GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

CAPÍTULO 6

Mariza A S. Bigonha e Roberto S. Bigonha

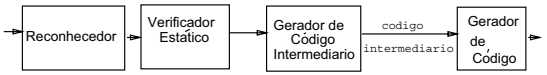
UFMG

9 de agosto de 2011

GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

- Vantagens
- Desvantagens

Estrutura lógica do *front-end* de um compilador:



· · · Geração de Código Intermediário

A verificação estática inclui:

- **Verificação de tipo**
- **Quaisquer verificações sintáticas que restarem após a análise sintática.**

  Exemplo: a verificação estática garante que um comando *break* em C esteja incorporado em um comando *while*, *for* ou *switch*.

  Se não houver uma dessas instruções envolventes, um erro é informado.

· · · Geração de Código Intermediário

Representações intermediárias de alto nível:

Estão próximas da linguagem fonte.

Representações de baixo nível:

Estão próximas da máquina alvo.

· · · Geração de Código Intermediário

- Formas
  - Notação posfixada.
  - Árvore de sintaxe.
  - Código de 3 endereços.
    - Quádruplas.
    - Triplas.
    - Triplas indiretas.
  - Forma de atribuição única estática (SSA)

· · · Geração de Código Intermediário

Representação de baixo nível: adequada para tarefas dependentes de máquina.
Exemplos: alocação de registradores e seleção de instrução.

O código de três endereços pode variar de alto até baixo nível, dependendo da escolha dos operadores.

Expressões: as diferenças entre as árvores de sintaxe e o código de três endereços superficiais.

Comandos de *loop*: uma árvore de sintaxe representa os componentes de um comando.

Código de três endereços contém rótulos e comandos de desvios para representar o fluxo de controle, como na linguagem de máquina.

· · · Geração de Código Intermediário

A escolha ou o projeto de uma representação intermediária varia de um compilador para outro.

Uma representação intermediária pode ser:

- uma linguagem de alto nível corrente.
- consistir em estruturas de dados internas que são compartilhadas pelas fases do compilador.

  Exemplo: o *front-end* do compilador C++ original gera C, tratando o compilador C como um *back-end*.

Notação Posfixada

- Exemplo 1:
- if A + B then A := C + D else A := −B + A

```
    A B +
    L1 jeqz
    A C D + :=
    L2 jump
L1: A B − A + :=
L2:
```

··· Notação Posfixada

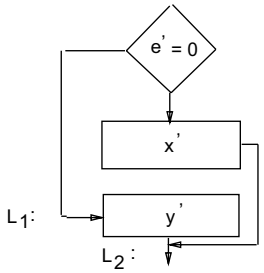
- Exemplo 2:

- a + b * c ⟶ abc*+

  if e then x else y

  ⇓

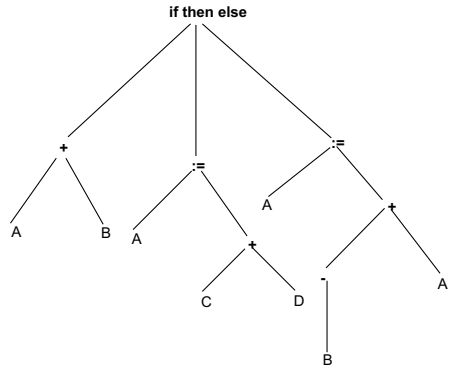
  $e' \; L_1 \; \text{jeqz} \; x' \; L_2 \; \text{jump} \; L_1 \text{:} \; y' L_2 \text{:}$



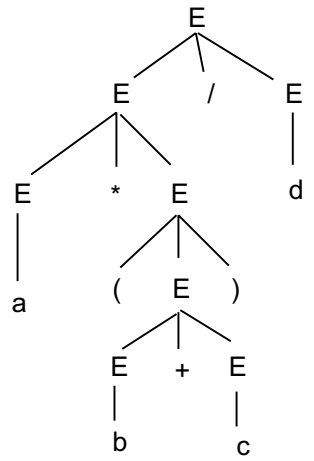
Árvore de Sintaxe

Nós: construções do programa fonte.
Filhos de um nó: componentes significativos de uma construção.

if A+B then A := C+D else A := −B+A

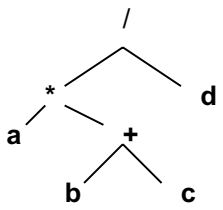


··· Árvore de Sintaxe

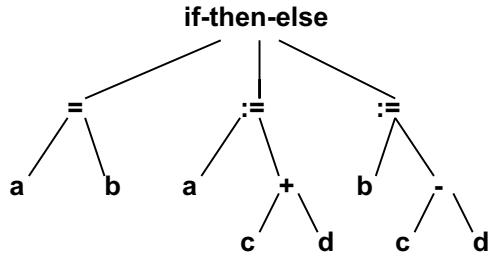


Outros Exemplos usando Árvore de Sintaxe

a * (b + c) / d



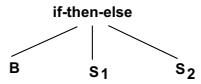
if a=b **then** a:=c+d **else** b:=c-d



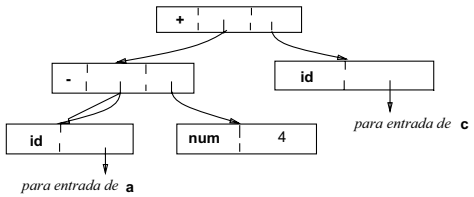
- Rótulos de folhas são, em geral, apontadores para a tabela de símbolos.

Construção de Árvores Sintáticas

• Árvores Sintáticas: **uma árvore sintática abstrata é uma forma condensada da árvore de reconhecimento muito útil para representar construções das linguagens.**



• Exemplo da árvore sintática para: a−4+c



Exemplos de Funções Usadas

MkNode (op, left, right): **cria um vértice operador com rótulo** op **e dois campos contendo apontadores para left e right.**

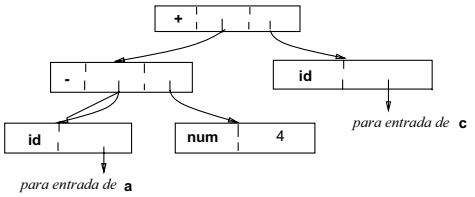
MkLeaf (id, entry) : **cria um vértice identificador com rótulo** id **e um campo contendo entry , um apontador para a entrada na Tabela de Símbolos para o identificador.**

MkLeaf (num, val) : **cria um vértice número com rótulo** num **e um campo contendo val , o valor do número.**

Definição Dirigida por Sintaxe para Produzir Árvores de Sintaxe ou DAGs

Produção	Regras Semânticas
1) $E \rightarrow E_1 + T$	$E.node =$ **new** $MkNode('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node =$ **new** $MkNode('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow$ id	$T.node =$ **new** $MkLeaf(\text{id}, \text{id}.entry)$
6) $T \rightarrow$ num	$T.node =$ **new** $MkLeaf(\text{num}, \text{num}.val)$

Exemplo da Construção de Árvores Sintáticas



(1) p_1 := MkLeaf (id, entrya);
(2) p_2 := MkLeaf (num, 4);
(3) p_3 := MkNode ("-" , p_1, p_2);
(4) p_4 := MkLeaf (id, entryc);
(5) p_5 := MkNode ("+" , p_3, p_4);

Note bem: **a árvore é construida bottom-up.**

Variantes das Árvores de Sintaxe:

Um grafo acíclico dirigido (DAG (Directed Acyclic Graph)) para uma expressão identifica as subexpressões comuns da expressão.

DAG possui folhas correspondendo aos operandos atômicos e códigos interiores correspondendo aos operadores.

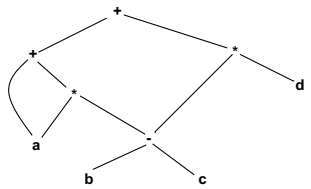
Diferença entre Árvore de Sintaxe e DAG

- Um nó N de um DAG tem mais de um pai se N representar uma subexpressão comum.
- Em uma árvore de sintaxe, a árvore para a subexpressão comum seria replicada tantas vezes quantas ocorresse a subexpressão na expressão original.

Um DAG não apenas representa as expressões mais sucintamente, como também fornece ao compilador dicas importantes em relação à geração de código eficiente para avaliar as expressões.

⋱ Grafos Acíclicos Dirigidos para Expressões

Exemplo de um DAG para a expressão:
a + a * (b − c) + (b − c) * d



• **Note bem:** *mkleaf* e *mknode* criam novos vértices somente quando necessário, retornando apontadores para vértices existentes, sempre que possível, com filhos e rótulos corretos. *mkleaf* (id,a) é repetida em 2, vértice construido pela chamada anterior *mkleaf* (id,a) é retornado, então p_1=p_2. Vértices retornados em 8 e 9 são os mesmos de 3 e 4. P., vértice em 10 ***deve ser o mesmo*** construido pela chamada *mknode* em 5.

Instruções para construir o DAG

(1) p_1:= *MkLeaf*(id, a);
(2) p_2:= *MkLeaf*(id, a);
(3) p_3:= *MkLeaf*(id, b);
(4) p_4:= *MkLeaf*(id, c);
(5) p_5:= *MkNode*("−" , p_3, p_4);
(6) p_6:= *MkNode*("*" , p_2, p_5);
(7) p_7:= *MkNode*("+" , p_1, p_6);
(8) p_8:= *MkLeaf*(id,b);
(9) p_9:= *MkLeaf*(id,c);
(10) p_{10}:= *MkNode*("−" ,p_8,p_9);
(11) p_{11}:= *MkLeaf*(id,d);
(12) p_{12}:= *MkNode*("*" ,p_{10},p_{11});
(13) p_{13}:= *MkNode*("+" ,p_7,p_{12});

O método Código Numérico para Construção de DAG

Freqüentemente, os nós de uma árvore de sintaxe ou DAG são armazenados em um arranjo de registros.

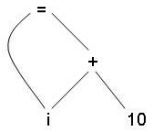
Cada linha do arranjo representa um registro, portanto um nó.

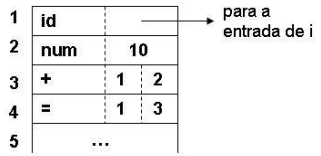
Em cada registro, o primeiro campo é um código de operação, indicando o rótulo do nó.

As folhas possuem um campo adicional, o qual contém o valor léxico e os nós interiores possuem dois campos adicionais indicando os filhos à esquerda e à direita.

⋱ O método Código Numérico para Construção de DAG

Nós de um DAG para $i = i + 10$ alocados em um arranjo





Nesse arranjo, a referência é feita aos nós dando o índice inteiro do registro para esse nó dentro do arranjo.

Esse inteiro tem sido historicamente chamado de código numérico para o nó ou para a expressão representada pelo nó.

No exemplo, o nó rotulado com "+" tem código numérico 3, e seus filhos da esquerda e da direita possuem os códigos numéricos 1 e 2, respectivamente.

O método Código Numérico para Construção de DAG

Na prática, é possível usar apontadores para os registros ou referências a objetos em vez de índices inteiros, MAS se armazenados em uma estrutura de dados apropriada, os códigos numéricos nos ajudam a construir os DAGs de expressão de forma eficiente.

Suponha que os nós sejam armazenados em um arranjo, como mostrado, e que cada nó seja referenciado por seu código numérico.

Seja a assinatura de um nó interior a tripla <op,l,r>, onde,

op é o rótulo,
l o código numérico de seu filho à esquerda, e
r o código numérico de seu filho à direita.

Um operador unário pode ser considerado como tendo $r = 0$.

O método Código Numérico para Construção de DAG

Algoritmo 6.3: Método código numérico para construir nós de um DAG.

ENTRADA: Rótulo op, nó l, e nó r.

SAÍDA: O código numérico de um nó no arranjo com assinatura $\langle op, l, r \rangle$.

MÉTODO:
Procure no arranjo um nó M com rótulo op, filho à esquerda l, e filho à direita r.

Se houver esse nó, retorne o código numérico de M.

Se não, crie no arranjo um novo nó N com rótulo op, filho à esquerda l e filho à direita r, e retorne seu código numérico.

O método Código Numérico para Construção de DAG

Considerações sobre o Algoritmo 6.3:

Embora o Algoritmo 6.3 gere a saída desejada, pesquisar o arranjo inteiro toda vez que tivermos de localizar um nó é muito caro, especialmente se o arranjo contiver expressões do programa todo.

Solução: usar uma tabela hash, na qual os nós são colocados em recipientes, cada um tendo tipicamente apenas alguns nós.

Para construir uma tabela hash com os nós de um DAG, é necessário uma função *hash* h que calcule o índice do recipiente para uma assinatura $\langle op, l, r \rangle$, distribuindo as assinaturas entre os recipientes, de tal modo que é improvável que qualquer recipiente receba muito mais do que uma fatia justa dos nós.

O índice do recipiente $h(op, l, r)$ é calculado deterministamente a partir de op, l, e r, de tal modo que é possível repetir o cálculo e sempre chegar ao mesmo índice do recipiente para o nó.

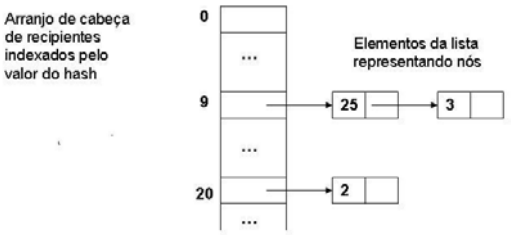
O método Código Numérico para Construção de DAG

Considerações sobre o Algoritmo 6.3:

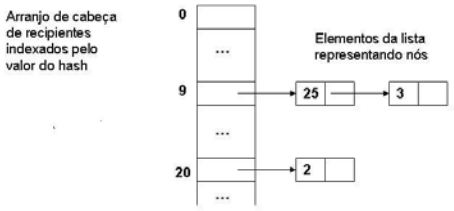
Implementação dos recipientes como listas encadeadas.

Um arranjo, indexado pelo valor hash, contém as *cabeças dos recipientes*, cada uma apontando para a primeira célula de uma lista.

Na lista encadeada para um recipiente, cada célula contém o código numérico de um dos nós alocados para esse recipiente. Ou seja, o nó op, l, e r pode ser encontrado na lista cuja cabeça esteja no índice $\langle op, l, r \rangle$ do arranjo.



· · · O método Código Numérico para Construção de DAG



Dado o nó de entrada op, l, e r, calcula-se o índice do recipiente $h(op, l, r)$ e procura-se na lista de células desse recipiente por um dado nó de entrada.

Pode ser necessário examinar todas as células dentro de um recipiente e, para cada código numérico v encontrado em uma célula, é preciso verificar se a assinatura $\langle op, l, r \rangle$ do nó de entrada casa com o nó de código numérico v na lista de células.

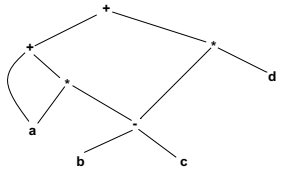
Se for encontrado um casamento, retorna-se v. Senão, sabe-se que nenhum nó desse tipo existe em nenhum outro recipiente, então é criada e incluída uma nova célula na lista de células para o índice do recipiente $h(op, l, r)$, e retorna-se o código numérico nessa nova célula.

Código de Três Endereços

- Seqüência de comandos da forma: x:=y op z
  onde:
  x, y e z são nomes, constantes, temporários gerados pelo compilador.
  op são operadores aritméticos e lógicos.

- Código de três endereços: representação linearizada da árvore de sintaxe ou de um dag .

DAG



Código de 3 endereços

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;

Endereços e Instruções

- Razão para o termo três endereços

  cada comando contém três endereços, dois para os operandos e um para o resultado.

O código de três endereços é construído a partir de dois conceitos:

endereços

instruções

Em termos da orientação por objetos, esses conceitos correspondem a classes, e os vários tipos de endereços e instruções correspondem a subclasses apropriadas.

· · · Endereços e Instruções

Um endereço pode ser um dos seguintes:

- Um nome. Nomes do programa fonte podem aparecer como endereços no código de 3 endereços. Em uma implementação, um nome do programa fonte é substituído por um apontador para sua entrada na tabela de símbolos, onde estão contidas todas as informações sobre este nome.

- Uma constante. Na prática, um compilador deve tratar com muitos tipos diferentes de constantes e variáveis.

- Um temporário gerado pelo compilador. É vantajoso, especialmente em compiladores otimizados, criar um nome distinto toda vez que um temporário é necessário. Esses temporários podem ser combinados, se possível, quando os registradores são alocados a variáveis.

Tipos de Comandos de Três Endereços

$$\begin{aligned} \mathsf{X} &:= (\mathsf{A} + \mathsf{B}) * \mathsf{C} \\ T_1 &:= \mathsf{A} + \mathsf{B} \\ T_2 &:= T_1 * \mathsf{C} \\ \mathsf{X} &:= T_2 \end{aligned}$$

- **1. Atribuição:** A := B op C
  A := op B
- **2. Comando de cópia:** x := y
- **3. Desvio condicional:** if A relop B goto L
- **4. Desvio incondicional:** goto L

Tipos de Comandos de Três Endereços

- **5. Chamada de procedimento:** param A_1
  ⋮ ⋮
  param A_n
  call P,n
- **6. Atribuição indexada:** A[I] := B
  B := A[I]
- **7. Atribuição com endereços e apontadores:** A := addr B
  C := ⋆A (⋆)
  ⋆A := C (⋆⋆)

(⋆) A é um apontador ou um temporário cujo r-value é uma localização. O r-value de C é feito igual ao conteúdo desta localização.
(⋆⋆) atribui ao r-value do objeto apontado por A o r-value de C.

Tipos de Comandos de Três Endereços

Exemplo:

A := if −B = A+C then A+B else −A

(1) $T_1 := -\mathsf{B}$
(2) $T_2 :=$ A+C
(3) if $T_1 \neq T_2$ goto (6)
(4) $T_3 :=$ A+B
(5) goto (7)
(6) $T_3 := -\mathsf{A}$
(7) A := T_3

Tipos de Comandos de Três Endereços

Outro Exemplo: `do` i = i+1; `while` (a[i] ¡ v);

Rótulos simbólicos.	Posição numérica.
L: t_1 = i + 1	**100:** t_1 = i + 1
i = t_1	**101:** i = t_1
t_2 = i * 8	**102:** t_2 = i * 8
t_3 = a [t_2]	**103:** t_3 = a [t_2]
if t_3 < v goto L	**104:** if t_3 < v goto 100

Implementação de Código de Três Endereços

Um comando de três endereços é uma forma abstrata do código intermediário.

A descrição de instruções de três endereços especifica os componentes de cada tipo de instrução, mas não especifica a representação dessas instruções em uma estrutura de dados.

- Implementação: como objetos ou registros com campos para os operandos e operador.

- Tipos de Representações:
  1. Quádruplas
  2. Triplas
  3. Triplas Indiretas.

Representação de Quádruplas

Uma quádrupla possui 4 campos: op, arg_1, arg_2, e $result$. O campo op contém um código interno para o operador.

Exceções a essa regra:

1. Instruções com operadores unários como x = `minus` y ou $x = y$ não usam arg_2. Observe que, para um comando de cópia como $x = y$, op é =, enquanto para a maioria das outras operações, o operador de atribuição é implícito.

2. Operadores como `param` não utilizam arg_2 nem $result$.

3. Comandos condicionais e incondicionais colocam o rótulo de destino em $result$.

··· Representação de Quádruplas

4 vetores:

	OPERADOR	OPERANDO1	OPERANDO2	OPERANDO3	
prox →	+	T_1	A	B	(1)
	=	T_1	0	(6)	(2)
	+	T_2	C	D	(3)
	:=	A	T_2		(4)
	goto	(9)			(5)

1 vetor:

	+	T_1	A	B	=	T_1	0	(6)	+	T_2	C	D	:=	A	T_2

prox →

- Todos campos inteiros.
- Operandos > 0 apontadores para tabela de símbolos.
- Operandos ≤ 0 representam constantes.

··· Representação de Quádruplas

Exemplo: Código de três endereços para a atribuição `a = b * - c + b * - c ;`

Código de três endereços

$$
\begin{aligned}
t_1 &= \text{minus c} \\
t_2 &= \text{b} * t_1 \\
t_3 &= \text{minus c} \\
t_4 &= \text{b} * t_3 \\
t_5 &= t_2 + t_4 \\
\text{a} &= t_5
\end{aligned}
$$

Quádruplas

	op	arg1	arg2	resultado
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

Representação de TRIPLAS

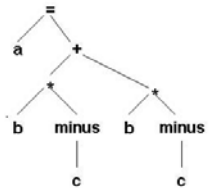
- Uma tripla tem apenas 3 campos: op, arg_1, e arg_2.
- Nas triplas, o resultado de uma operação $x\ op\ y$ é dado por sua posição, em vez de por um nome temporário explícito.
- Números entre parênteses: apontadores para a própria estrutura da tripla.

Exemplo de uma atribuição: `a = b*-c+b*-c;`

Quádruplas

	op	arg1	arg2	resultado
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

Árvore de sintaxe e triplas



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

··· Representação de TRIPLAS

```
A := -B * ( C + D )
```

(0) −, B
(1) +, C, D
(2) *, (0), (1)
(3) :=, A, (2)

```
A := B[ I ]
```

(0) = [ ], B, I
(1) :=, A, (0)

```
A[ I ] := B
```

(0) [ ]=, A, I
(1) :=, (0), B

triplas ≡ código de dois endereços.

Mais Exemplos de Triplas

if A + B then A := C + D else A := −B + A

- **Quádruplas**

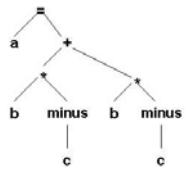
(1) T_1 := A + B
(2) if T_1 = 0 goto (6)
(3) T_2 := C + D
(4) A := T_2
(5) goto (9)
(6) T_3 := −B
(7) T_4 := T_3 + A
(8) A := T_4

- Triplas

(1) A + B	(1) +, A, B
(2) if (1) = 0 goto (6)	(2) if (1) = 0 goto (6)
(3) C + D	(3) +, C, D
(4) A := (3)	(4) :=, A, (3)
(5) goto (9)	(5) goto (9)
⋮	

Representação de Triplas Indiretas

Triplas indiretas: lista de apontadores para triplas, em vez de uma lista das próprias triplas.



instruções	
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)
	...		

Com triplas indiretas um compilador otimizado pode movimentar uma instrução reordenando a lista `instruções`, sem afetar as próprias triplas.

Implementação em Java: arranjo de objetos do tipo `instrução` é parecido com a representação de tripla indireta, pois Java trata elementos do arranjo como referências a objetos.

··· Representação de Triplas Indiretas

Outro Exemplo:

```
A := -B * ( C + D )
```

Comandos		Triplas	
(1)	(14)	(14)	−, B
(2)	(15)	(15)	+, C, D
(3)	(16)	(16)	*, (14), (15)
(4)	(17)	(17)	:=, A, (16)

Forma de Atribuição Única Estática

Atribuição única estática ((SSA) Static Single-Assignment) é uma representação intermediária que facilita certas otimizações de código.

Diferenças entre SSA e Código de 3 Endereços:

1. **Todas as atribuições em SSA são para variáveis com nomes distintos; daí o termo atribuição única estática.**

Código de 3 endereços. **Forma de atribuição única estática.**

```
p = a + b          p1 = a + b
q = p - c          q1 = p1 - c
p = q * d          p2 = q1 * d
p = e - p          p3 = e - p2
q = p + q          q2 = p3 + q1
```


··· Forma de Atribuição Única Estática

A mesma variável pode ser definida em dois caminhos de fluxo de controle diferentes em um programa.

Por exemplo, o programa fonte:

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

possui dois caminhos de fluxo de controle nos quais a variável `x` é definida.

··· Forma de Atribuição Única Estática

Usando diferentes nomes para `x` na parte verdadeira e na parte falsa do comando condicional,

que nome deve ser usado na atribuição `y = x * a`?

2. **SSA usa uma convenção notacional, chamada de função ϕ, para combinar as duas definições de `x`:**

```
if ( flag ) x1 = -1; else x2 = 1;
x3 = φ(x1, x2);
```

· · · Forma de Atribuição Única Estática

Aqui, $\phi(x_1, x_2)$ tem o valor

- x_1 se o fluxo de controle passar pela parte verdadeira do comando condicional,
- x_2 se o fluxo de controle passar pela parte falsa.

```
if ( flag ) x1 = -1; else x2 = 1;
x3 = φ(x1, x2);
```

Isso significa dizer que o valor do argumento da função f retornado corresponde ao caminho tomado no fluxo de controle para se chegar ao comando de atribuição contendo a função ϕ.

Comparação da Formas Intermediárias

Quádruplas e SSA	Triplas	Triplas Indiretas
Necessita gerência de temporários	—	—
Sabemos a área dos temps e seus tipos.	Necessita pesquisa no código.	Necessita pesquisa no código.
Fácil rearranjo do código (otimização)	Difícil.	Fácil.
Polui tabela de símbolos (temps).	—	—

FIM