro referentes aos problemas no código

Etapa de síntese (*back-end*)

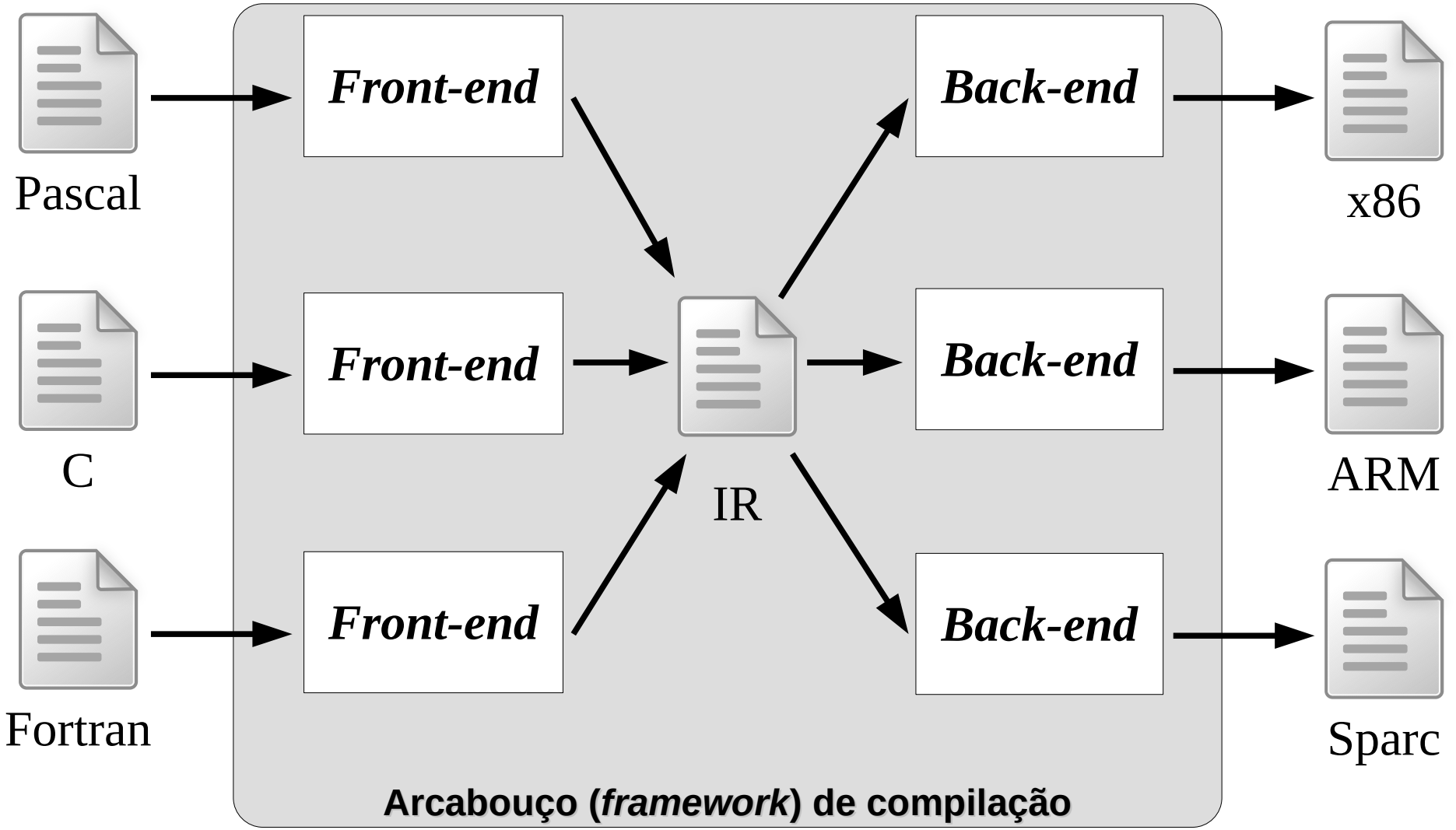
- Gera o código objeto considerando os recursos disponíveis na arquitetura alvo
  - **Seleciona as instruções** para implementar cada operação do código intermediário (IR)
  - Define quais valores manter nos registradores (**alocação**)
  - Define a **distribuição das instruções** entre os recursos disponíveis para o processamento (**escalonamento**)
- Garante a conformidade com as interfaces do sistema
- Automação mais complexa e difícil (**- sucesso**)

Etapa de síntese (*back-end*)

- **Entrada:**
  - Representação intermediária (IR)
  - Tabela de símbolos
- **Etapas:**
  - Geração de código
  - Otimização dependente da máquina (local)
- **Saída:**
  - Código objeto para a máquina alvo
  - Mensagens de erro

Compiladores multi-plataforma

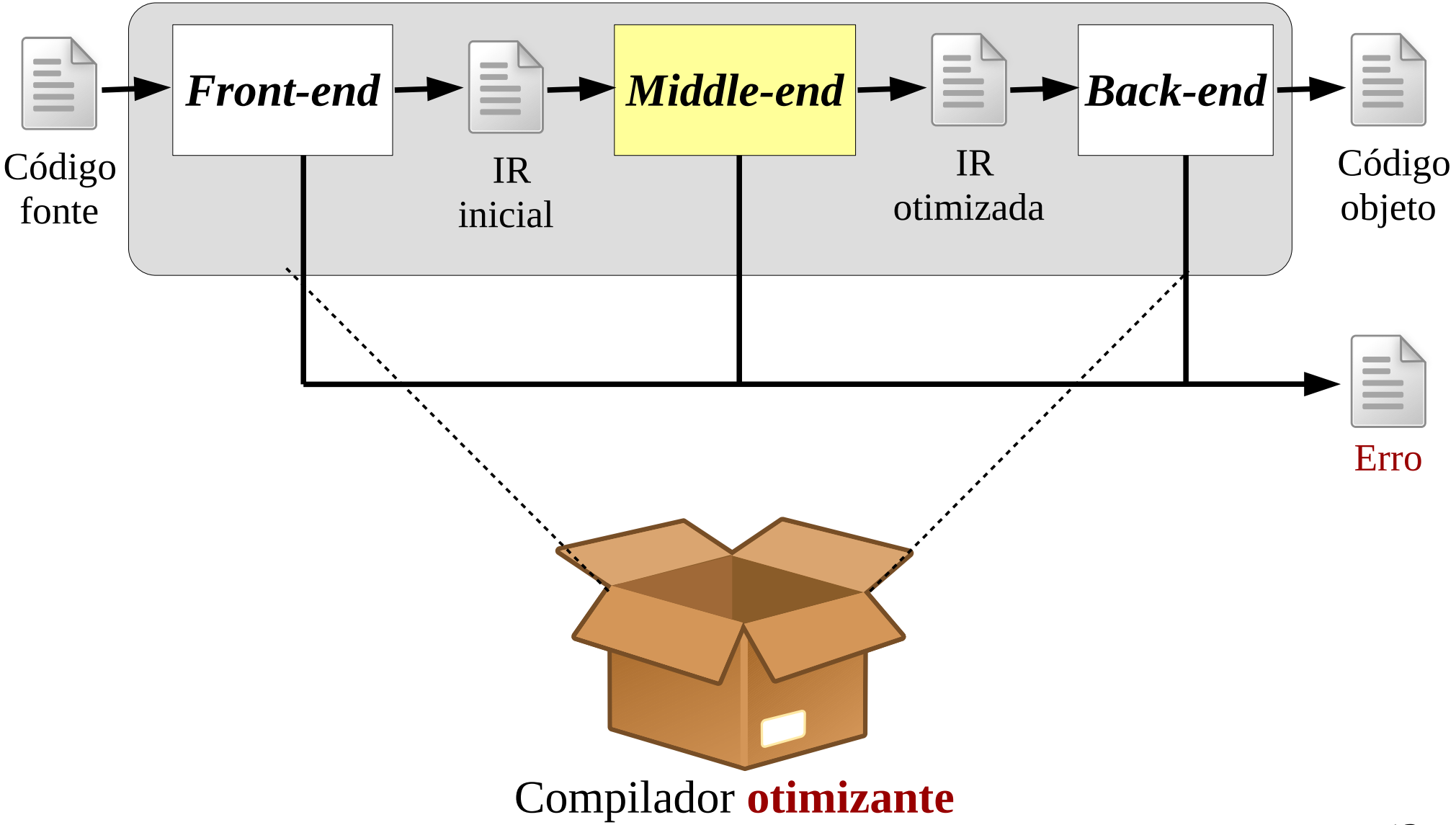
- Ambientes atuais admitem **diferentes *front-ends* e *back-ends***



Compiladores multi-plataforma

- Dificuldades dos projetos multi-plataforma:
  - Todo conhecimento específico da linguagem deve estar no *front-end*
  - Deve codificar todas as características em uma única representação intermediária (IR)
  - Todo conhecimento específico da arquitetura alvo deve estar no *back-end*
- Sucesso limitado em ambientes com IRs de baixo-nível (ex: LLVM)
- Área de pesquisa ativa (ex: *Graal* e *Truffle*)

Estrutura básica do compilador



Etapa de otimização (*middle-end*)

- **Transforma sintaticamente a IR** de modo que o *back-end* possa gerar um código objeto melhor
  - Não garante uma solução ótima (**indecidível**)
  - **Otimização de alto nível:** independente da máquina (**global**)

- **Aplica passos de otimização** sobre o código intermediário
  - Número de passos de otimizações varia muito entre os compiladores
    - Quanto mais otimizações, mais tempo é gasto nessa fase
  - Cada passo pode envolver uma série de análises e transformações
  - Aplicação depende de condições específicas no código

- Passos de otimização **interagem entre si**
  - Define oportunidades para outras otimizações serem aplicadas
  - Sequências distintas → diferentes códigos e desempenho

Etapa de otimização (*middle-end*)

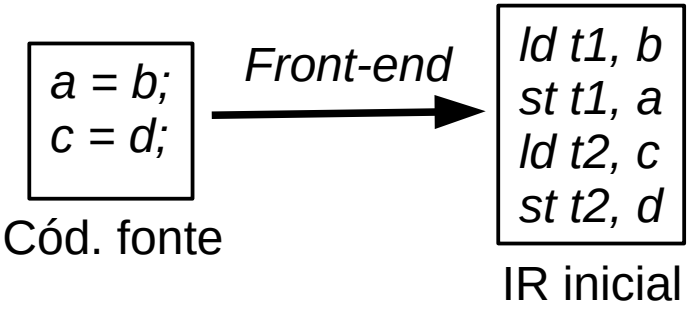
- **Entrada:** IR inicial

- **Etapas:**
  - Análise da **IR corrente**
    - Gera conhecimento contextual do código
  - Aplicação das transformações
    - Forma monolítica ou estruturada

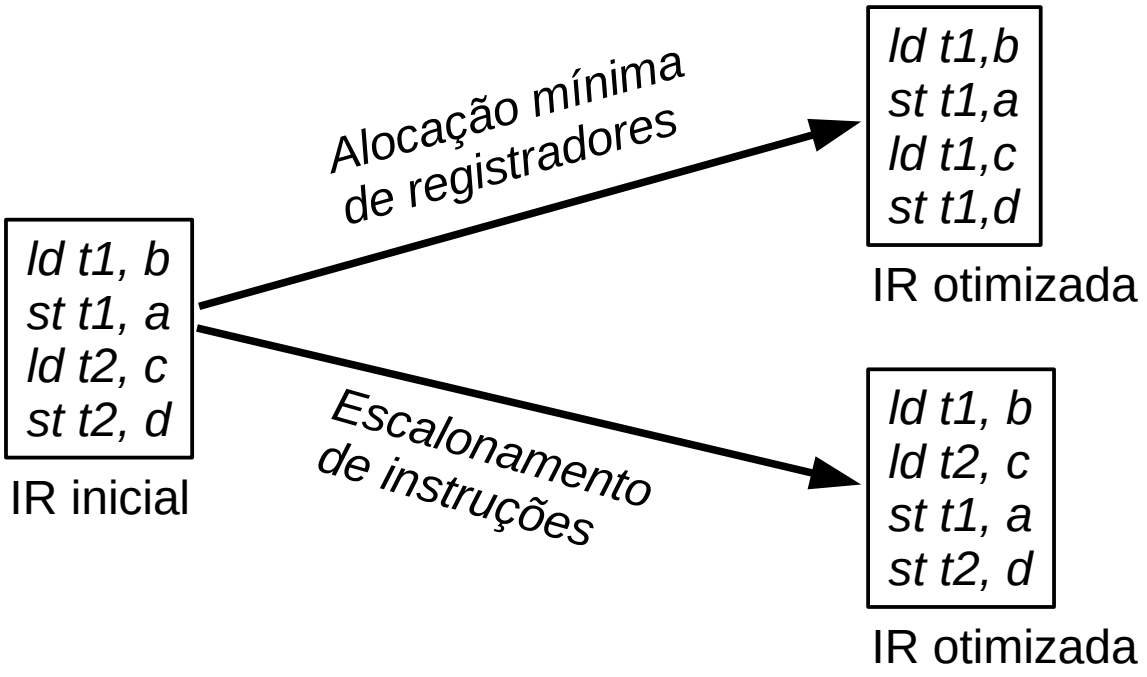
- **Saída:** IR otimizada

Etapa de otimização (*middle-end*)

- **Ex:** considere a síntese



Durante a otimização pode ocorrer:



Propicia otimizar o uso dos registradores

Não permite o escalonamento de instruções

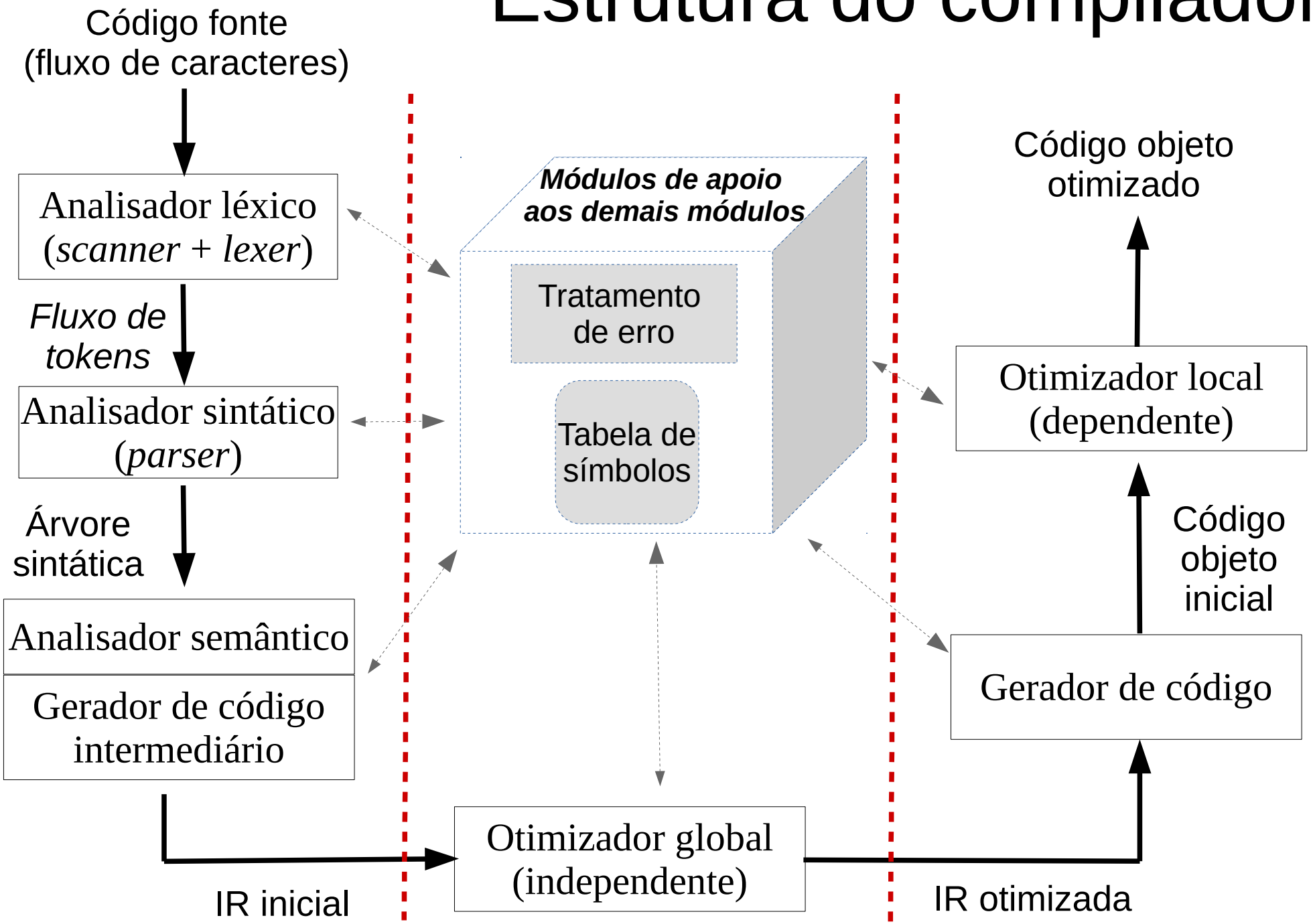
Propicia paralelismo, podendo reduzir o tempo de execução

Não permite alocação mínima de registradores

Projeto do compilador

- A compilação eficiente é uma **tarefa complexa**:
  - **Abstração da instruções de alto nível**
    - Não têm relação direta com as instruções de baixo nível
  - Necessidade de técnicas capazes de **melhorar a qualidade do código** gerado na compilação
    - Análise do código fonte e sua tradução em um versão mais eficiente sem afetar sua funcionalidade original
  - Complexidade aumenta com as linguagens mais recentes e com as novas arquiteturas de computadores
    - Custo do ***front-end*** é da ordem de ***O(n)*** ou ***O(n log n)***
    - ***Middle-end*** e ***Back-end*** pertencem a classe **NP-completo**

Estrutura do compilador



Análise léxica

- Transforma uma sequência de caracteres que representa o programa em uma **sequência de símbolos básicos** da linguagem (***tokens***)
  - Reconhecer os itens léxicos (*tokens*) do programa
    - Extração e classificação de **sequências elementares completas** (átomos ou **lexemas**)
    - Eliminação de **delimitadores e comentários**
    - Identificação de **palavras reservadas**
  - Atribuir identificador a cada item léxico
  - Relato e/ou recuperação de **erros léxicos**
    - **Ex:** fim inesperado de arquivo, mal formação de lexemas, etc.

- *Analisador léxico pode ser gerado de forma automática a partir de uma especificação léxica definida por* ***expressões regulares***

Análise léxica

- Lê o fluxo de caracteres do código fonte (***scanning***) e os agrupa em palavras (**lexemas**)
  - Leitura é feita da esquerda (início) para a direita (fim)

- Cada lexema retorna um ***token*** no formato:

<nm_token, vlr_atributo>

  - **nm_token:** símbolo abstrato de identificação do lexema
  - **vlr_atributo:** dados do lexema
    - **Ex:** posição na tabela de símbolos

Análise léxica

- **Exemplo 1 -** considere o trecho de programa:

if (resp != esp) x = 0;

  - **Lexemas:** *if , (, resp , != , esp ,) , x , = , 0 , ; , EOF*

  - ***Tokens:*** *<**if**>, <**(**>, <**ID**,1>, <**!=**>, <**ID**,2>, <**)**> , <**ID**,3> , <**=**> ,<**cnt_int**,0> ,<**;**> ,<**EOF**>*

    OBS: Os identificadores *resp*, *esp* e *x* ocupam as posições 1, 2 e 3 na tabela de símbolos, respectivamente

Análise léxica

- **Exemplo 2 -** considere o trecho de programa:

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

Código

Fonte: (Aluisio, 2011)

```
i  n  t sp  g  c  d  (  i  n  t sp  a  , sp  i
n  t sp  b  ) nl  { nl sp sp  w  h  i  l  e sp
(  a sp  !  = sp  b  ) sp  { nl sp sp sp sp  i
f sp  (  a sp  > sp  b  ) sp  a sp  -  = sp  b
; nl sp sp sp sp  e  l  s  e sp  b sp  -  = sp
a  ; nl sp sp  } nl sp sp  r  e  t  u  r  n sp
a  ; nl  } nl
```

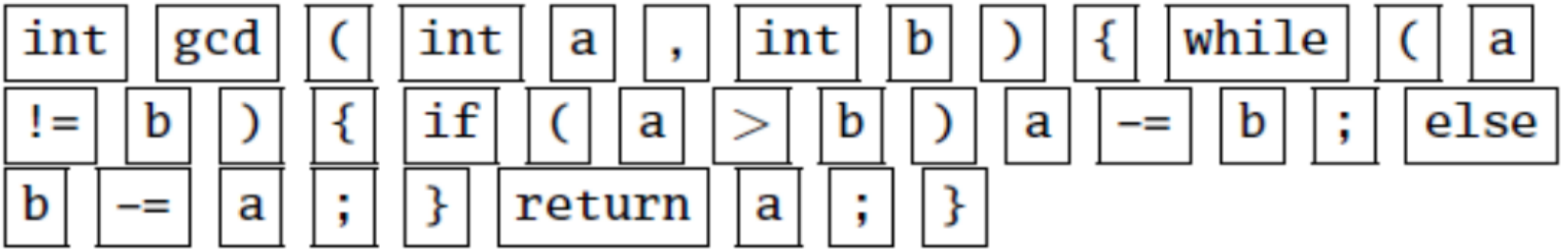
Fluxo de caracteres

Análise léxica

- **Exemplo 2 (continuação):**

Fonte: (Aluisio, 2011)

Lexemas:



***Tokens*:**

<int>, <ID,1>, <(>, <int>, <ID,2>, <,>, <int>, <ID,3>, <)>,

<{>, <while>, <(>, <ID,2>, <!=>, <ID,3>, <)>,

<{>, <if>, <(>, <ID,2>, <!=>, <ID,3>, <)>,

<ID,2>, <-=>, <ID,3>, <;>, <else>, <ID,3>, <-=>, <ID,2>, <;>, <}>,

<return>, <ID,2>, <;>, <}>

Análise sintática

- Agrupa os *tokens* em estruturas sintáticas da linguagem
  - Validar a **estrutura gramatical** da sequência de *tokens*
    - Estrutura da linguagem é especificada por uma **gramática livre de contexto**
  - Construir uma **IR tipo árvore** que mostra a estrutura gramatical
    - **Árvore Sintática Abstrata (AST)**
  - Reportar erros sintáticos
    - **Ex:** “; esperado”, “parêntese desbalanceado“, etc.

- Também é possível criar um **gerador automático** a partir de uma especificação da sintaxe da linguagem

- Resulta em uma árvore que representa a estrutura do programa:
  - **Concreta:** codifica toda a estrutura sintática do programa
  - **Abstrata:** codifica apenas as partes essenciais

Gramática livre de contexto (CFG)

- **Exemplo:** Fonte: (Dubach, 2018)

1	$goal \rightarrow expr$
2	$expr \rightarrow expr\ op\ term$
3	$\quad\quad \mid term$
4	$term \rightarrow number$
5	$\quad\quad \mid id$
6	$op \rightarrow +$
7	$\quad\quad \mid -$

$$S = goal$$
$$T = \{number, id, +, -\}$$
$$N = \{goal, expr, term, op\}$$
$$P = \{1, 2, 3, 4, 5, 6, 7\}$$

Produção	Derivação
	goal
1	expr
2	expr op term
5	expr op y
7	expr - y
2	expr op term - y
4	expr op 2 - y
6	expr + 2 - y
3	term + 2 - y
5	x + 2 - y

Gramática livre de contexto (CFG)

- **Exemplo:** Fonte: (Dubach, 2018)

1	$goal$	$\rightarrow$	$expr$
2	$expr$	$\rightarrow$	$expr\ op\ term$
3		$\mid$	$term$
4	$term$	$\rightarrow$	$number$
5		$\mid$	id
6	op	$\rightarrow$	$+$
7		$\mid$	$-$

$$
\begin{aligned}
S &= goal \\
T &= \{number, id, +, -\} \\
N &= \{goal, expr, term, op\} \\
P &= \{1, 2, 3, 4, 5, 6, 7\}
\end{aligned}
$$

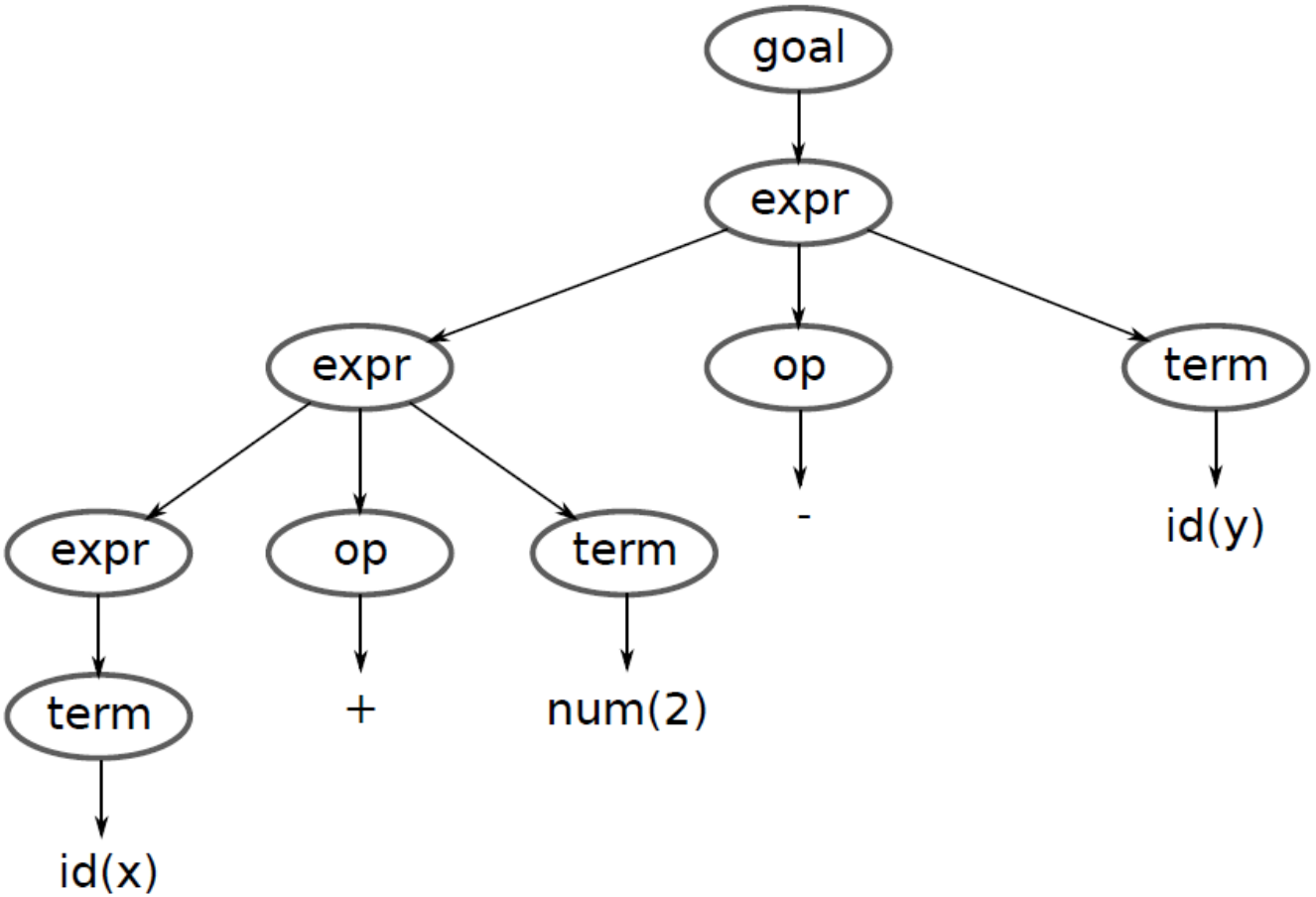
Produção	Derivação
	goal
1	expr
2	expr op term
5	expr op y
7	expr - y
2	expr op term - y
4	expr op 2 - y
6	expr + 2 - y
3	term + 2 - y
5	x + 2 - y

Reconhecer uma sentença válida envolve construir uma árvore sintática a partir da CFG

Árvore de derivação

- **Exemplo:**

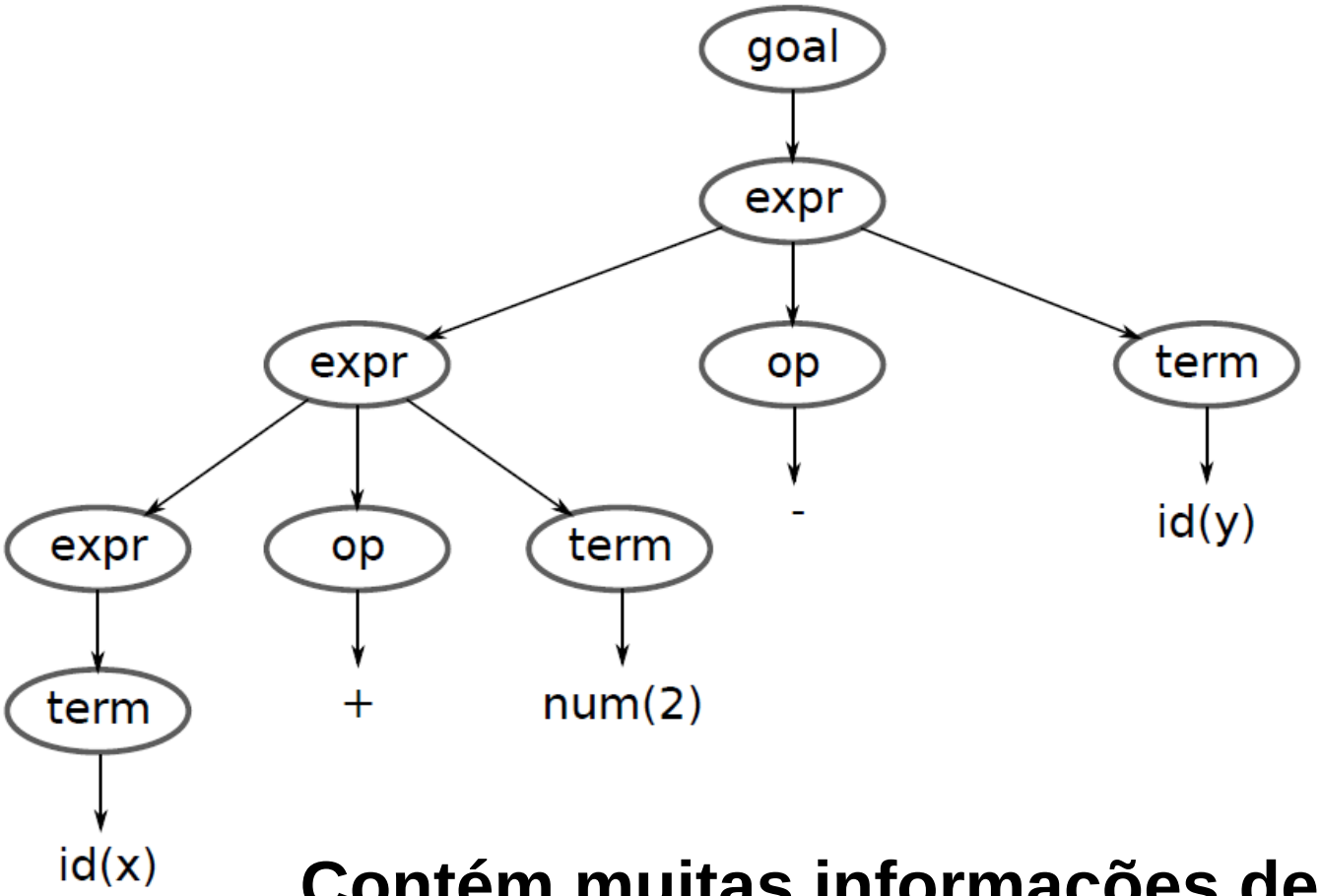
Fonte: (Dubach, 2018)



Árvore de derivação

- **Exemplo:**

Fonte: (Dubach, 2018)



Contém muitas informações desnecessárias

Árvore sintática abstrata (AST)

- **Exemplo:**

Fonte: (Dubach, 2018)

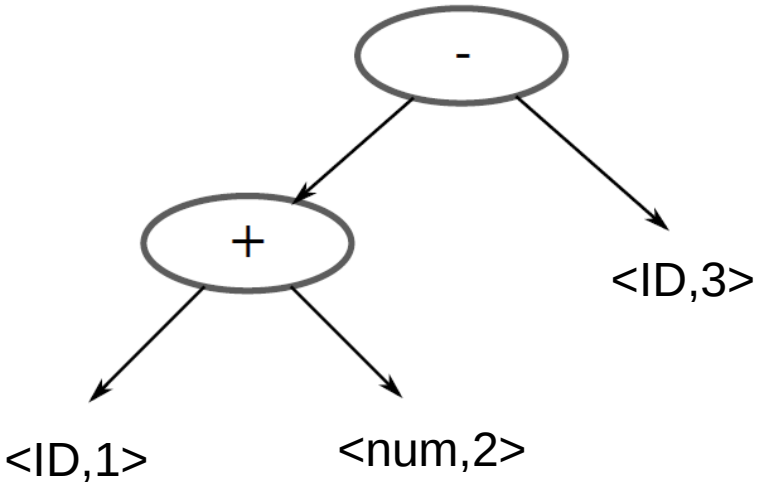


Tabela de símbolos

	Token	Atributo
1	ID	x
2	num	2
3	ID	y
...	...	...

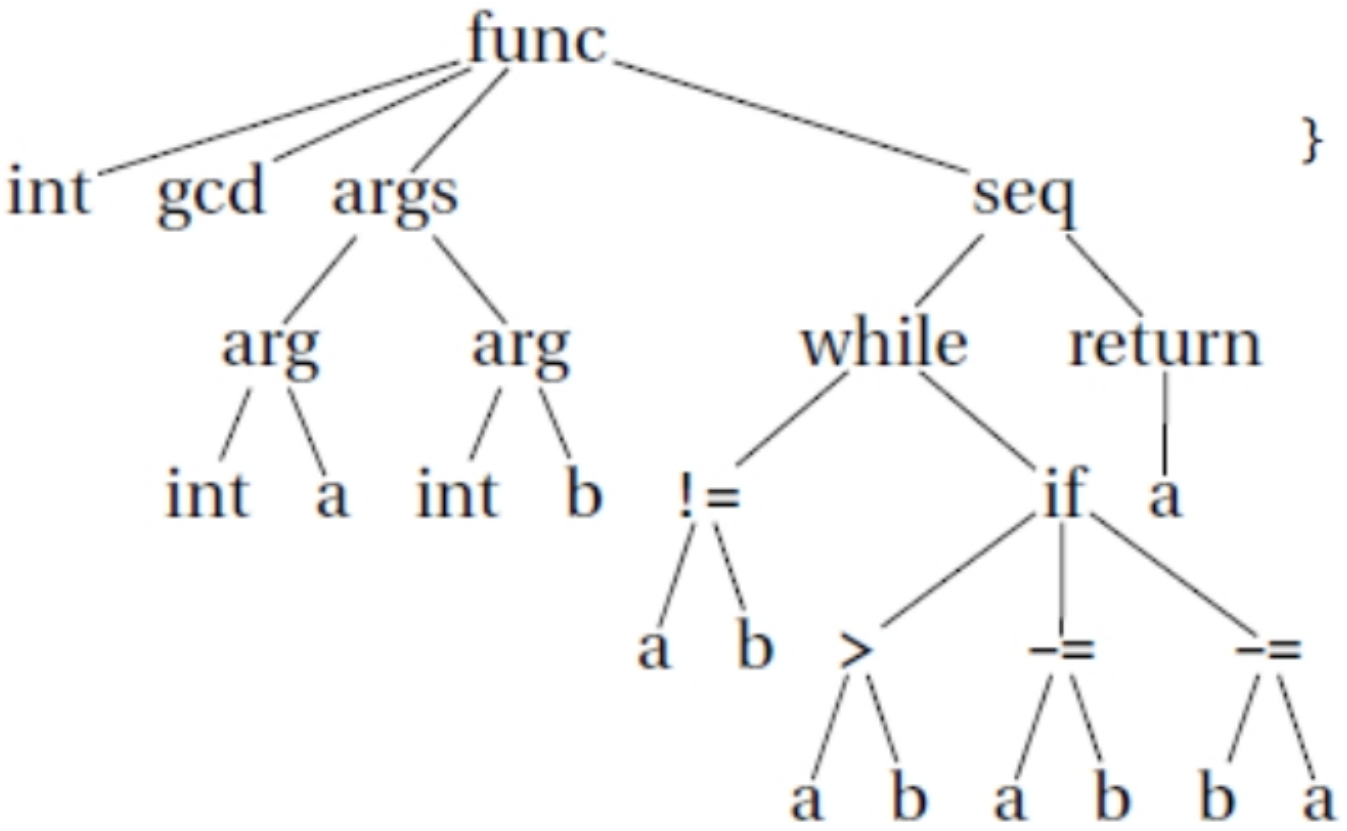
- AST sumariza a estrutura gramatical (**+ concisa**)
  - Não inclui informações da derivação
- Estrutura mais utilizada pelos compiladores
- AST é uma forma de representação intermediária (IR)

Árvore sintática abstrata (AST)

- **Exemplo 2:**

  Fonte: (Aluisio, 2011)

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```



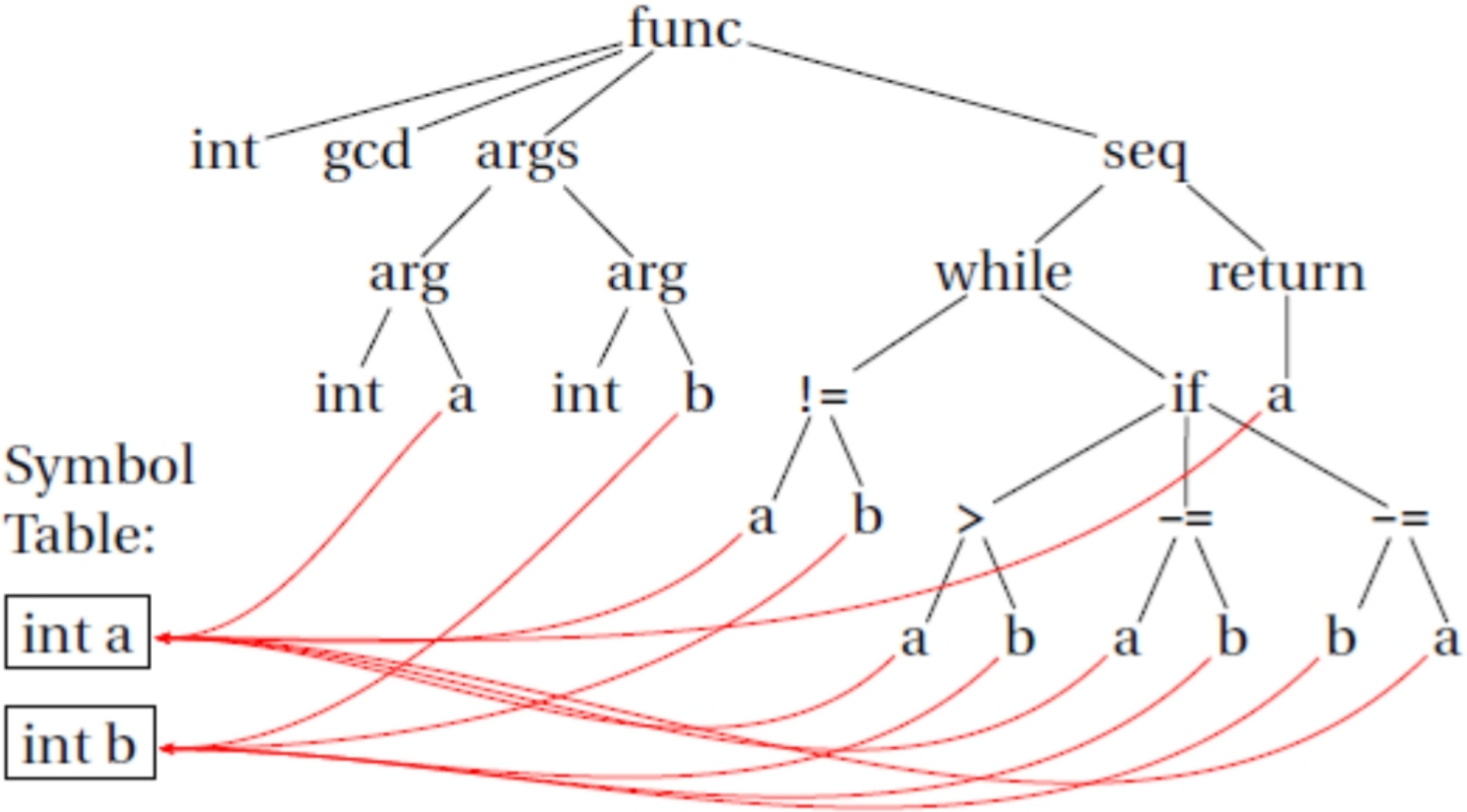
Análise semântica

- Verifica a consistência semântica do programa fonte com a definição da linguagem
  - Usa a AST e as informações da tabela de símbolos
    - Consistência entre declaração e uso de variáveis (**análise de escopo**)
    - Alocação de memória para as variáveis
  - Análise do contexto
    - Algoritmos para gramáticas sensíveis ao contexto não são eficientes
    - **Tabela de símbolos** é usada para tratar aspectos sensíveis ao contexto da sintaxe
  - Verificação de tipo (**parte importante do processo**)
    - Checa se cada operador possui operandos compatíveis
    - Podem ocorrer conversões automática de tipos (**coerções**)
  - Relato de erros de contexto e de tipos
    - **Ex:** variável não declarada, número de parâmetros inconsistente, incompatibilidade de tipos, etc.

Análise semântica

- **Exemplo 1:** verificação de tipo

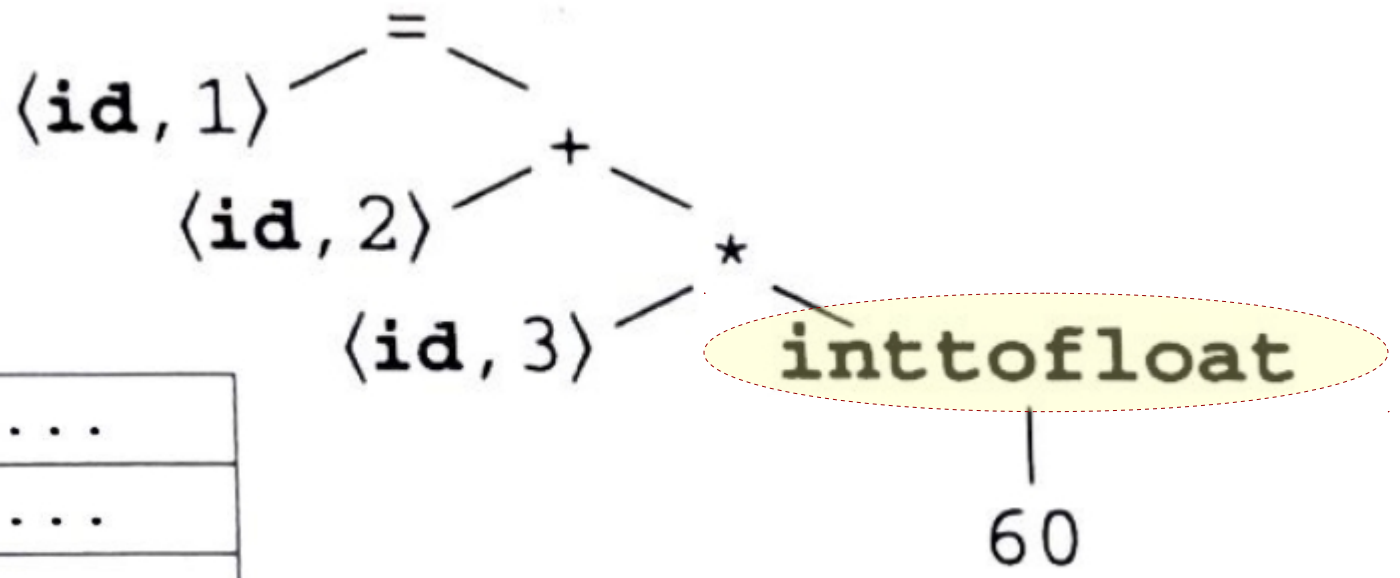
Fonte: (Aluisio, 2011)



Análise semântica

- **Exemplo 2:** coerção

Fonte: (Aho, 2008)



position	...
initial	...
rate	...

TABELA DE SÍMBOLOS

Geração de código intermediário

- Gera uma **representação intermediária (IR)** usada internamente pelo compilador
  - IR explícita de **baixo nível** ou um tipo de linguagem de máquina
  - Independe da linguagem e da máquina (na teoria)
  - Programa para uma **máquina abstrata**

- IR deve ter 2 propriedades importantes (**desempenho**):
  - Ser facilmente produzida
  - Ser facilmente traduzida para a **máquina alvo**

- Essa etapa pode ser realizada em conjunto com a análise semântica

Geração de código intermediário

- **Código de 3 endereços:**
  - É uma das IR mais utilizadas
  - Sequência de instruções do tipo ***assembly* com 3 operandos**:
    - Atribuições possui **no máximo um operador** no lado direito
    - Compilador precisa gerar um **nome temporário** para guardar o resultado das instruções (**registrador**)
    - Algumas instruções possuem **menos de 3 operandos**

Ex: considerando que as variáveis são do tipo *float*, a atribuição ***x = y + z * 10;*** gera o seguinte código de 3 endereços:

t1 = inttofloat(10)

*t2 = id3 * t1*

t3 = id2 + t2

id1 = t3

Fonte: (Aho, 2008)

Geração de código intermediário

- **Exemplo 2:**

Fonte: (Aluisio, 2011)

```
L0: sne   $1,  a, b
    seq   $0, $1, 0
    btrue $0, L1    % while (a != b)
    sl    $3,  b, a
    seq   $2, $3, 0
    btrue $2, L4    % if (a < b)
    sub   a,   a, b % a -= b
    jmp   L5
L4: sub   b,   b, a % b -= a
L5: jmp   L0
L1: ret    a
```

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

Otimização

- Modificação no código para melhorar desempenho
  - Tempo de execução é o mais usual
  - **Outras métricas:** espaço requerido, consumo de energia, etc.

- Otimizador deve atender aos requisitos de projeto:
  - **Preservar a semântica** do programa compilado (**ser correta**)
    - Produz o efeito desejado para todas as entradas possíveis
    - Precisa de **verificação formal**
  - Deve **ser eficiente** (melhorar desempenho de muitos programas)
  - **Tempo de compilação** precisa continuar **razoável**
  - **Esforços de projeto** precisam ser **administráveis**
    - Manter o sistema simples para garantir custos viáveis de engenharia e manutenção

Otimização

- **Otimização global:**
  - Otimização de **alto nível** ou **independente da máquina**
    - Otimização baseada apenas em análises de código (ex: fluxo de dados)
  - Otimiza o **código intermediário** (IR)
  - Desempenho depende da “criatividade” de seu projetista
    - Quais e como são implementados os passos de otimização

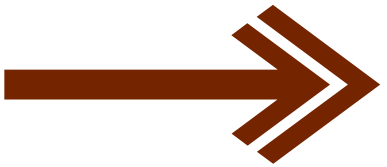
- **Otimização local:**
  - Otimização de **baixo nível** ou **dependente da máquina**
  - Melhorias no **código objeto**
  - Desempenho depende do conhecimento sobre a arquitetura
    - Decisões se baseiam nas restrições impostas pelos recursos disponíveis

Otimização

- **Exemplo:** otimização de alto nível

```
t1 = inttofloat(10)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Código original



```
t1 = id3 * 10.0
id1 = id2 + t1
```

Código otimizado

Fonte: (Aho, 2008)

Geração de código

- Gera o código em linguagem objeto a partir da representação intermediária (IR)

- Mapeia cada nó da AST ou instrução de 3 endereços em instruções na linguagem objeto (**seleção de instruções**)
  - Gerar código de máquina (relocável ou absoluto) ou de montagem (*assembly*)
  - Busca produzir código rápido e compacto
  - Aproveita as características arquiteturais
    - **Ex:** modos de endereçamento disponíveis
  - Pode ser visto como um problema de casamento de padrões (***pattern matching***)
    - Métodos *ad hoc*, *pattern matching*, programação dinâmica

Geração de código

- **Alocação de registradores** é um aspecto crítico dessa etapa
  - Assegura que os **dados estejam disponíveis nos registradores** quando forem usados
  - Gerencia um conjunto limitado de recursos
  - Pode mudar a escolha de instruções e/ou incluir novas instruções de *load* e *store*
  - Alocação ótima é um problema **NP-completo**
    - Problema de **coloração de grafos**
    - Compiladores adotam **soluções aproximadas**

- **Alocação de memória** para os identificadores
  - Decisões sobre alocação de espaço podem ser durante a geração do código intermediário ou do código objeto

Geração de código

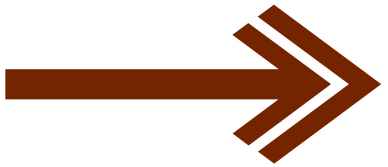
- **Escalonamento de instruções**
  - Busca evitar paradas do *hardware* devido a dependência dos dados entre instruções adjacentes (*stall* e *interlock*)
  - Tentar usar todas as unidades funcionais de forma produtiva
  - Pode aumentar o tempo de vida das variáveis
    - Mudança na alocação
  - Escalonamento ótimo é um problema NP-completo
    - Métodos heurísticos

Geração de código

- **Exemplo:**

```
t1 = id3 * 10.0
id1 = id2 + t1
```

Representação intermediária (código 3 endereços)

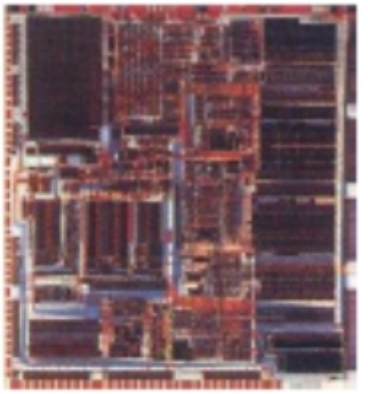


```
LDF   R2, id3
MULF  R2, R2, #10.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

Código objeto (*assembly*)

Fonte: (Aho, 2008)

Geração de código

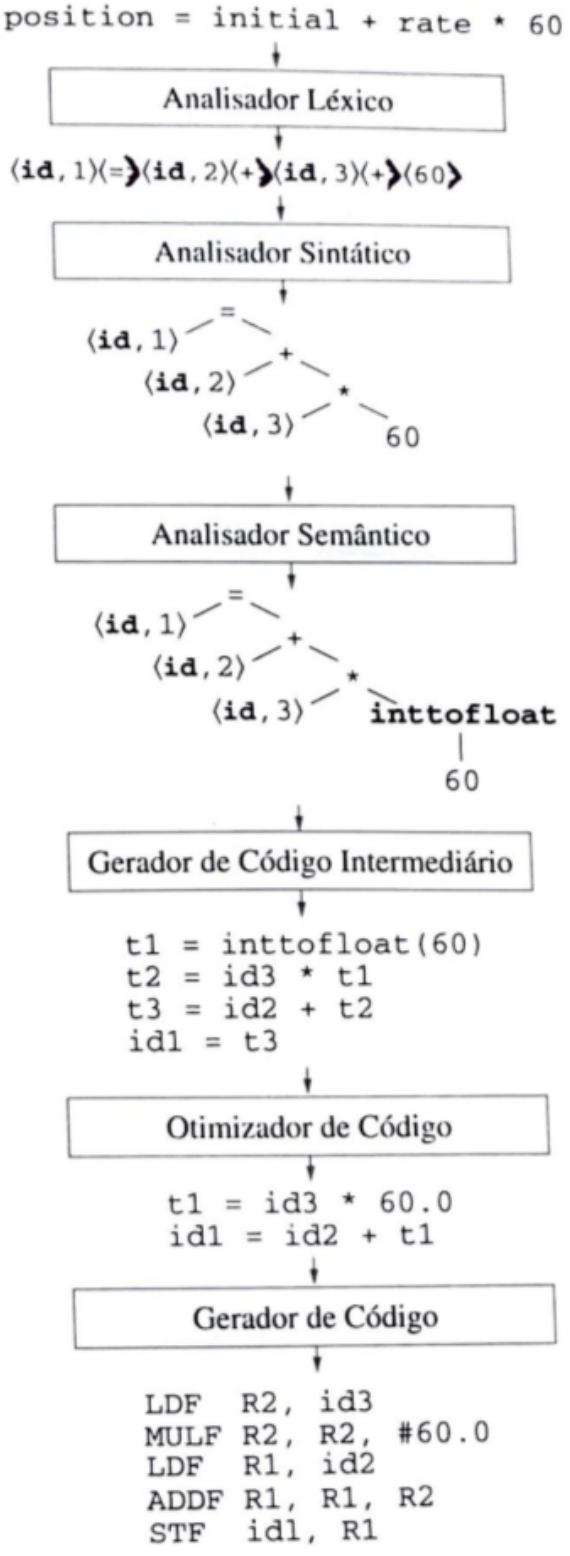


- **Ex 2:** código para assembler 80386

```
gcd:  pushl %ebp              % Save FP
      movl  %esp,%ebp
      movl  8(%ebp),%eax      % Load a from stack
      movl  12(%ebp),%edx     % Load b from stack
.L8:  cmpl  %edx,%eax
      je    .L3               % while (a != b)
      jle   .L5               % if (a < b)
      subl  %edx,%eax         % a -= b
      jmp   .L8
.L5:  subl  %eax,%edx         % b -= a
      jmp   .L8
.L3:  leave                   % Restore SP, BP
      ret
```

Fonte: (Aluisio, 2011)

Visão Geral



position	...
initial	...
rate	...

TABELA DE SÍMBOLOS

Fonte: (Aho, 2008)

Formas de organização

- Fases de um compilador podem ser executadas **separadamente** (em seqüência) **ou combinada em passos**

- **Compilação em etapas (sequencial):**
  - A execução de uma fase termina antes de iniciar a próxima
  - **Vantagem:** possibilidade de otimizações no código
  - **Desvantagem:** aumento do tempo de compilação

- **Compilação em um passo:**
  - Programa-objeto é produzido à medida que o programa-fonte é processado
  - **Vantagem:** eficiência na compilação
  - **Desvantagem:** dificuldade em otimizar o código

- Ambientes de compilação modernos adotam um meio termo
  - Fases agrupadas em *front-end* (análise), *middle-end* (otimização) e *back-end* (síntese)

Referências Bibliográficas

- Aho, A.V.; Lam, M.S.; Sethi, R.; Ullman, J.D. Compiladores: Princípios, técnicas e ferramentas, 2ª edição, Pearson, 2008
- Aluisio, S. material da disciplina “Teoria da Computação e Compiladores”, ICMC/USP, 2011
- Dubach, C. material da disciplina “*Compiling Techniques*”, University of Edinburgh, 2018
- Henriques, M. A. A. material da disciplina “Compiladores”, UNICAMP, 2006