

MAB 471  
2012.1

# 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 significado



# 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 significado



# 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 significado



# 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 significado



# 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 significado



# 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 significado



# 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 significado





# 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 significado



# 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 significado



## 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 técnicas 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  $F_+$ ,  $F_-$ ,  $F_x$ , e  $F_{\div}$

| $F_x$  | 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  | <i>Goal</i>   | → | <i>Expr</i>          | $$$ = \$1;$                  |
| 2  | <i>Expr</i>   | → | <i>Expr + Term</i>   | $$$ = F_+(\$1, \$3);$        |
| 3  |               |   | <i>Expr - Term</i>   | $$$ = F_-(\$1, \$3);$        |
| 4  |               |   | <i>Term</i>          | $$$ = \$1;$                  |
| 5  | <i>Term</i>   | → | <i>Term * Factor</i> | $$$ = F_x(\$1, \$3);$        |
| 6  |               |   | <i>Term / Factor</i> | $$$ = F_{\div}(\$1, \$3);$   |
| 7  |               |   | <i>Factor</i>        | $$$ = \$1;$                  |
| 8  | <i>Factor</i> | → | <i>( Expr )</i>      | $$$ = \$2;$                  |
| 9  |               |   | <u>number</u>        | $$$ = \text{type of num};$   |
| 10 |               |   | <u>ident</u>         | $$$ = \text{type of ident};$ |





## Exemplo – Construção de AST

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

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



## Example – Emissão de IR linear

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

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



## 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

---

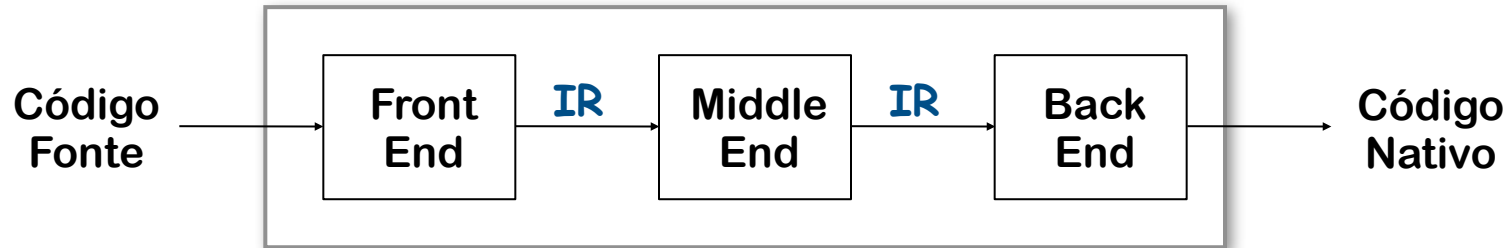
E se é preciso fazer ações que não se encaixam bem no framework da tradução dirigida por sintaxe?

- 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 apropriada é fundamental



# Tipos de Representações Intermediárias

---

Três grandes categorias

- Estrutural
  - Gráficas
  - Muito usada em tradução fonte para fonte
  - Tende a ser grande
- Linear
  - Pseudo-código para máquina abstrata
  - Nível de abstração varia
  - Simples e compacta
  - Mais fácil de rearrumar o código
- Híbrida
  - Combinação de grafos e código linear
  - Grafos de fluxo de controle

Exemplo:  
ASTs

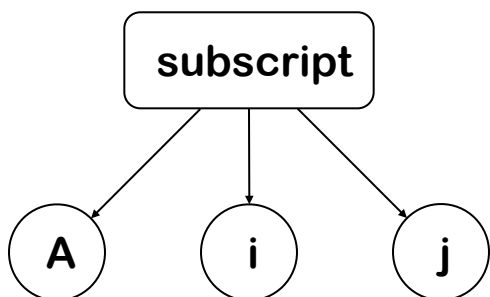
Exemplos:  
Código 3 endereços  
Código máq. de pilha

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

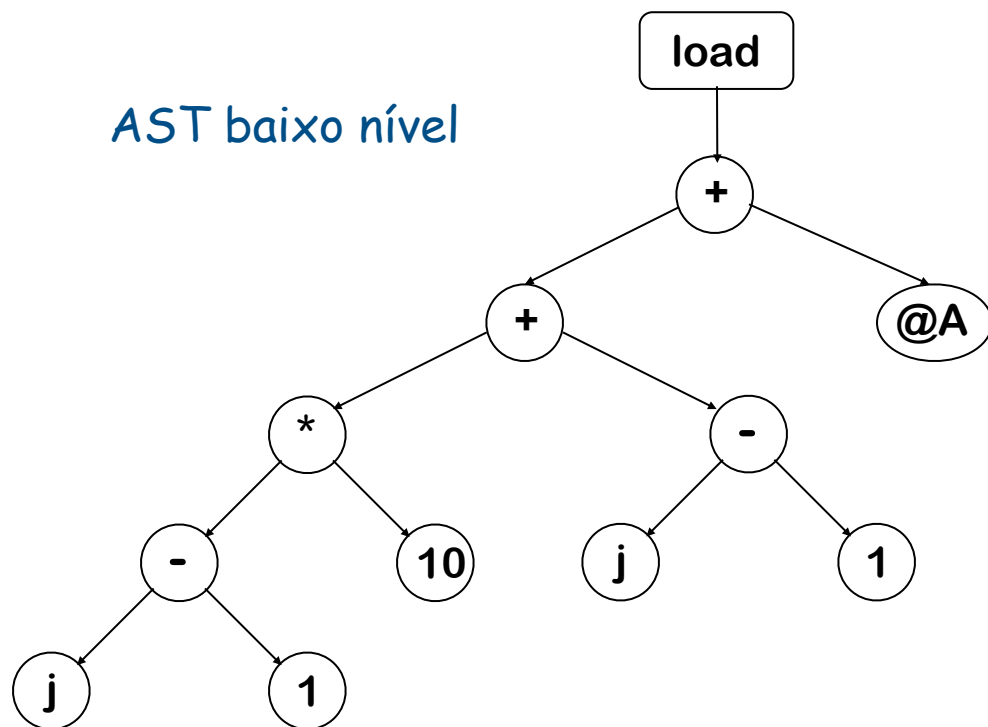
```
loadI 1      => r1
sub   rj, r1 => r2
loadI 10     => r3
mult  r2, r3 => r4
sub   ri, r1 => r5
add   r4, r5 => r6
loadI @A     => r7
add   r7, r6 => r8
load  r8     => rAij
```

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:



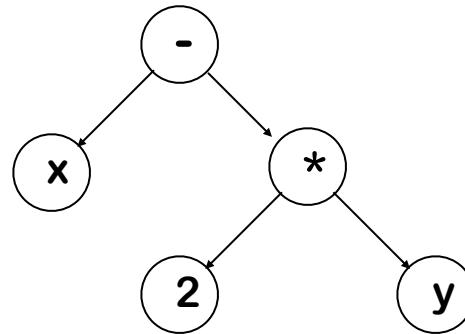
`loadArray A, i, j`

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:

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

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

Exemplo:

$$z \leftarrow x - 2 * y \quad \text{vira}$$

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:

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

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

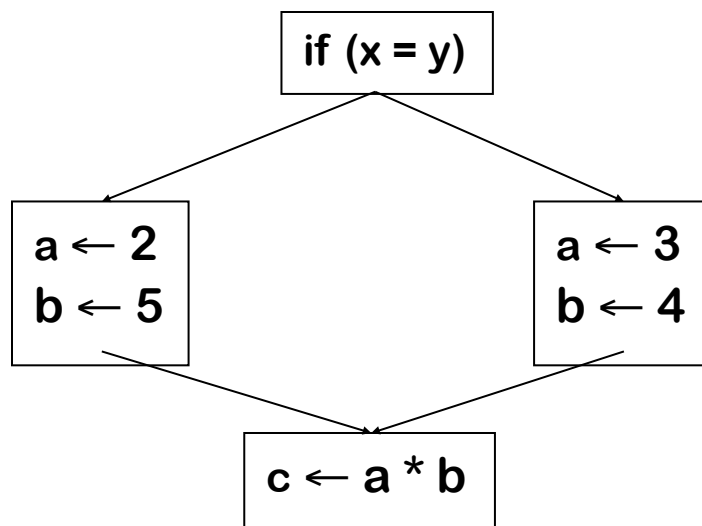


# Grafo de Fluxo de Controle

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  $\phi$  (seleção) para fazer funcionar

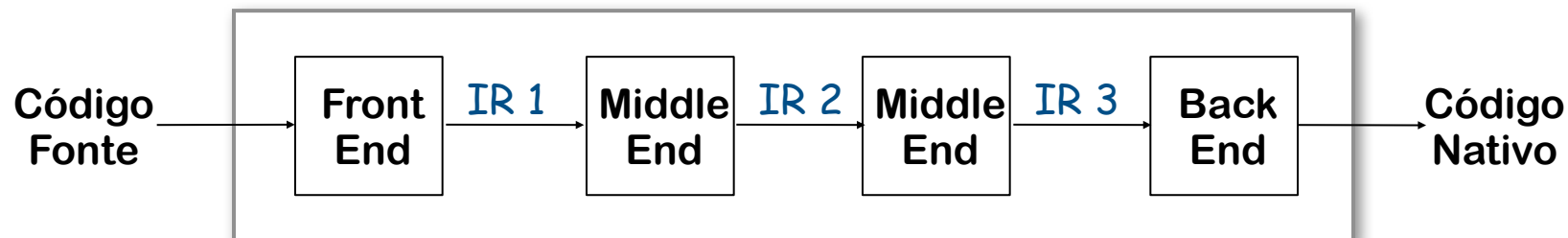
| Original  | Forma SSA   |
|---|---|
| <pre>x ← ...<br/>y ← ...<br/>while (x &lt; k)<br/>  x ← x + 1<br/>  y ← y + x</pre> | <pre>x<sub>0</sub> ← ...<br/>Y<sub>0</sub> ← ...<br/>if (x<sub>0</sub> ≥ k) goto next<br/>loop:  x<sub>1</sub> ← <math>\phi(x_0, x_2)</math><br/>      Y<sub>1</sub> ← <math>\phi(Y_0, Y_2)</math><br/>      x<sub>2</sub> ← x<sub>1</sub> + 1<br/>      Y<sub>2</sub> ← Y<sub>1</sub> + x<sub>2</sub><br/>      if (x<sub>2</sub> &lt; k) goto loop<br/>next:  ...</pre> |

## 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...

---

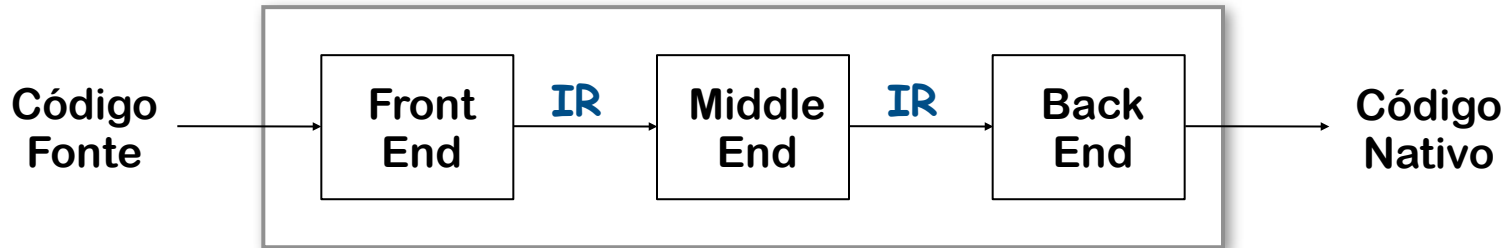
Representar código é apenas parte de uma IR

### Outros componentes necessários

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



# Análise Semântica



- Front end (analisador léxico + analisador sintático) produz uma árvore sintática abstrata
- Middle end faz a análise semântica para checar a validade da árvore e prepará-la com informação necessária para geração de código
- Back end faz a geração de código final





# Informação Semântica

---

- Tabela de Símbolos
  - Normalmente se usa várias, para os diferentes espaços de nomes e escopos
  - Mapeia nomes em todo o programa para elementos reais da linguagem: variáveis, funções, classes, métodos
- Checagem de Tipos
  - Análise estática dos tipos usados em todas as expressões e comandos do programa
  - Informação sobre o tipo dos elementos do programa fica na tabela de símbolos



# Projetando a Análise Semântica

---

- Regras de escopo
  - Escopo aninhado, redefinição de variáveis, separação dos espaços de nomes...
  - Afeta a estrutura das tabelas de símbolos
  - Quando a amarração dos nomes com os elementos reais é feita?
- Regras de Tipagem
  - Tipos existentes
  - Tipos permitidos em expressões e comandos
  - Tipos definidos pelo usuário
  - Equivalência de tipos, nominal ou estrutural



# Metodologia

---

- Percorrer e anotar a AST em múltiplas passadas
- Cada passada fica responsável por uma verificação semântica, e pode fornecer informação para passadas subsequentes
- Implementação pode ser simples, com um método nas classes da AST para cada passada, ou podemos fazer um esquema mais elaborado com o padrão Visitor
- Em geral todas as passadas percorrem a árvore em profundidade, com ações específicas sendo executadas em pré-ordem, ordem, ou pós-ordem



# 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 ...  
}
```

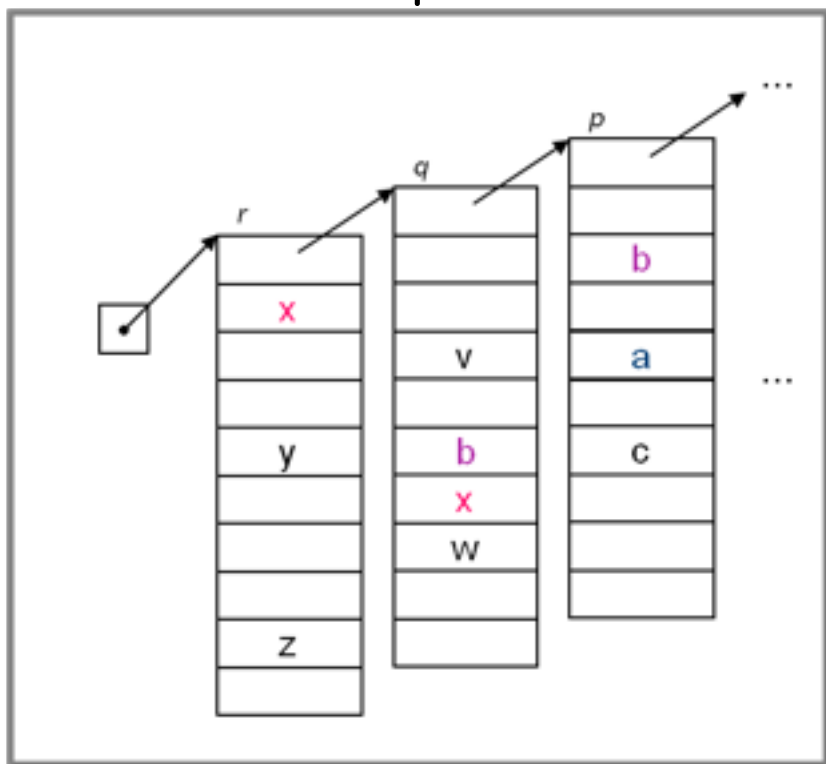
```
B0: {  
  int a, b, c  
B1: {  
  int v, b, x, w  
B2: {  
  int x, y, z  
  ....  
  }  
B3: {  
  int x, a, v  
  ...  
  }  
  ...  
}
```



# 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.



# Considerações sobre Escopo

---

- Escopo dos parâmetros da função é separado do corpo da função?
- Variáveis podem ser redefinidas no mesmo escopo? E em escopos aninhados?
  - Se puderem ser redefinidas no mesmo escopo a estrutura de tabela de símbolo dos slides anteriores não serve
- Variáveis e funções podem ter o mesmo nome? E quanto a variáveis e nomes de classes? E métodos? E tipos definidos pelo usuário? E quanto a espaços de nomes definidos pelo usuário (pacotes Java, namespaces C++)?
  - Em geral cada espaço de nomes deve ter sua própria tabela de símbolos ou pilha de tabelas de símbolo



# Checagem de Tipos

---

- Checagem estática de tipos é a maior parte do analisador semântico
- Verifica o uso consistente dos tipos
  - Operandos
  - Lado direito com lado esquerdo de atribuições
  - Interfaces de funções
  - Membros de estruturas
- Todo termo da linguagem envolve tipos de alguma forma
- Várias linguagens adiam essa verificação para a execução do programa: tipagem dinâmica
  - Requer geração de código e uma ambiente de execução mais elaborados





# Tipos Simples

---

- Tipos primitivos da linguagem
  - Pré-definidos pela especificação
  - inteiros, caracteres, números de ponto flutuante, booleanos, o tipo void
  - Detalhes como tamanho podem ser parte da especificação (Java), ou ficar em aberto para o implementador (C)
- Linguagem também define regras de compatibilidade entre esses tipos
  - C e Java têm vários tamanhos de inteiros, e regras de conversão entre os diferentes tamanhos
  - O mesmo vale para inteiros e números de ponto flutuante
  - Qual o tipo de  $x+y$  se  $x$  é inteiro e  $y$  é p.f.? Existem linguagens em que essa operação é inválida, e outras em que  $x+y$  é sempre adição inteira e existe outro operador para adição de p.f.



# Construtores de Tipos

---

- Mecanismos na linguagem para criação de novos tipos
  - Enumerações, vetores, matrizes, ponteiros, estruturas, uniões, funções, classes...
  - Tipos novos podem ter nomes próprios (classes em Java, typedef de C) ou serem anônimos
- Compilador precisa representar a estrutura de cada tipo
  - Os tipos normalmente formam sua própria mini-linguagem com sua AST
- Como os tipos são comparados? Pode ser por nome (Java) ou pela estrutura (C)



# Vetores e Matrizes

---

- Tipos de vetor formados a partir de um tipo base
- O tamanho do vetor pode fazer parte do tipo ou não
- Equivalência entre tipos de vetores
  - Pode-se usar variáveis de tipo vetor em uma atribuição? O vetor é copiado ou é criado um alias?
  - Vetores com tamanhos diferentes são compatíveis?
  - E quanto a tipos base diferentes (vetores em Java)
- Como funcionam as matrizes? São vetores de vetores, ou uma estrutura contígua?



# Estruturas (structs)

---

- Generalização de vetores, com uma coleção de valores de tipos diferentes
  - Os nomes dos campos são convertidos para offsets no código gerado
- Qual o espaço de nomes dos campos? Cada tipo de estrutura tem o seu?
- Equivalência de estruturas
  - Nominal (como nas classes Java) ou estrutural, como em ML?
  - Na equivalência estrutural dois tipos struct são o mesmo tipo se têm campos equivalentes na mesma ordem
  - Algoritmo recursivo



# Tipos Recursivos

---

- Uma estrutura ou classe pode ter um campo com o mesmo tipo dela?
  - Em C e C++ não, mas pode ter um ponteiro para ela
  - Em Java sim
- Tipos recursivos complicam algoritmos de equivalência estrutural, mas são triviais em equivalência nominal
- Implementação de tipos recursivos sempre usa algum tipo de indireção (ponteiros), seja explicitamente, como em C, ou implicitamente, como em Java



# Funções

---

- Tipos específicos para funções só são necessários quando se tem ponteiros de função e mecanismos similares (funções como valores)
- Na prática tipos de função são equivalentes a tipos de estrutura, com problemas similares
- Outros aspectos importantes são a amarração de funções e qual o espaço de nomes das funções



# Equivalência de Tipos em C

---

- Mistura de equivalência estrutural e nominal
  - Structs e unions usam equivalência nominal módulo typedefs
  - Vetores, ponteiros e funções usam equivalência estrutural
  - Várias regras de coerção entre tipos primitivos
  - Structs "anônimas" têm um nome definido pelo compilador
  - O código a seguir não compila com erro de tipo na atribuição

```
struct { int a; } foo;  
struct { int a; } bar;  
foo.a = 2;  
bar = foo;
```



# TINY Tipado

---

- Vamos acrescentar declarações de variáveis e tipos a TINY, seguindo a seguinte gramática:

```
var-decl : var id ':' tipo
tipo : tipo-simp | tipo-estrut
tipo-simp : int | bool | char
tipo-estrut : array '[' num '] of tipo
              | record campos end
              | '^' tipo
campos : id ':' tipo { ';' id ':' tipo }
```

- A verificação de tipos é estrutural, e não é permitido atribuições envolvendo arrays e records
- O tamanho do array não conta para equivalência