Compiladores
(IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

Síntese

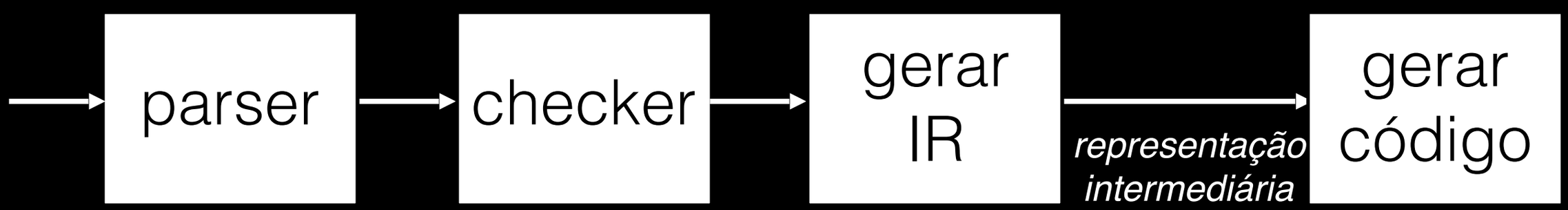
Compiladores costumam ser organizados em uma série de passadas…

Ao derivar conhecimento sobre o código, precisa transmitir esta informação entre passadas

Portanto, o compilador precisa de uma representação dos fatos que deriva a partir de um programa

Representações intermediárias de código

Geração de Código Intermediário



O que representar?

Como representar?

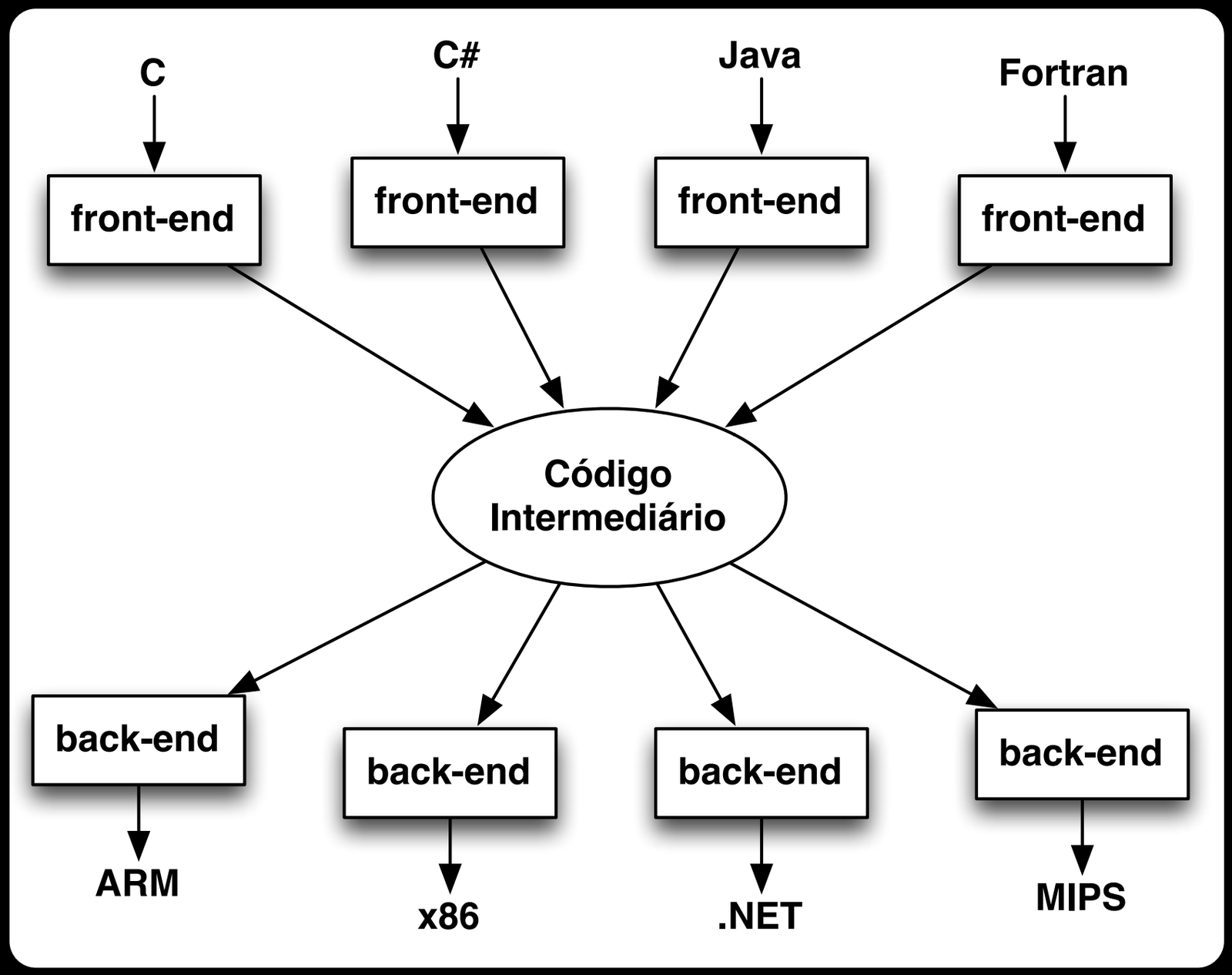
Geração de Código Intermediário

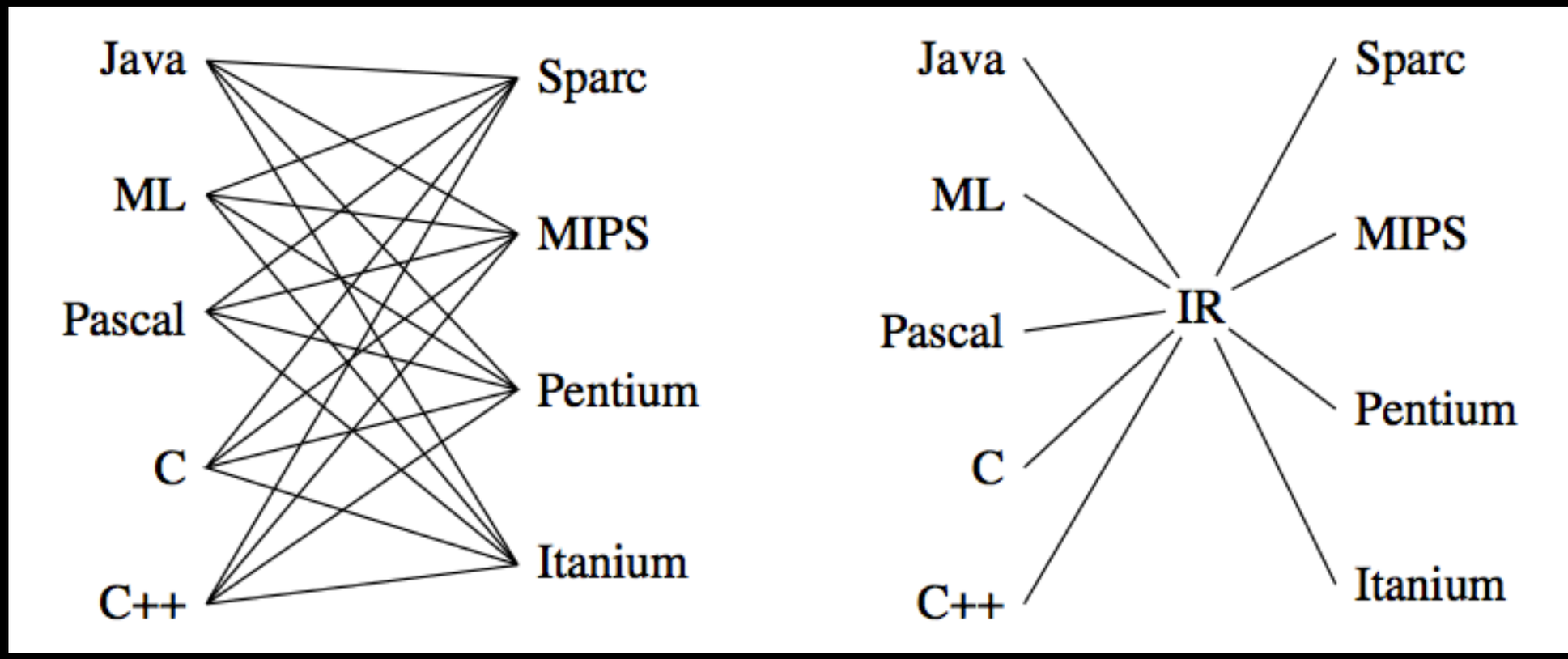
- Uma representação intermediária é uma linguagem abstrata que pode expressar operações de máquina sem se ater a muitos detalhes específicos de uma arquitetura

- Um compilador que visa portabilidade utiliza IR para modularizar as tarefas de tradução

Qual a principal vantagem de criar representações intermediárias?

Separação permite criar múltiplos compiladores





Adaptado de Andrew Appel
Modern Compiler Implementation in Java

Representação intermediária (IR)

- Precisa ser expressiva o suficiente para registrar fatos úteis que o compilador precisa transmitir
  - informações da análise semântica, por ex.
- Pode ser ‘aumentada’ com estruturas auxiliares
  - tabelas de símbolos

Representações Intermediárias

- Existem vários tipos de representações intermediárias
- Representações de alto e baixo nível
- A escolha varia de acordo com o compilador
  - o compilador original de C++ gerava C

Taxonomia

- *Graphical* IRs
  - DAGs, *parse trees*…
- *Linear* IRs
  - ILOC, 3-address code, SSA…
- *Hybrid* IRs
  - combinar elementos gráficos e lineares

Eixos

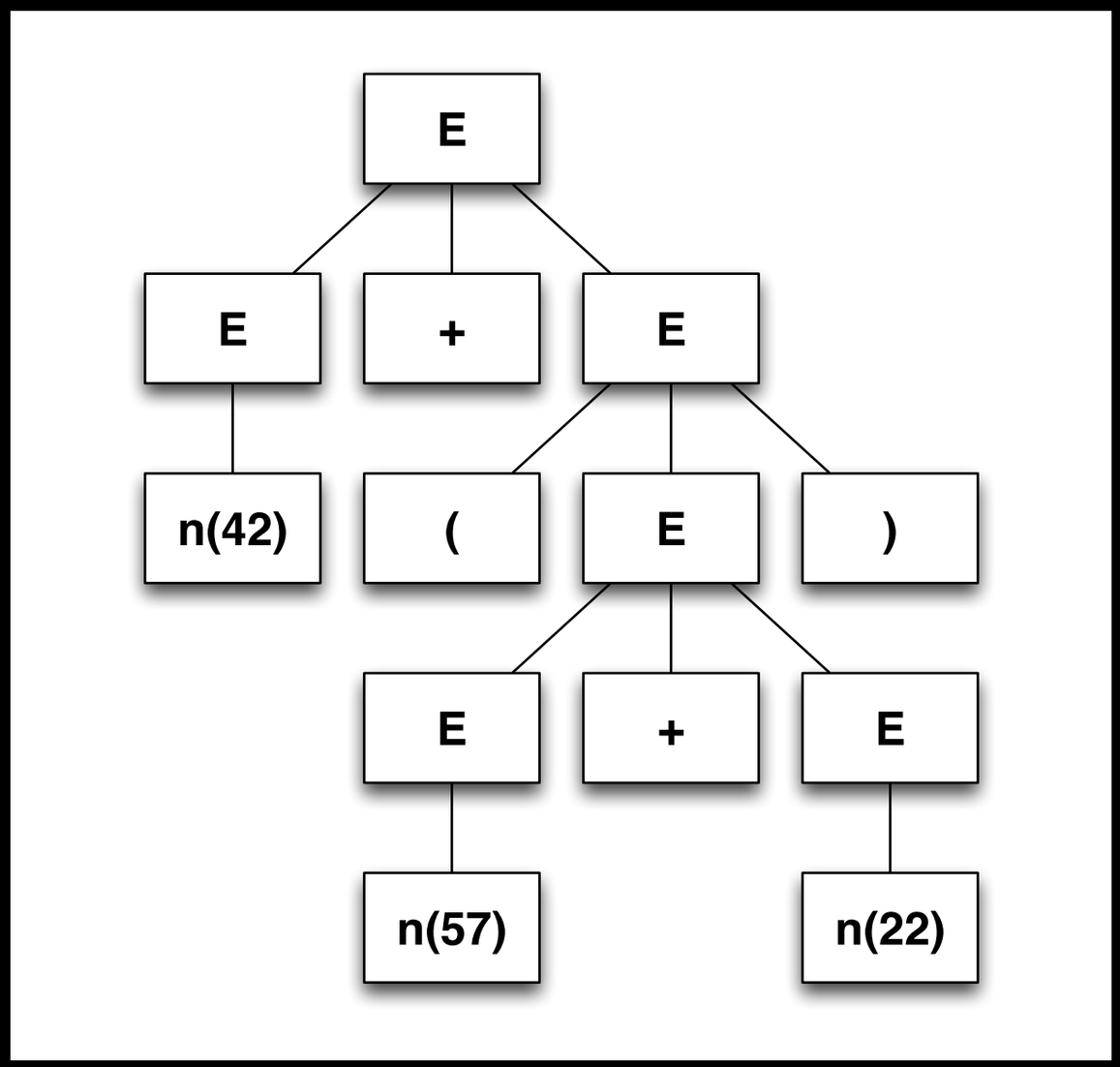
- *Organização Estrutural*
  - árvores vs. linear
- *Nível de abstração*
  - near-source vs. low-level
- *Nomeação*
  - esquema de nomes utilizado

Exemplos

- Direct Acyclic Graphs for Expressions
- Three Address code
- Static Single Assignment
- Control-flow, call, points-to graphs
- Tree expressions
- …

Parse Trees

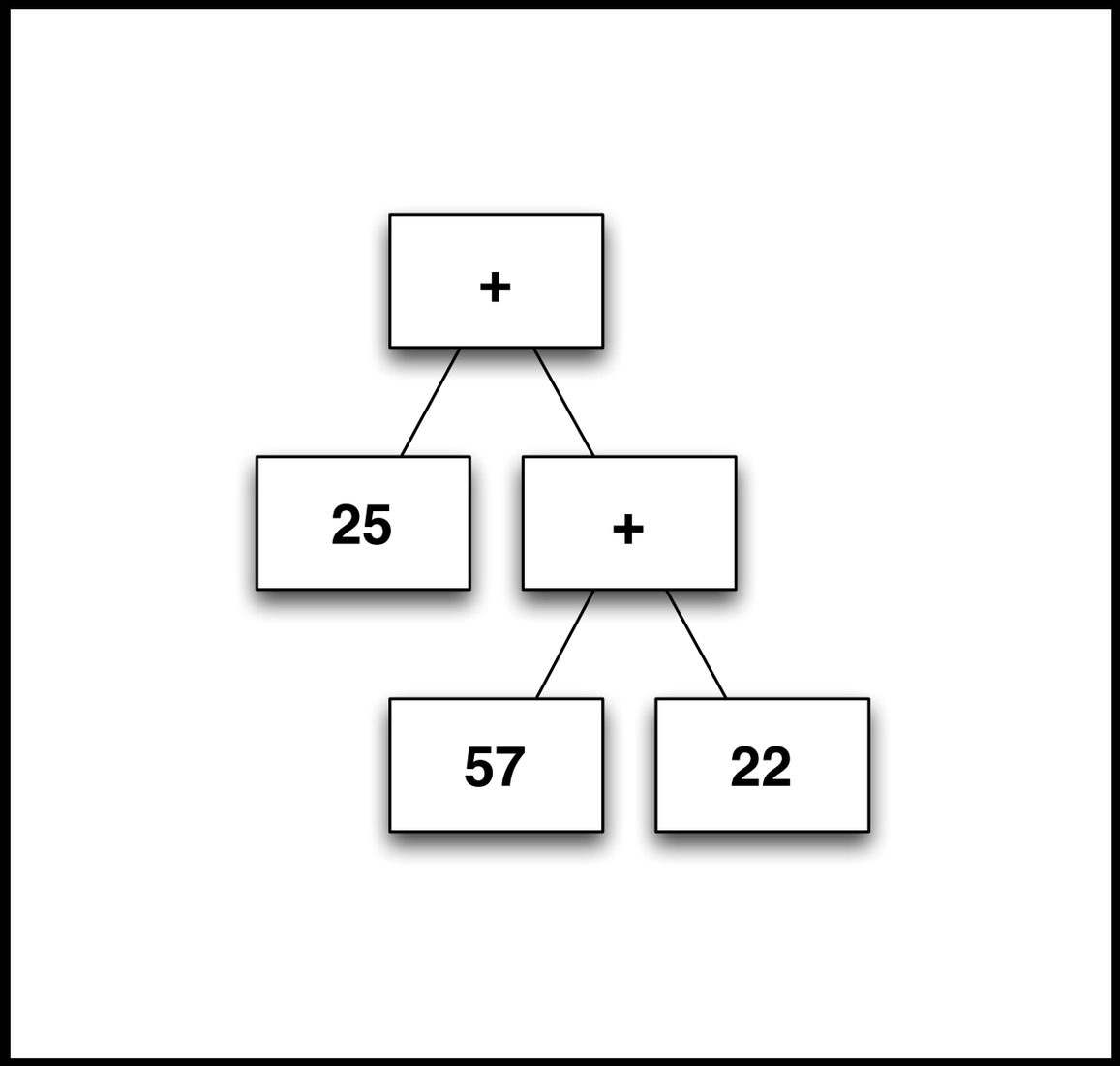
42 + (57 + 22)



$$E \rightarrow \mathrm{n} \mid (\, E \,) \mid E + E$$

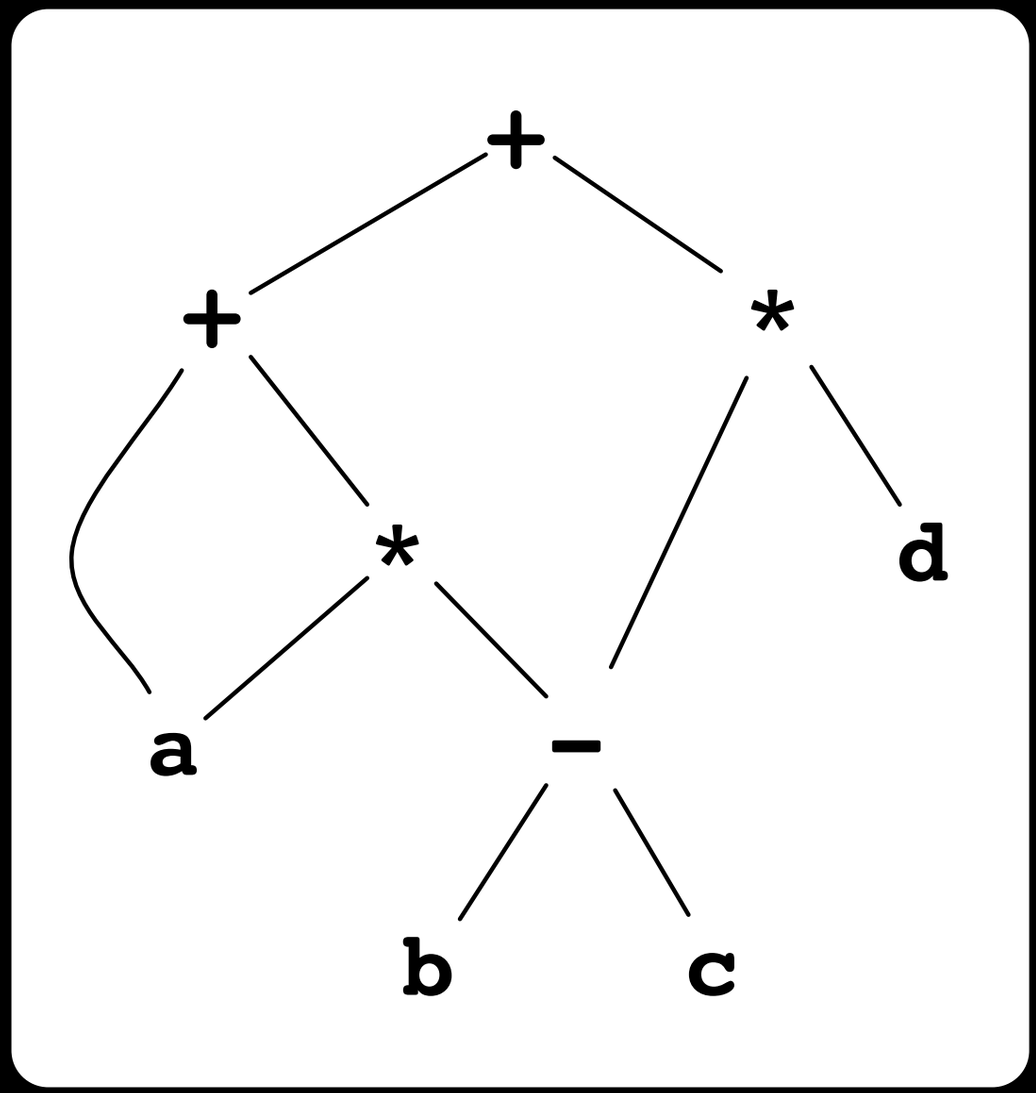
Abstract Syntax Trees

25 + (57 + 22)



$$E \rightarrow \mathrm{n} \mid (\ E\ ) \mid E + E$$

DAG for Expressions



`a+a*(b-c)+(b-c)*d`

Representações Lineares

- Alternativas a representações gráficas
- Sequências de instruções que executam em ordem, impondo uma ordem clara e útil
- Se aproximam de código assembly para uma máquina abstrata
- Geralmente precisa codificar mecanismos de transferência de controle entre pontos do programa (*jumps* e *conditional branches*)

Tipos de IR Lineares

- *One-address code*: modela o comportamento de acumuladores e máquinas baseadas em pilha; código compacto.

- *Two-address code*: modela máquinas que tem operações destrutivas, se tornou menos popular com a redução de restrições de memória

- *Three-address code*: modela máquinas em que a maioria das operações recebem dois operandos e produzem resultado (popularidade de RISC)

Stack-Machine Code

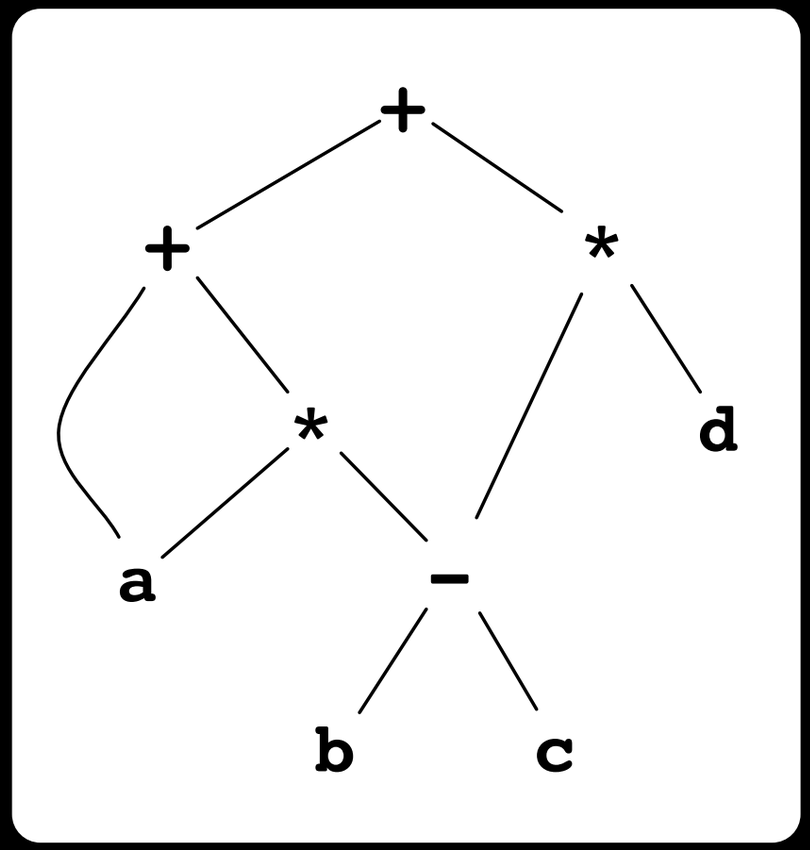
- Assume a presença de uma pilha de operandos
- Operações pegam operandos da pilha e empilham os resultados de volta
- Código compacto, não produz necessidade de nomeação que resulta de three-address code
- Pilha faz com que resultados sejam transitórios, a não ser que sejam salvos na memória

```
push 2
push b
multiply
push a
subtract
```

Código de Três Endereços

- Frequentemente usado como código intermediário
- Abstrai um *assembler*, onde cada instrução básica referencia no máximo 3 endereços
- No máximo um operador no lado direito das instruções
- Formato: x := y op z
- Exemplo: **x + y * z**
  é reescrito como t_1 := y * z
  t_2 := x + t_1

Representação Linear



$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

Conceitos Básicos

- Endereços e instruções
- Em termos OO, estes conceitos correspondem a classes e os vários tipos de endereços e instruções são subclasses

Endereços

- Nomes: variáveis do programa fonte
  - por conveniência utilizaremos o mesmo nome das variáveis;
  - na implementação um nome é associado com o ponteiro para a tabela de símbolos, onde informação é mantida

Endereços

- Constantes
  - pode exigir conversão de tipos (int to float, por ex.)
- Temporários
  - em compiladores que visam otimização, é útil gerar nomes distintos para variáveis temporárias;
  - podem ser combinados ao alocar registradores

Instruções

- Atribuições do tipo **`x = y op z`**, onde **`op`** é uma operação aritmética ou lógica e **`x`**, **`y`**, e **`z`** são endereços
- Atribuições do tipo **`x = op y`**, onde **`op`** é uma operação unária
- Cópia: **`x = y`**, onde **`x`** recebe valor de **`y`**

Instruções

- Desvio incondicional do tipo **`goto` *`L`***. A instrução marcada com **`L`** é a próxima a ser executada
- Desvios condicionais
  - **`if x goto` *`L`***
  - **`ifFalse x goto` *`L`***
  - **`if x relop y goto` *`L`***

Instruções

- Chamadas de procedimento
  - **`param` x_1**
  - **`param` x_2**
  - …
  - **`param` x_n**
  - **`call p,n`**
  - **`return y //opcional`**

Instruções

- Instruções indexadas
  - `x = y[i]`
  - `x[i] = y`

Instruções

- Atribuições de ponteiros
  - `x = &y`
  - `x = *y`
  - `*x = y`

```
do i=i+1; while(a[i]<v);
```

```
L: t1 = i+1
   i = t1
   t2 = i*8
   t3 = a[t2]
   if t3<v goto L
```

```
100: t1 = i+1
101: i = t1
102: t2 = i*8
103: t3 = a[t2]
104: if t3<v goto 100
```

Labels simbólicos vs. posições numéricas

Operadores

- Escolha de operadores é uma questão importante ao definir uma representação intermediária
- O conjunto de operadores deve ser expressivo o suficiente para implementar operações da linguagem fonte
- Nível de abstração pode facilitar/dificultar geração de IR e otimização de código

Estruturas de Dados

- A representação das instruções de três endereços pode se dar por meio de objetos e/ou registros com campos para os operadores e operandos
- Exemplos: quadruples, triples, indirect triples

Quadruples

a = b*-c+b*-c

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

op	arg_1	arg_2	*result*
minus	c		t_1
*	b	t_1	t_2
minus	c		t_3
*	b	t_3	t_4
+	t_2	t_4	t_5
=	t_5		a
…			

Quadruples

- Um “quad”, contém quatro campos: op, arg_1, arg_2, e result
- Instruções com operadores unários não usam arg_2
- Operadores como param, não usam arg_2 ou result
- Desvios colocam label em result

Array of Pointers

`a = b*-c+b*-c`

instrução	
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
…	

	op	*arg₁*	*arg₂*
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
…			

Gerando código IR

Assuma uma regra da gramática: $E \rightarrow E_1 + E_2$

E o caso de:

$S \rightarrow \textbf{id} := E$

?

Produção	Regra Semântica
$S \rightarrow \textbf{id} := E$	**S.code := E.code || gen(top.get(id.lexeme) '=' E.addr)**
$E \rightarrow E_1 + E_2$	**E.addr := new Temp();** **E.code := E_1.code || E_2.code || gen(E.addr '=' E_1.addr '+' E_2.addr)**
$E \rightarrow - E_1$	**E.addr := new Temp();** **E.code := E_1.code || gen(E.addr '= uminus' E_1.addr)**
$E \rightarrow (E_1)$	**E.addr := E_1.addr;** **E.code := E_1.code**
$E \rightarrow \textbf{id}$	**E.addr := top.get(id.lexeme);** **E.code := ''**

Acesso a *Arrays*

- Podem ser acessados rapidamente se forem armazenados em posições consecutivas
- Se a *largura* de cada elemento (espaço necessário) for w, então o i-ésimo elemento do *array* A está na localização: $base + i * w$
- A fórmula pode ser generalizada para duas ou mais dimensões
- Elementos podem não ser numerados a partir de 0, mas numerados a partir de um dado valor low, neste caso a fórmula fica: $base + (i-low) * w$

Expressões Booleanas

- Podem ser avaliadas de duas formas:
  - codificar true e false como números, e avaliar as expressões da mesma forma que expressões matemáticas.
  - usar o fluxo de controle, i.e. representar um valor por uma posição atingida no programa.
    - Usada em if-then-else, while-do etc. Permite "curto-circuito" de expressões: se E_1 é avaliado como true, na expressão E_1 or E_2, não é necessário avaliar E_2.

Short-Circuit Code

```
if (x < 100 || x > 200 && x !=y) x=0;
```

```
        if x<100 goto L2
        ifFalse x>200 goto L1
        ifFalse x!=y  goto L1
L2:     x = 0
L1:     …
```

Fluxo de controle

- Associamos a cada expressão booleana desvios para labels caso a expressão seja avaliada para *true* ou *false*.

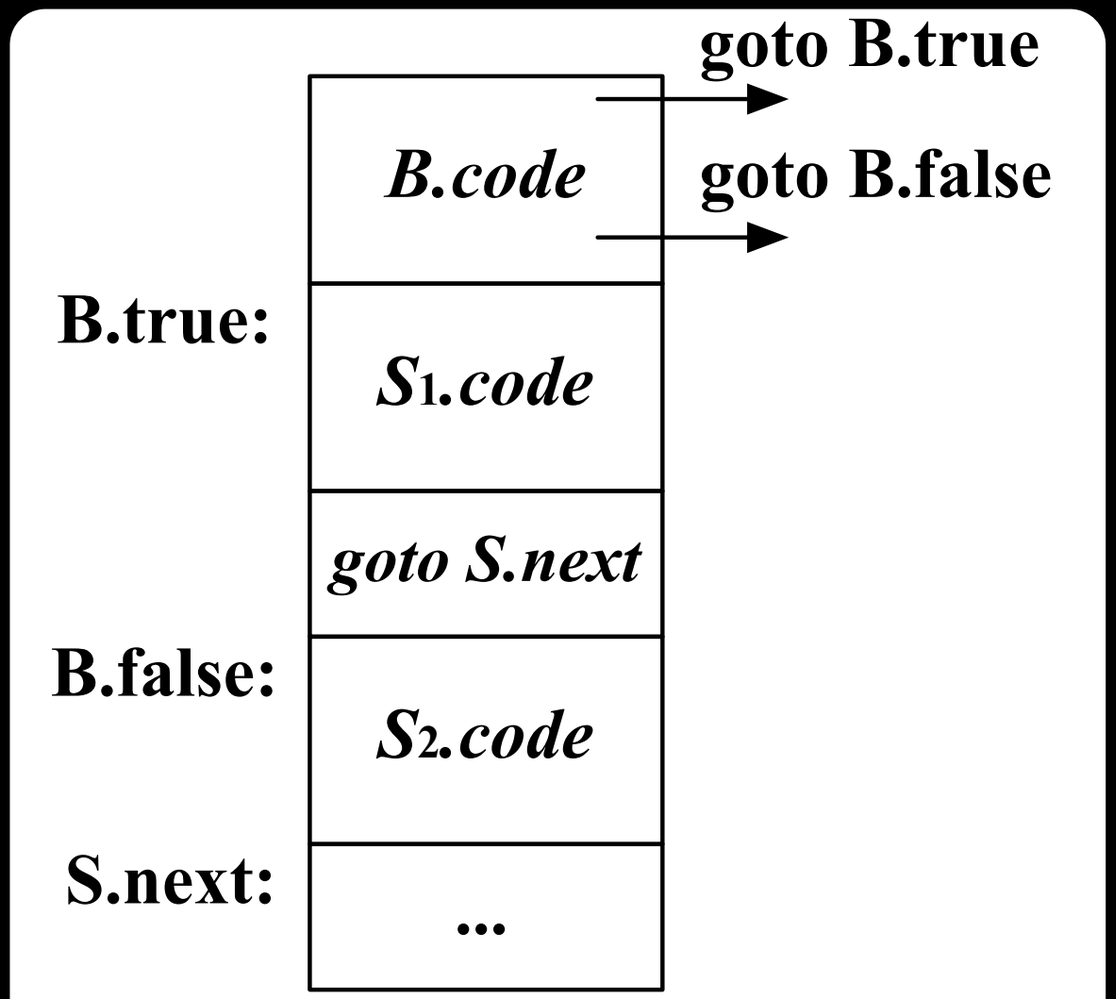
- E.true e E.false

Produção	Regra Semântica
$P \rightarrow S$	**S.next = newlabel();** **P.code := S.code || label(S.next)**
$S \rightarrow$ **assign**	**S.code := assign.code**
$S \rightarrow$ **if(**B**) then** S_1	

$$S \rightarrow \textbf{if}(B)\ \textbf{then}\ S_1$$

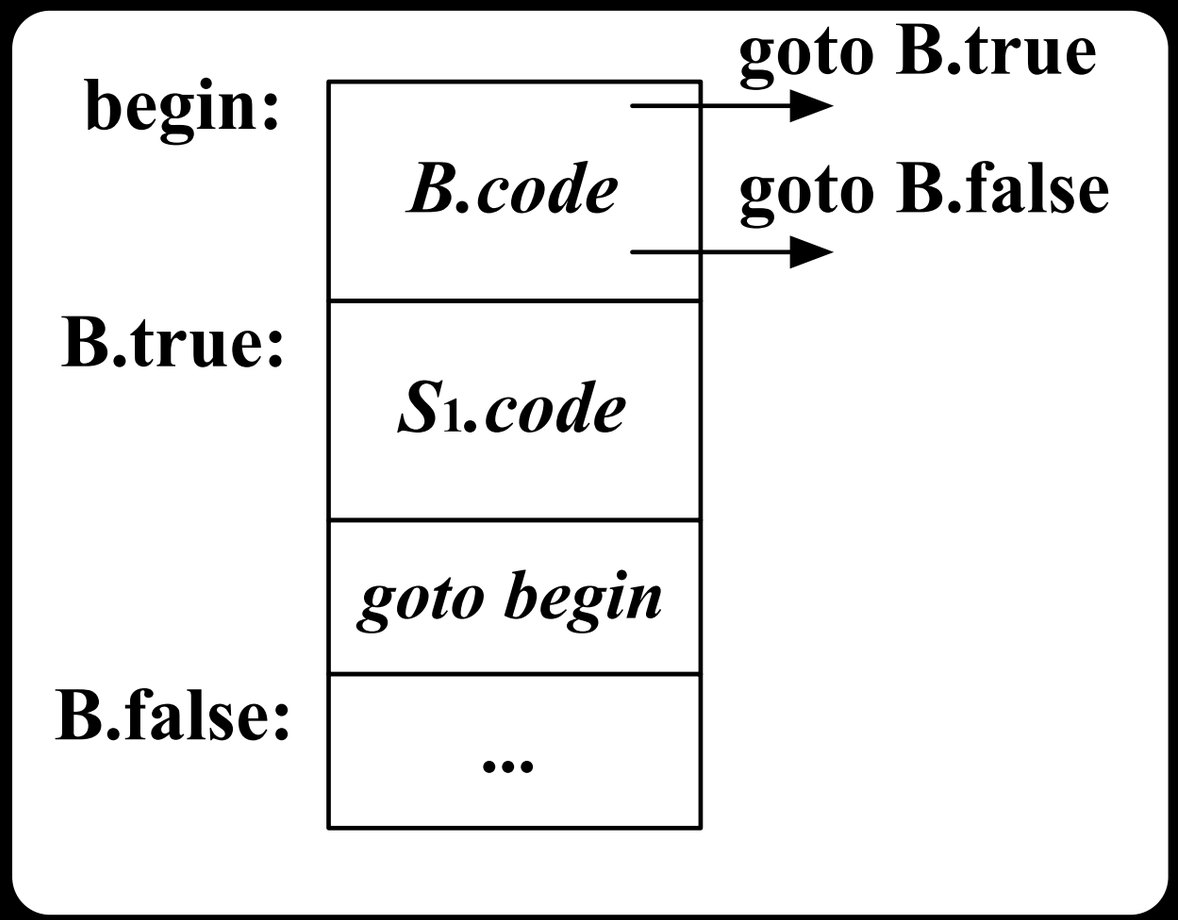
Produção	Regra Semântica
$P \rightarrow S$	**S.next = newlabel();** **P.code := S.code || label(S.next)**
$S \rightarrow$ **assign**	**S.code := assign.code**
$S \rightarrow$ **if(**B**) then** S_1	**B.true = newlabel();** **B.false = S_1.next = S.next;** **S.code := B.code || label(B.true) || S_1.code**

$S \rightarrow$ **if**(B) **then** S_1 **else** S_2



Produção	Regra Semântica
$P \rightarrow S$	**S.next = newlabel();** **P.code := S.code || label(S.next)**
$S \rightarrow$ **assign**	**S.code := assign.code**
$S \rightarrow$ **if(**B**) then** S_1	**B.true = newlabel();** **B.false = S_1.next = S.next;** **S.code := B.code || label(B.true) || S_1.code**
$S \rightarrow$ **if(**B**) then** S_1 **else** S_2	**B.true = newlabel(); B.false = newlabel();** **S_1.next = S_2.next = S.next;** **S.code := B.code || label(B.true) || S1.code** **|| gen('goto' S.next)** **|| label(B.false) || S2.code**

$S \rightarrow \textbf{while}(B)\ S_1$



Produção	Regra Semântica
$P \rightarrow S$	**S.next = newlabel();** **P.code := S.code || label(S.next)**
$S \rightarrow$ **assign**	**S.code := assign.code**
$S \rightarrow$ **if(**B**) then** S_1	**B.true = newlabel();** **B.false = S_1.next = S.next;** **S.code := B.code || label(B.true) || S_1.code**
$S \rightarrow$ **if(**B**) then** S_1 **else** S_2	**B.true = newlabel(); B.false = newlabel();** **S_1.next = S_2.next = S.next;** **S.code := B.code || label(B.true) || S1.code** **|| gen('goto' S.next)** **|| label(B.false) || S2.code**
$S \rightarrow$ **while(**B**)** S_1	**begin = newlabel(); B.true = newlabel(); B.false = S.next;** **S_1.next = begin;** **S.code := label(begin) || B.code ||** **label(B.true) || S_1.code || gen('goto' begin)**
$S \rightarrow S_1\ S_2$	**S_1.next = newlabel();** **S_2.next = S.next;** **S.code := S_1.code || label(S_1.next) || S_2.code**