MAB 471
2011.2

Análise Semântica

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

Além da Sintaxe

Há um nível de corretude além da gramática

```
foo(a,b,c,d) {
   int a, b, c, d;
     …
}
bar() {
   int f[3],g[0], h, i, j, k;
   char *p;
   foo(h,i,“ab”,j, k);
   k = f * i + j;
   h = g[17];
   printf(“<%s,%s>.\n”,p,q);
   p = 10;
}
```

Para gerar código, precisamos entender seu sigficado

Além da Sintaxe

Há um nível de corretude além da gramática

```
foo(a,b,c,d) {
   int a, b, c, d;
     …
}
bar() {
   int f[3],g[0], h, i, j, k;
   char *p;
   foo(h,i,“ab”,j, k);
   k = f * i + j;
   h = g[17];
   printf(“<%s,%s>.\n”,p,q);
   p = 10;
}
```

O que há de errado?

Para gerar código, precisamos entender seu sigficado

Além da Sintaxe

Há um nível de corretude além da gramática

```
foo(a,b,c,d) {
   int a, b, c, d;
     …
}
bar() {
   int f[3],g[0], h, i, j, k;
   char *p;
   foo(h,i,“ab”,j, k);
   k = f * i + j;
   h = g[17];
   printf(“<%s,%s>.\n”,p,q);
   p = 10;
}
```

O que há de errado?

(vamos contar…)

Para gerar código, precisamos entender seu sigficado

Além da Sintaxe

Há um nível de corretude além da gramática

```
foo(a,b,c,d) {
   int a, b, c, d;
     …
}
bar() {
   int f[3],g[0], h, i, j, k;
   char *p;
   foo(h,i,“ab”,j, k);
   k = f * i + j;
   h = g[17];
   printf(“<%s,%s>.\n”,p,q);
   p = 10;
}
```

O que há de errado?

(vamos contar…)

- número de argumentos de foo()

Para gerar código, precisamos entender seu sigficado

Além da Sintaxe

Há um nível de corretude além da gramática

```
foo(a,b,c,d) {
   int a, b, c, d;
     …
}
bar() {
   int f[3],g[0], h, i, j, k;
   char *p;
   foo(h,i,“ab”,j, k);
   k = f * i + j;
   h = g[17];
   printf(“<%s,%s>.\n”,p,q);
   p = 10;
}
```

O que há de errado?

(vamos contar…)

- número de argumentos de foo()
- declarou g[0], usou g[17]

Para gerar código, precisamos entender seu sigficado

Além da Sintaxe

Há um nível de corretude além da gramática

```
foo(a,b,c,d) {
   int a, b, c, d;
     …
}
bar() {
   int f[3],g[0], h, i, j, k;
   char *p;
   foo(h,i,“ab”,j, k);
   k = f * i + j;
   h = g[17];
   printf(“<%s,%s>.\n”,p,q);
   p = 10;
}
```

O que há de errado?

(vamos contar…)

- número de argumentos de foo()
- declarou g[0], usou g[17]
- “ab” não é int

Para gerar código, precisamos entender seu sigficado

Além da Sintaxe

Há um nível de corretude além da gramática

```
foo(a,b,c,d) {
   int a, b, c, d;
     …
}
bar() {
   int f[3],g[0], h, i, j, k;
   char *p;
   foo(h,i,“ab”,j, k);
   k = f * i + j;
   h = g[17];
   printf(“<%s,%s>.\n”,p,q);
   p = 10;
}
```

O que há de errado?

(vamos contar…)

- número de argumentos de foo()
- declarou g[0], usou g[17]
- “ab” não é int
- dimensão errada no uso de f

Para gerar código, precisamos entender seu sigficado

Além da Sintaxe

Há um nível de corretude além da gramática

```
foo(a,b,c,d) {
   int a, b, c, d;
     …
}
bar() {
   int f[3],g[0], h, i, j, k;
   char *p;
   foo(h,i,“ab”,j, k);
   k = f * i + j;
   h = g[17];
   printf(“<%s,%s>.\n”,p,q);
   p = 10;
}
```

O que há de errado?

(vamos contar…)

- número de argumentos de foo()
- declarou g[0], usou g[17]
- “ab” não é int
- dimensão errada no uso de f
- variável não declarada q

Para gerar código, precisamos entender seu sigficado

Além da Sintaxe

Há um nível de corretude além da gramática

```
foo(a,b,c,d) {
   int a, b, c, d;
     …
}
bar() {
   int f[3],g[0], h, i, j, k;
   char *p;
   foo(h,i,“ab”,j, k);
   k = f * i + j;
   h = g[17];
   printf(“<%s,%s>.\n”,p,q);
   p = 10;
}
```

O que há de errado?

(vamos contar…)

- número de argumentos de foo()
- declarou g[0], usou g[17]
- “ab” não é int
- dimensão errada no uso de f
- variável não declarada q
- 10 não é uma string

Tudo isso está além da sintaxe

Para gerar código, precisamos entender seu sigficado

Além da Sintaxe

Para gerar código, o compilador deve responder as questões:

- "x" é escalar, array, ou função? "x" foi declarado?
- Há nomes não declarados? Declarados mas não usados?
- Qual declaração de "x" é usada por uma ref. a "x"?
- A expressão "x * y + z" é corretamente tipada?
- Em "a[i,j,k]", a tem três dimensões?
- Onde "z" pode ficar? (registrador, local, global, heap, estática)
- Em "f ← 15", como representar 15?
- Quantos argumentos "foo()" recebe? E "printf()" ?
- "*p" referencia o resultado de um "malloc()" ?
- "p" & "q" se referem ao mesmo local na memória?
- "x" é definido a

Além do poder expressivo de uma CFG

Além da Sintaxe

Essas questões são parte da análise semântica

- Respostas dependem de valores, não de categorias sintáticas
- Questões e respostas usam informação não-local
- Respostas podem precisar de computação

Como responder essas questões?

- Usar métodos formais
  - Gramáticas sensíveis ao contexto?
  - Gramáticas de Atributos
- Usar técnicas ad-hoc
  - Tabelas de símbolos
  - Código ad-hoc

Além da Sintaxe

Essas questões são parte da análise semântica

- Respostas dependem de valores, não de categorias sintáticas
- Questões e respostas usam informação não-local
- Respostas podem precisar de computação

Como responder essas questões?

- Usar métodos formais
  - Gramáticas sensíveis ao contexto?
  - Gramáticas de Atributos
- Usar técnicas ad-hoc
  - Tabelas de símbolos
  - Código ad-hoc

Em análise sintática os formalismos ganharam.

Além da Sintaxe

Essas questões são parte da análise semântica

- Respostas dependem de valores, não de categorias sintáticas
- Questões e respostas usam informação não-local
- Respostas podem precisar de computação

Como responder essas questões?

- Usar métodos formais
  - Gramáticas sensíveis ao contexto?
  - Gramáticas de Atributos
- Usar técnicas ad-hoc
  - Tabelas de símbolos
  - Código ad-hoc

Em análise sintática os formalismos ganharam.

Em análise semântica as ténicas ad-hoc ganharam.

Análise Ad-hoc

Tradução dirigida por sintaxe

- Usa parser shift-reduce bottom-up
- Associa trecho de código a cada produção
- Executa o trecho a cada redução
- Código arbitrário dá muita flexibilidade
  - Inclusive a habilidade de dar um tiro no próprio pé

Para fazer funcionar

- Precisa de nomes para cada símbolo de uma regra gramatical
  - Yacc e derivados usam **$$**, **$1**, **$2**, … **$n**, esquerda pra direita
- Mecanismo de avaliação pós-ordem
  - Natural no algoritmo LR(1)

Exemplo — Tipagem

- Assume tabelas de tipagem F_+, F_-, $F_\times$, e $F_\div$

$F_\times$	Int 16	Int 32	Float	Double
Int 16	Int 16	Int 32	Float	Double
Int 32	Int 32	Int 32	Float	Double
Float	Float	Float	Float	Double
Double	Double	Double	Double	Double

1	*Goal*	→	*Expr*	$$ = $1;
2	*Expr*	→	*Expr* + *Term*	$$ = F_+($1,$3);
3		|	*Expr* - *Term*	$$ = F_-($1,$3);
4		|	*Term*	$$ = $1;
5	*Term*	→	*Term* * *Factor*	$$ = $F_\times$($1,$3);
6		|	*Term* / *Factor*	$$ = $F_\div$($1,$3);
7		|	*Factor*	$$ = $1;
8	*Factor*	→	(*Expr*)	$$ = $2;
9		|	number	$$ = type of num;
10		|	ident	$$ = type of ident;

Exemplo — Construção de AST

- Assume construtores para cada nó
- Assume que a pilha guarda ponteiros pros nós

1	*Goal*	→	*Expr*	$$ = $1;
2	*Expr*	→	*Expr* + *Term*	$$ = MakeAddNode($1,$3);
3		|	*Expr* - *Term*	$$ = MakeSubNode($1,$3);
4		|	*Term*	$$ = $1;
5	*Term*	→	*Term* * *Factor*	$$ = MakeMulNode($1,$3);
6		|	*Term* / *Factor*	$$ = MakeDivNode($1,$3);
7		|	*Factor*	$$ = $1;
8	*Factor*	→	(*Expr*)	$$ = $2;
9		|	number	$$ = MakeNumNode(token);
10		|	ident	$$ = MakeIdNode(token);

Example — Emissão de IR linear

- Assume que NextRegister() retorna nome de reg. virtual
- Assume que Emit() formata código assembly

1	*Goal*	→	*Expr*	
2	*Expr*	→	*Expr* + *Term*	$$ = *NextRegister*(); *Emit*(add, $1,$3, $$);
3		|	*Expr* - *Term*	$$ = *NextRegister*(); *Emit*(sub, $1, $3, $$);
4		|	*Term*	$$ = $1;
5	*Term*	→	*Term* * *Factor*	$$ = *NextRegister*(), *Emit*(mult,$1, $3, $$)
6		|	*Term* / *Factor*	$$ = *NextRegister*()' *Emit*(div, $1, $3, $$);
7		|	*Factor*	$$ = $1;

Example — Emissão de IR linear

- Assume que NextRegister() retorna nome de reg. virtual
- Assume que Emit() formata código assembly
- Assume que EmitLoad() lida com endereçamento e carrega um valor em um registrador

8	*Factor* →	(*Expr*)	$$ = $2;
9	|	number	$$ = *NextRegister*(); *Emit*(loadi, Value(lexeme), $$);
10	|	ident	$$ = *NextRegister*(); *EmitLoad*(ident, $$);

Usos Típicos

- Construção de tabela de símbolos
  - Entra informação de declarações à medida que são processadas
  - Usa tabela para checar erros à medida que o parsing progride

- Checagem de erros e tipos simples
  - Definição antes de uso → busca na referência
  - Dimensão, tipo, ... → checado quando encontrado
  - Consistência dos tipos de uma expressão → na redução da exp.
  - Interfaces de procedimentos são mais difíceis se não quiser impor definições antes de usos (ou protótipos)
    - → Construir representação para listas de parâmetros e tipos
    - → Criar lista de sítios para checagem
    - → Checagem offline

Limitações

- Força avaliação em uma ordem específica: pós-ordem
  - Esquerda pra direita
  - Bottom up

- Implicações
  - Declarações antes de usos
  - Informação de contexto não pode ser passada para "baixo"
    - → Como saber de dentro de qual regra você foi chamado?
  - Poderia usar globais?
    - → Requer inicialização e pensar direito nas soluções
  - Pode influenciar o projeto da linguagem, para ser mais fácil de analisar dessa maneira

Tradução Dirigida por Sintaxe

Como encaixar em um parser LR(1)?

```
stack.push(INVÁLIDO);
stack.push(s0);                          // estado inicial
token = scanner.next_token();
loop {
        s = stack.top();
        if ( ACTION[s,token] == "reduce A→β" ) then {
            stack.popnum(2*|β|);         // desempilha 2*|β| símbolos
          s = stack.top();
          stack.push(A);                 // empilha A
          stack.push(GOTO[s,A]);         // empilha próximo estado
        }
        else if ( ACTION[s,token] == "shift si" ) then {
                stack.push(token); stack.push(si);
                token ← scanner.next_token();
        }
        else if ( ACTION[s,token] == "accept"
                                & token == EOF )
                then break;
        else erro de sintaxe;
}
```

Parser LR(1) com ações TDS

```
stack.push(INVÁLIDO);
stack.push(NULL);
stack.push(s0);                    // estado inicial
token = scanner.next_token();
loop {
        s = stack.top();
        if ( ACTION[s,token] == "reduce A→β" ) then {

          /* insira ações aqui (switch) */

          stack.popnum(3*|β|);      // desempilha 3*|β| símbolos
          s = stack.top();
          stack.push(A);            // empilha A
          stack.push(GOTO[s,A]);    // empilha próximo estado
        }
        else if ( ACTION[s,token] == "shift si" ) then {
                stack.push(token); stack.push(si);
                token ← scanner.next_token();
        }
        else if ( ACTION[s,token] == "accept"
                                & token == EOF )
                then break;
        else erro de sintaxe;
}
```

Para adicionar ações YACC:

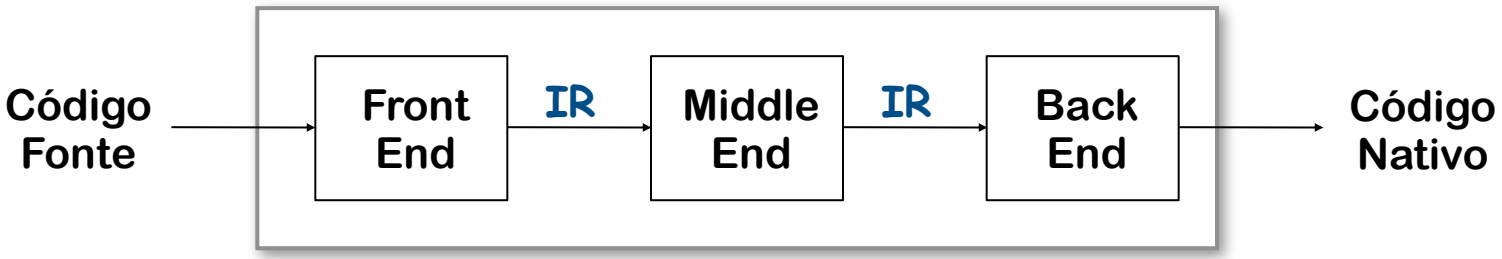
- Empilhe 3 items por símbolo ao invés de 2 (3º é $$)
- Switch na seção de processar reduções
  - → Switch no número da produção
  - → Cada cláusula tem o trecho de código para aquela produção
  - → Substitui nomes apropriados para $$, $1, $2, ...
- Aumento modesto no tempo
- 50% de aumento na pilha

Análise em ASTs

E se é preciso fazer ações que não se encaixam bem no framework da tradução dirigida por sintaxe?

- Construir a AST usando tradução dirigida por sintaxe
- Fazer as ações em uma ou mais passagens pela árvore
  - Várias maneiras de se estruturar em uma linguagem OO
  - Faz computação arbitrária e controla a ordem
  - Múltiplas passadas se necessário

Representações Intermediárias



- Front end - produz uma representação intermediária (IR)
- Middle end - transforma a IR do front end em uma IR equivalente que é mais eficiente (otimização)
- Back end - transforma a IR final em código nativo

- IR codifica conhecimento do compilador sobre o programa

Representações Intermediárias

- Decisões no projeto da IR afetam a eficiência do compilador e do código que ele gera

- Propriedades importantes
  - Facilidade de geração
  - Facilidade de manipulação
  - Tamanho dos programas
  - Expressividade
  - Nível de Abstração

- A importância de diferentes propriedades varia entre diferentes compiladores
  - Escolher a IR apropriedada é fundamental

Tipos de Representações Intermediárias

Três grandes categorias

- Estrutural
  - Gráficas
  - Muito usada em tradução fonte para fonte
  - Tende a ser grande

Exemplo:
ASTs

- Linear
  - Pseudo-código para máquina abstrata
  - Nível de abstração varia
  - Simples e compacta
  - Mais fácil de rearrumar o código

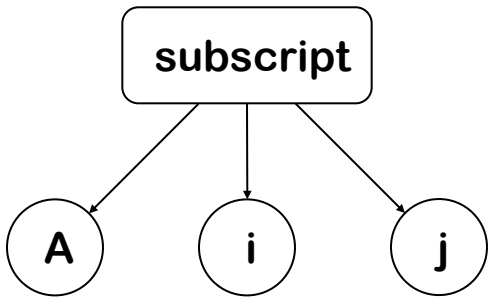
Exemplos:
Código 3 endereços
Código máq. de pilha

- Híbrida
  - Combinação de grafos e código linear
  - Grafos de fluxo de controle

Exemplo:
CFGs

Nível de Abstração

- O nível de detalhe exposto em uma IR influencia a possibilidade de diferentes otimizações
- Duas representações para acesso a array:



AST alto nível:
Boa para desambiguar acessos

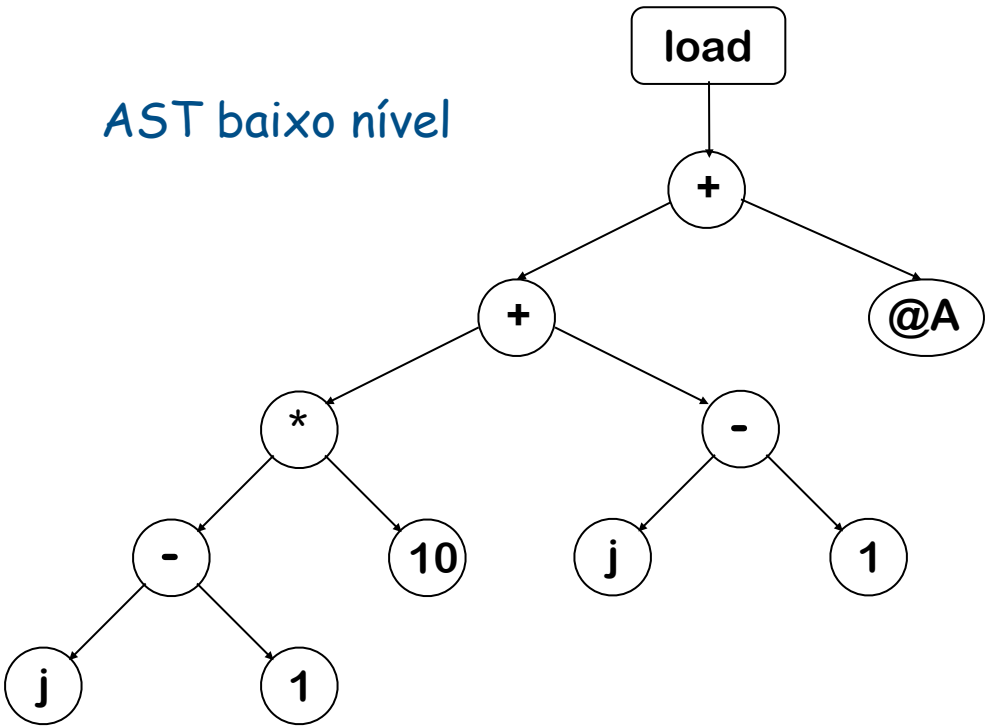
```
loadI 1       => r1
sub   rj, r1  => r2
loadI 10      => r3
mult  r2, r3  => r4
sub   ri, r1  => r5
add   r4, r5  => r6
loadI @A      => r7
add   r7, r6  => r8
load  r8      => rAij
```

Código linear de baixo nível:
Bom para cálculo de endereço mais eficiente

Nível de Abstração

- IRs estruturais normalmente são alto nível
- IRs lineares normalmente são baixo nível
- Não necessariamente verdade:

AST baixo nível

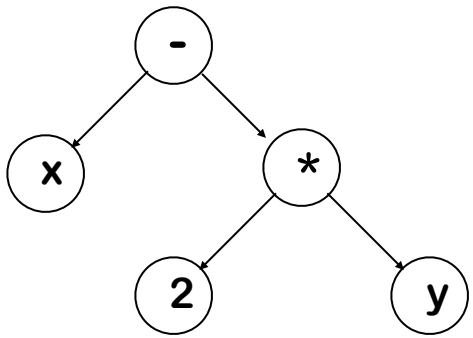



```
loadArray A,i,j
```

Código lin. alto nível

Árvore de Sintaxe Abstrata (AST)

Uma árvore de sintaxe abstrata é a árvore sintática com os nós para a maioria dos não-terminais removido



x - 2 * y

- Pode usar forma linearizada da árvore
  - Mais fácil de manipular do que ponteiros

    `x 2 y * -` em forma pós-fixada

    `- * 2 y x` em forma pré-fixada
- S-expressions (Scheme, Lisp) e XML são essencialmente ASTs

Código de Máquina de Pilha

Burroughs B-5000, p-code Pascal, Smalltalk, Java

- Exemplo:

  x - 2 * y vira

```
push x
push 2
push y
mul
sub
```

Vantagens

- Compacta (muitas operações precisam só de 1 byte - bytecode)
- Nomes dos temporários são implícitos
- Simples de gerar código e executar

Útil para transmissão de código

Código de Três Endereços

Muitas representações diferentes

- Em geral, código de 3 endereços tem comandos da forma:

$$x \leftarrow y \ \underline{op} \ z$$

Com 1 operador (op) e até 3 nomes (x, y, & z)

Exemplo:

$z \leftarrow x - 2 * y$ vira

Vantagens:

- Lembra uma máquina RISC simples
- Introduz nomes para temporários
- Forma compacta

Código de Três Endereços

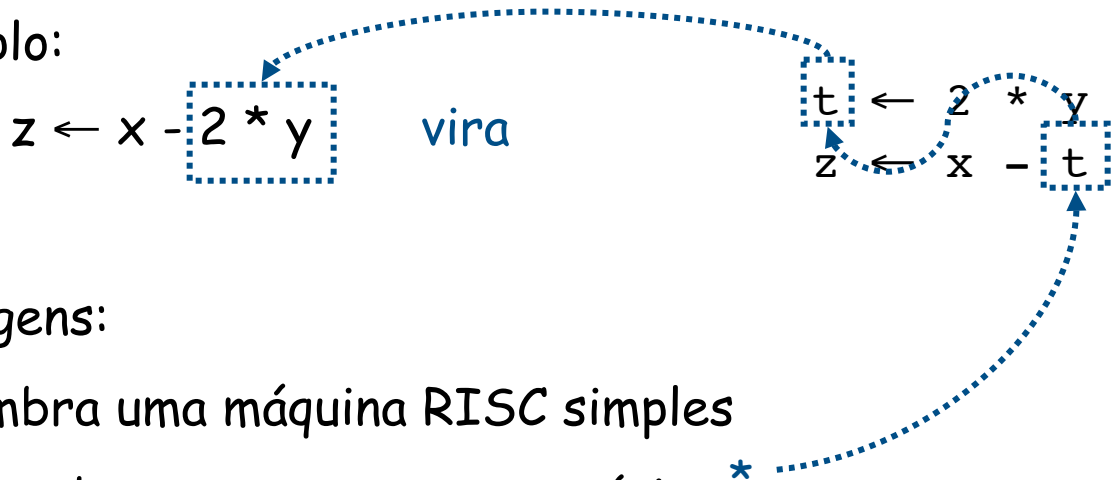
Muitas representações diferentes

- Em geral, código de 3 endereços tem comandos da forma:

$$x \leftarrow y \ \underline{op} \ z$$

Com 1 operador (op) e até 3 nomes (x, y, & z)

Exemplo:



Vantagens:

- Lembra uma máquina RISC simples
- Introduz nomes para temporários *
- Forma compacta

Código de 3 Endereços: Quádruplas

Representação simples de código de 3 endereços

- Tabela de k * 4 inteiros
- ou vetor de registros
- Fácil de reordenar
- Nomes explícitos

O compilador FORTRAN original usava "quads"

```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

Código de 3 endereços

load	1	y	
loadi	2	2	
mult	3	2	1
load	4	x	
sub	5	4	3

Quádruplas

Código de 3 Endereços: Triplas

- Índice é nome implícito do destino
- 25% menos espaço que quads
- Muito mais difícil de reordenar, a não ser que índices nas operações sejam ponteiros

(1)	`load`	`y`	
(2)	`loadI`	`2`	
(3)	`mult`	`(1)`	`(2)`
(4)	`load`	`x`	
(5)	`sub`	`(4)`	`(3)`

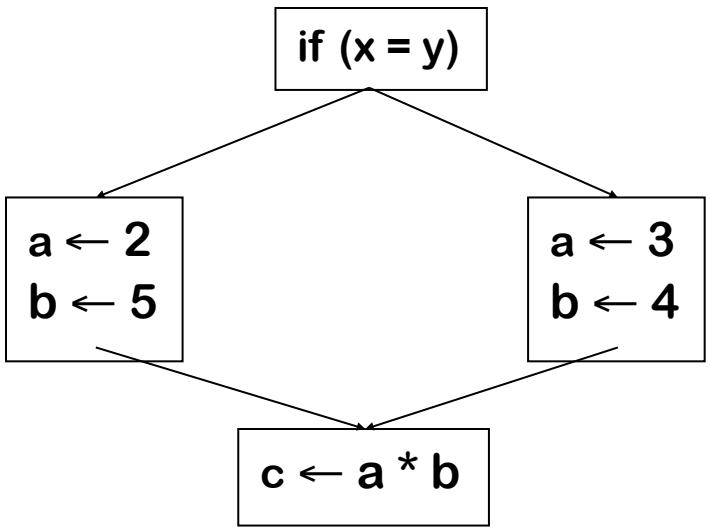
Nomes implícitos não ocupam espaço

Grafo de Fluxo de Controle

Modela a transferência de controle em um procedimento

- Nós do grafo são blocos básicos
  - Pode usar quads, triplas ou outra representação linear
- Arestas no grafo representam fluxo de controle

Exemplo



Forma SSA

- Ideia principal: definir cada nome apenas uma vez
- Introduzir funções ϕ (seleção) para fazer funcionar

Original

```
x ← …
y ← …
while (x < k)
   x ← x + 1
   y ← y + x
```

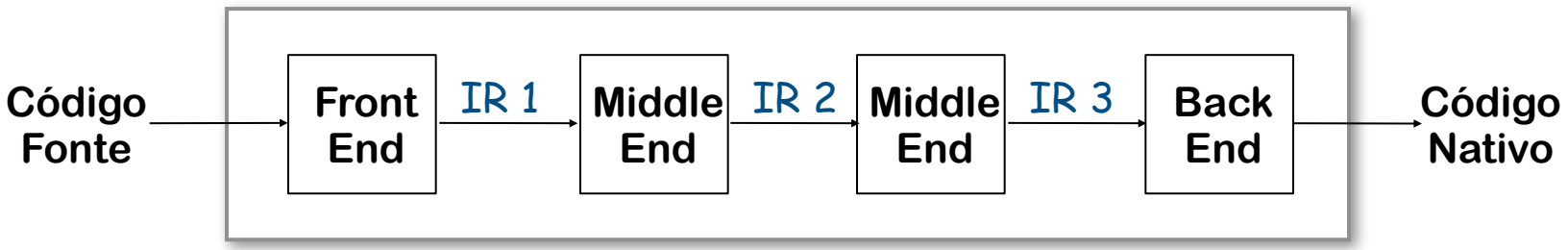
Forma SSA

```
      x0 ← …
      y0 ← …
      if (x0 >= k) goto next
loop:    x1 ← φ(x0,x2)
         y1 ← φ(y0,y2)
         x2 ← x1 + 1
         y2 ← y1 + x2
         if (x2 < k) goto loop
next:      …
```

Vantagens da forma SSA

- Análise mais precisa
- Algoritmos mais rápidos

Usando Múltiplas Representações



- Repetidamente reduzir o nível da representação intermediária
  - Cada representação é voltada para certas otimizações
- Exemplo: *GCC*
  - AST, GIMPLE, RTL
- Exemplo: V8
  - AST, Hydrogen, Lithium

O Resto...

Representar código é apenas parte de uma IR

Outros componentes necessários

- Tabela de Símbolos
- Tabela de constantes
  - Representação, tipo
  - Local de armazenamento, offset
- Mapa de armazenamento
  - Layout geral
  - Registradores virtuais

Tabelas de Símbolo com Escopo

- O problema
  - Compilador precisa de um registro para cada declaração
  - Escopo léxico aninhado admite múltiplas declarações
- A interface
  - insert(**nome**, **nível**): cria registro para **nome** em **nível**
  - lookup(**nome**): retorna registro para **nome**
- Muitas implementações foram propostas
- Vamos usar uma simples e que funciona bem para um compilador pequeno (poucos níveis léxicos, poucos nomes)

Tabelas de símbolos são estruturas em tempo de compilação para resolver referências para nomes.
Veremos as estruturas em tempo de execução correspondentes na geração de código.

Exemplo

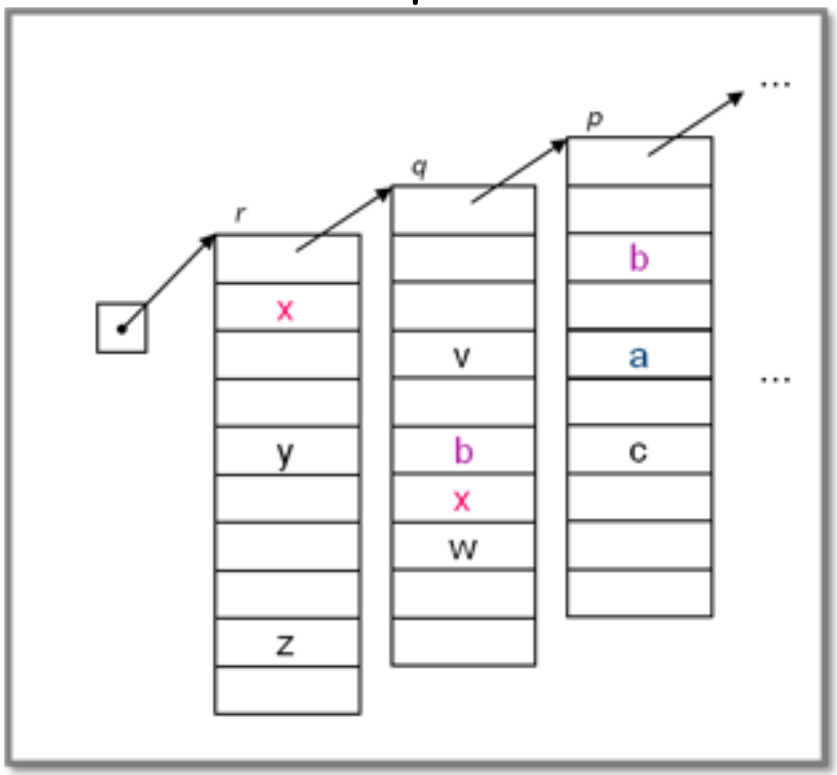
```
procedure p {
        int a, b, c
        procedure q {
            int v, b, x, w
            procedure  r {
                int x, y, z
                ....
            }
            procedure s {
                int x, a, v
                ...
            }
            ... r ... s
        }
        ... q ...
}
```

```
B0: {
        int a, b, c
B1:      {
            int v, b, x, w
B2:         {
                int x, y, z
                ....
            }
B3:         {
                int x, a, v
                ...
            }
            ...
        }
        ...
}
```

Tabelas de Símbolo com Escopo

Ideia geral

- Criar nova tabela para cada escopo
- Encadeá-las para busca



Implementação em "resma de tabelas"

- insert() pode precisar criar uma tabela
- sempre insere no nível corrente
- lookup() percorre cadeia de tabelas e retorna primeira ocorrência do nome

Se o compilador tem que preservar a tabela (para depuração, por exemplo), essa ideia é bastante prática.

Tabelas individuais são tabelas hash.