

Representação Intermediária

Geração de IR

Hervé Yviquel

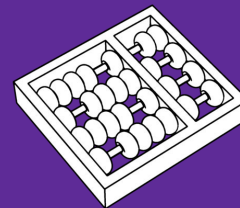
herve@ic.unicamp.br

Universidade Estadual de Campinas (Unicamp)

Instituto de Computação (IC)

Laboratório de Sistemas de Computação (LSC)

MC921 • Projeto e Construção de Compiladores • 2023 S2



UNICAMP

Aulas Anteriores

Resumo

- Overview
- Lexer
- Parser
- Análise semântica

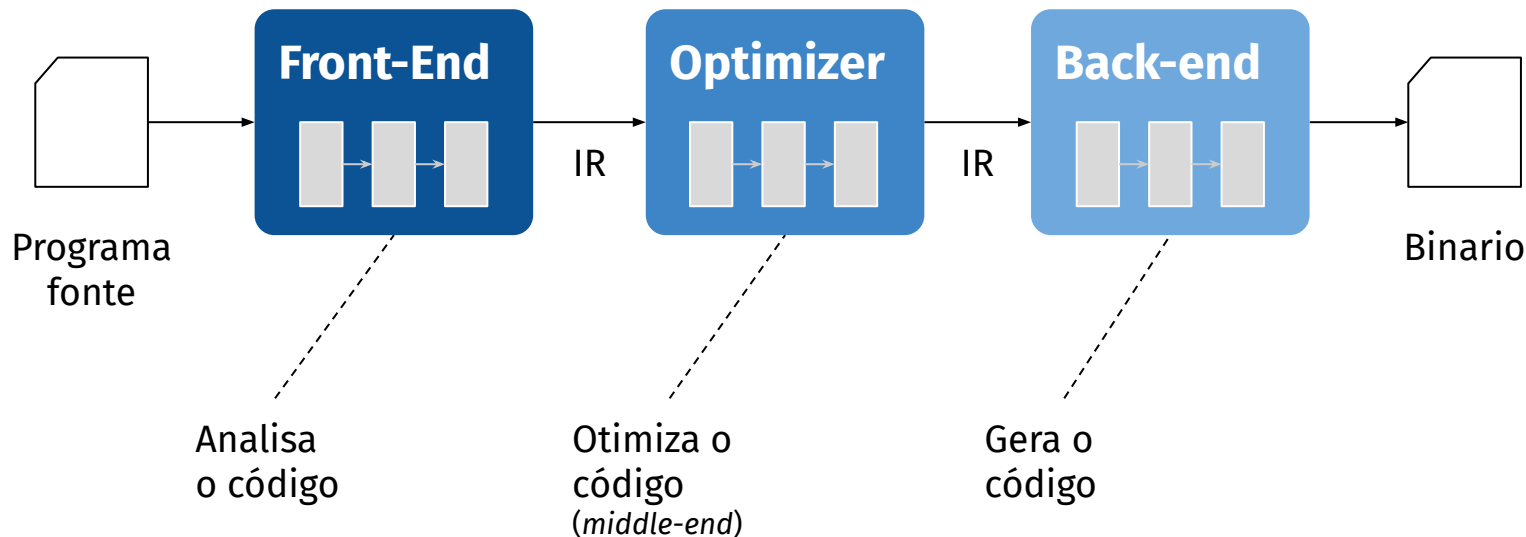
Aula de Hoje

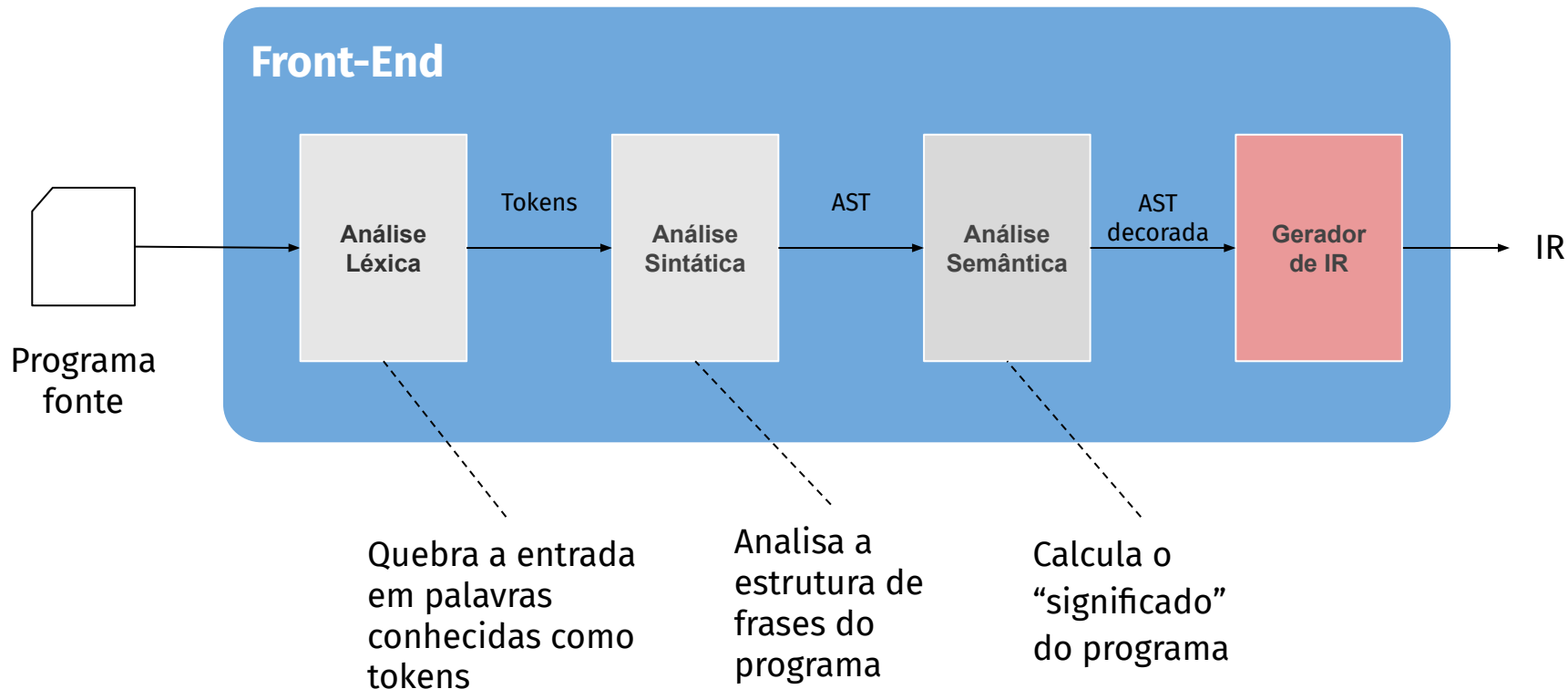
Plano

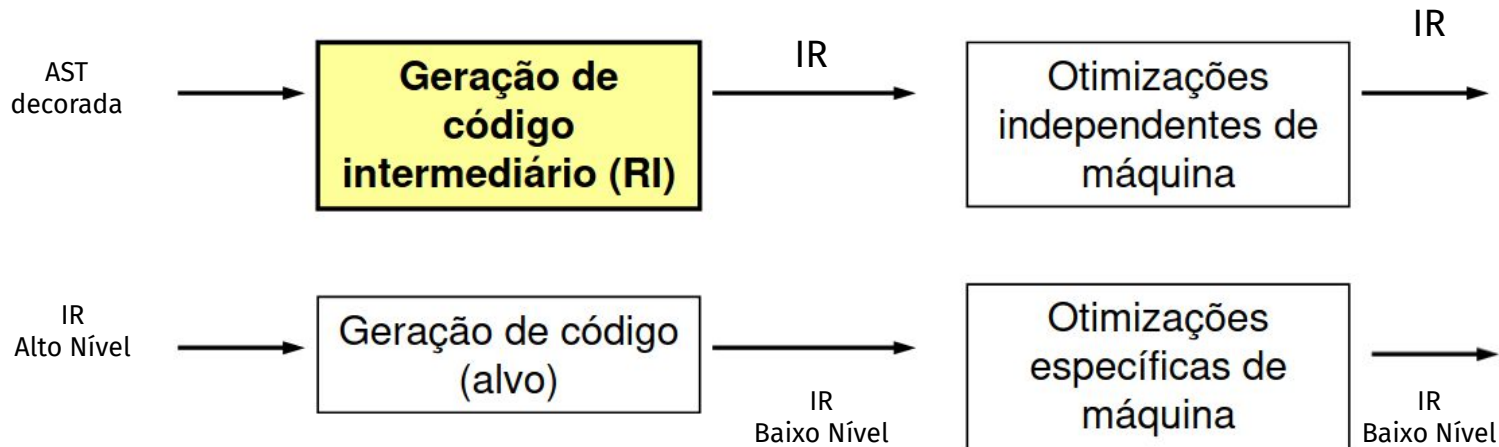
- Visão Geral do Front-end
- Representação Intermediária
- Tipos de IRs
- IRs de Compiladores
- Exemplo de IR

Visão Geral do Front-end





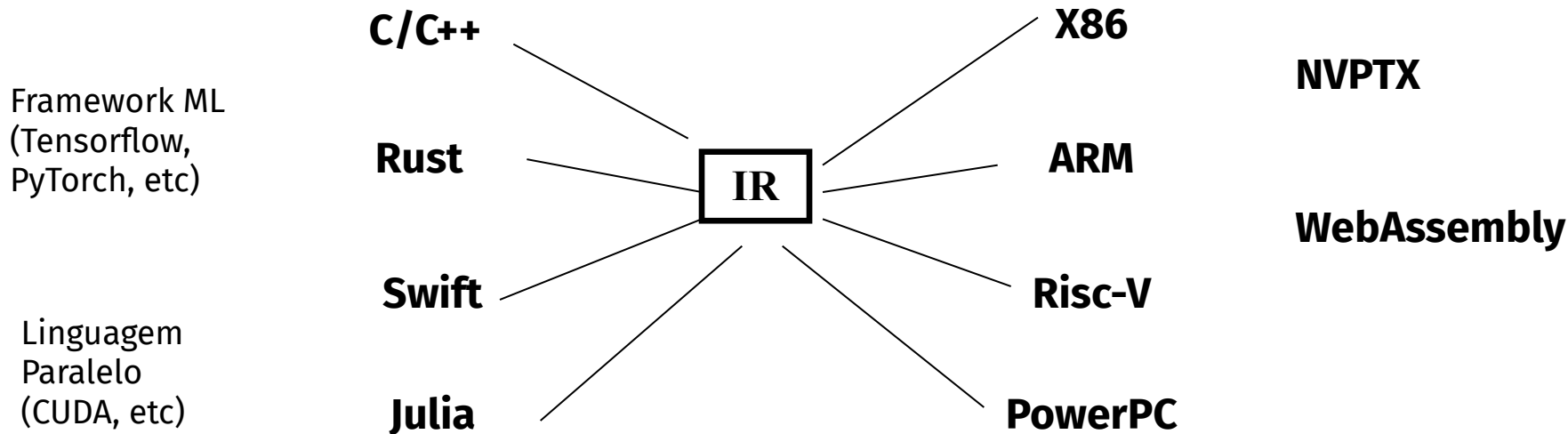




Representação Intermediária



- Queremos compiladores para N linguagens
 - direcionados para M máquinas diferentes.
- IR nos possibilita elaborar N front-ends e M back-ends
 - ao invés de N.M compiladores



- Facilmente construída a partir da análise semântica
- Conveniente para a tradução para linguagem de máquina
- Facilmente alterada (re-escrita) durante as transformações (otimizações) de código
- Adequação à linguagem e à arquitetura alvo
- O custo de gerar e manipular a IR
- Nível, estrutura, expressividade da IR

Steven Muchnick

“Intermediate-language design is largely an art, not a science”.

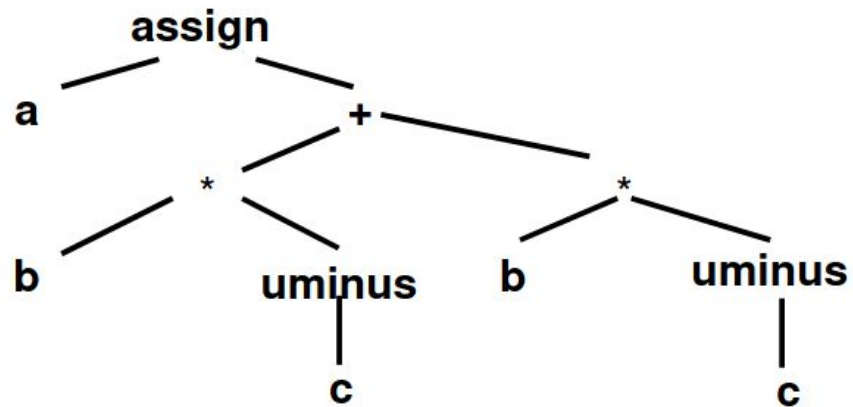
Tipos de IRs



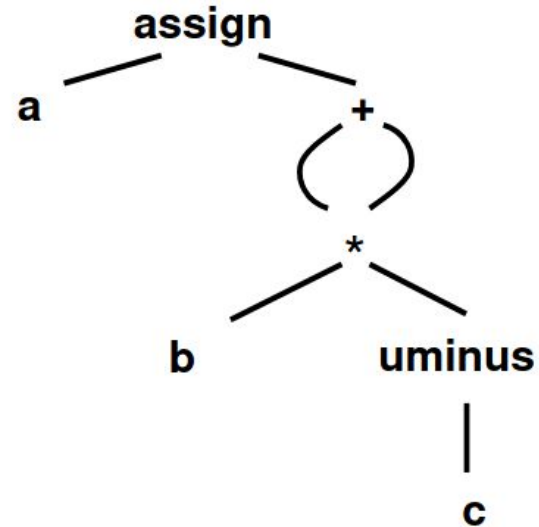
1. IR gráfica
 - Árvore ou Grafo
 2. IR linear
 - Pseudo-código
 3. IR híbrida
 - mistura de gráfica e linear
- Alto nível
 - acesso a arrays
 - chamada de funções
 - Nível Médio
 - composta por descrições de operações simples
 - busca/armazenamento de valores, soma, movimentação, saltos, etc

- Representação gráfica
 - Árvores ou Grafos acíclicos dirigidos
- Manipulação pode ser feita através de gramáticas
- Pode ser tanto de alto-nível como nível médio
- Usada tipicamente para representar fluxo computação (ex. $a := b + c$) e não de controle (ex. if statement)
- Construção
 - Um DAG é associado a cada expressão
 - Nós internos do DAG são operadores
 - Nós folha representam operandos: variáveis e constantes
 - Nós possuem rótulos que representam o último valor computado para aquele nó

$a := b * -c + b * -c$



Árvore

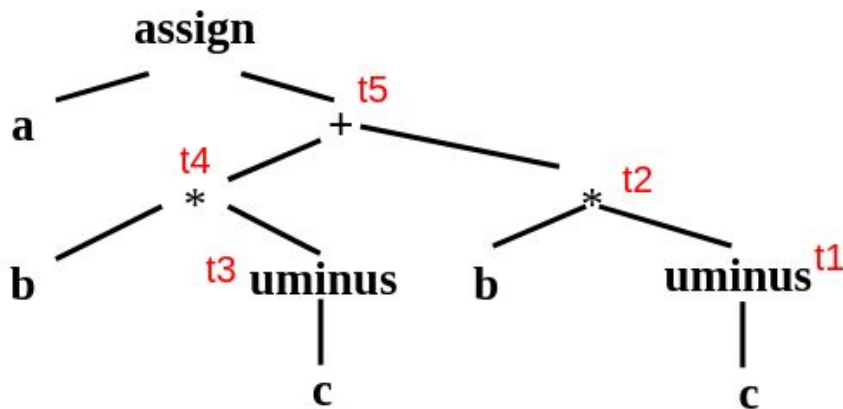


DAG

- Detectar sub-expressões comuns
 - automaticamente durante a construção (t1 no exemplo a seguir)
- Detectar os identificadores cujos valores são usados no bloco
 - São as folhas (a, b, c)
- Detectar sentenças que geram valores que podem ser usados fora do bloco
 - São aquelas sentenças que geraram $\text{nó}(x) = n$ durante a construção e ainda temos $\text{nó}(x) = n$ ao final, onde x é uma raiz (a1 no exemplo a seguir)

- IRs lineares se assemelham a um pseudo-código para alguma máquina abstrata
 - Bytecode do Java (interpretado ou traduzido)
 - GCC RTL
 - LLVM IR
 - Código de três endereços

- Forma geral: $x := y \text{ op } z$
- Representação linearizada de uma árvore sintática, ou DAG



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

$a := b * -c + b * -c$

- *Three Address Code*
 - Abreviado TAC ou 3AC
- Tipos de sentenças:
 - **atribuição:** $x := y \text{ op } z$ ou $x := y$ ou $x := \text{op } y$
 - **saltos:** goto L
 - **desvios condicionais:** if x relop y goto L
 - **chamadas a procedimentos:** param x and call p,n
 - **retorno:** return y
 - **arrays:** $x := y[i]$ ou $x[i] := y$

```
{  
  ...  
  prod = 0;  
  i = 1;  
  while (i ≤ 20) {  
    prod = prod + a[i] * b[i];  
    i = i + 1;  
  }  
  ...  
}
```

```
(1)  prod := 0  
(2)  i := 1  
(3)  t1 := 4 * i  
(4)  t2 := a[t1]  
(5)  t3 := 4 * i  
(6)  t4 := b[t3]  
(7)  t5 := t2 * t4  
(8)  t6 := prod + t5  
(9)  prod := t6  
(10) t7 := i + 1  
(11) i := t7  
(12) if i ≤ 20 goto (3)
```


- **Quádruplas** – (op, arg1, arg2, result)
 - (1) \rightarrow (*, b, t1, t2)
- **Triplas** –
 - (0) \rightarrow (-, c,)
 - (1) \rightarrow (*, b, (0))
 - (2) \rightarrow (-,c,)

a := b * -c + b * - c

(0)	t1	:=	- c
(1)	t2	:=	b * t1
(2)	t3	:=	- c
(3)	t4	:=	b * t3
(4)	t5	:=	t2 + t4
(5)	a	:=	t5

- Static Single-Assignment form (SSA)
 - É uma propriedade da IR
- As variáveis são atribuídas uma única vez

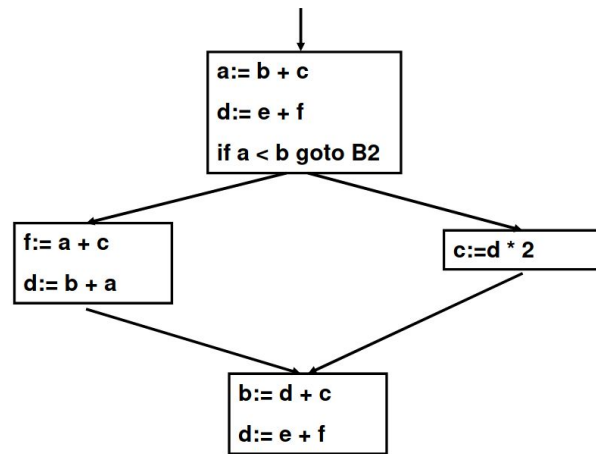
```
a = 10 + 20;  
b = 2 * a;  
a = a + 1;
```



```
%1 = 10  
%2 = 20  
a_1 = %1 + %2  
%4 = 2  
b_1 = %4 * a_1  
%5 = 1  
a_2 = a_1 + %5  
...
```

- Combina elementos
 - das IRs gráficas (estrutura)
 - das IRs lineares
- Usa IR linear para blocos de código sequencial
 - Bloco básico (*basic block* ou BB)
- Usa uma representação gráfica
 - *Control Flow Graph* (CFG)
 - para representar o fluxo de controle entre esses blocos

- Representação gráfica do código de 3 endereços
 - Nós -> Computação
 - Blocos básicos
 - Arestas -> Fluxo de controle
 - if/else, while/for, break, etc)
- Muito usado
 - em coletas de informações sobre o programa (Análise estática)
 - para algoritmos de otimização de código



- Sequência de instruções consecutivas
- Fluxo de Controle
 - Entra no início
 - Sai pelo final
 - Não existem saltos para dentro ou do meio para fora da sequência

t1 = a * a

t2 = a * b

t3 = b * 3

t4 = t2 - t3

- Entrada
 - Sequência de código 3-endereços
- Defina os líderes (iniciam os BBs)
 - Primeira Sentença é um líder
 - Toda sentença que sucede imediatamente um goto, condicional ou incondicional, é um líder
 - Todo alvo de um goto, condicional ou incondicional, é um líder
- Os BBs são compostos pelos líderes e todas as instruções subsequentes até o próximo líder (exclusive)

Exemplo: Quicksort



```
void quicksort(m,n)
int m,n;
{
    int i,j;
    int v,x;
    if ( n <= m ) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while(1) {
        do i = i+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if (i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Fig. 10.2. C code for quicksort.

```
(1)  i := m-1
(2)  j := n
(3)  t1 := 4*n
(4)  v := a[t1]


---


(5)  i := i+1
(6)  t2 := 4*i
(7)  t3 := a[t2]
(8)  if t3 < v goto (5)


---


(9)  j := j-1
(10) t4 := 4*j
(11) t5 := a[t4]
(12) if t5 > v goto (9)


---


(13) if i >= j goto (23)


---


(14) t6 := 4*i
(15) x := a[t6]
```

```
(16) t7 := 4*i
(17) t8 := 4*j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4*j
(21) a[t10] := x
(22) goto (5)


---


(23) t11 := 4*i
(24) x := a[t11]
(25) t12 := 4*i
(26) t13 := 4*n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4*n
(30) a[t15] := x
```

Fig. 10.4. Three-address code for fragment in Fig. 10.2.

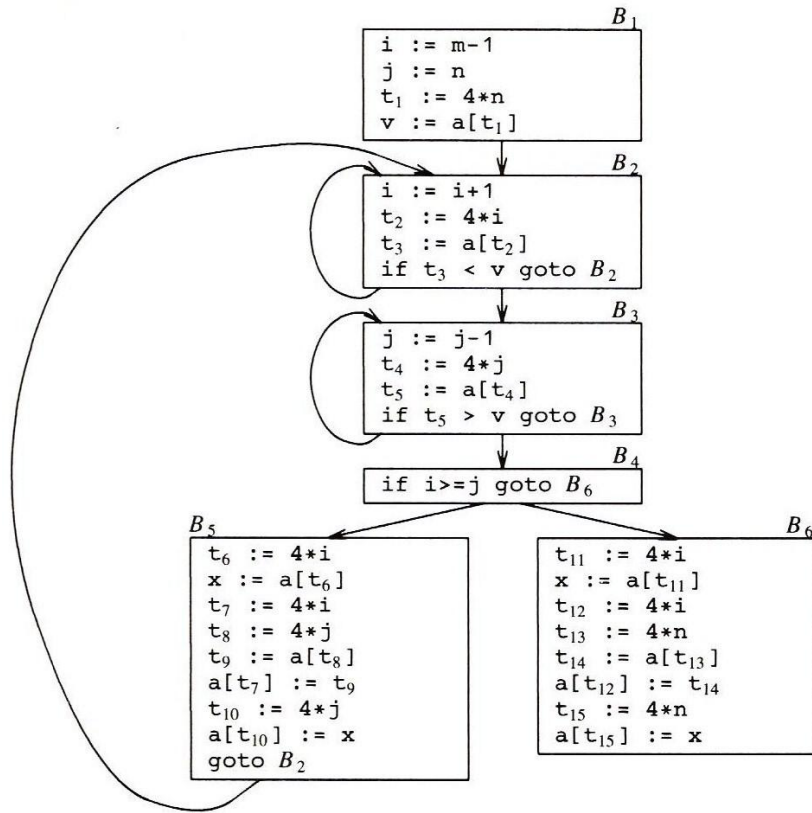


Fig. 10.5. Flow graph.

Exemplos de IR



d := (a+b)*c

```
(insn 8 6 10 (set (reg:SI 2)
                  (mem:SI (symbol_ref:SI ("a")))))
(insn 10 8 12 (set (reg:SI 3)
                  (mem:SI (symbol_ref:SI ("b")))))
(insn 12 10 14 (set (reg:SI 2)
                   (plus:SI (reg:SI 2) (reg:SI 3))))
(insn 14 12 15 (set (reg:SI 3)
                   (mem:SI (symbol_ref:SI ("c")))))
(insn 15 14 17 (set (reg:SI 2)
                   (mult:SI (reg:SI 2) (reg:SI 3))))
(insn 17 15 19 (set (mem:SI (symbol_ref:SI ("d")))
                   (reg:SI 2)))
```

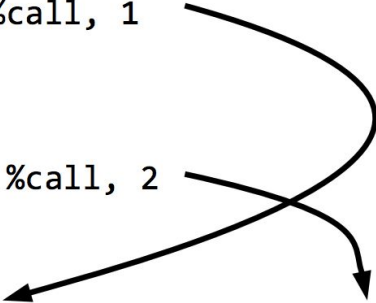
```
int ssa2() {  
    int y, z;  
    y = f();  
    if (y < 0)  
        z = y + 1;  
    else  
        z = y + 2;  
    return z;  
}
```

```
define i32 @ssa2() nounwind {  
entry:  
    %call = call i32 @f()  
    %cmp = icmp slt i32 %call, 0  
    br i1 %cmp, label %if.then, label %if.else
```

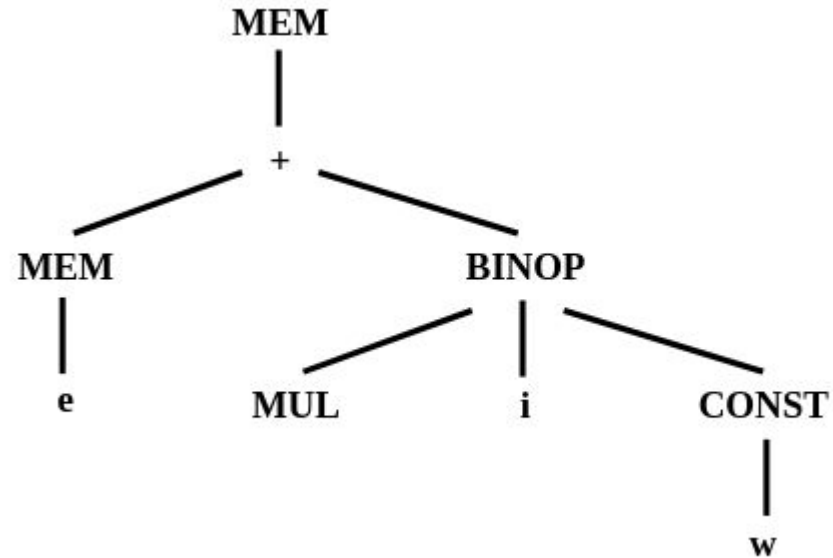
```
if.then:  
    %add = add nsw i32 %call, 1  
    br label %if.end
```

```
if.else:  
    %add1 = add nsw i32 %call, 2  
    br label %if.end
```

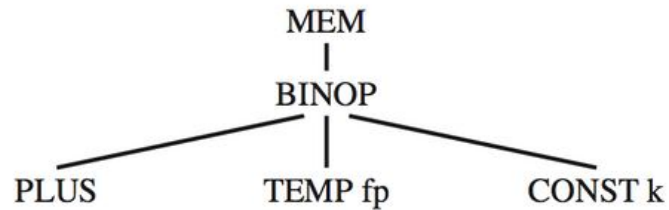
```
if.end:  
    %z.0 = phi i32 [ %add, %if.then ], [ %add1, %if.else ]  
    ret i32 %z.0  
}
```



- Tipos de operações (nós): const, binop, mem, call, etc
 - Exemplo: `a[i]`



- Acesso a variáveis:



$\text{MEM}(\text{BINOP}(\text{PLUS}, \text{TEMP } fp, \text{CONST } k))$

- if $x < 5$

$\text{CJUMP}(\text{LT}, x, \text{CONST}(5), t, f)$

uC IR



- Código 3 endereços
- Single State Assignment (SSA)
- Inspirado do LLVM IR
- Particularidade
 - Todos os acessos a variáveis são feitos com load/store

<code>('alloc_type', varname)</code>	<code># Allocate on stack (ref by register) a variable of a given type.</code>
<code>('global_type', varname, value)</code>	<code># Allocate on heap a global var of a given type. value is optional.</code>
<code>('load_type', varname, target)</code>	<code># Load the value of a variable (stack/heap) into target (register).</code>
<code>('store_type', source, target)</code>	<code># Store the source/register into target/varname.</code>
<code>('literal_type', value, target)</code>	<code># Load a literal value into target.</code>
<code>('elem_type', source, index, target)</code>	<code># Load into target the address of source (array) indexed by index.</code>
<code>('get_type', source, target)</code>	<code># Store into target the address of source.</code>

```
('add_type', left, right, target)    # target = left + right
('sub_type', left, right, target)    # target = left - right
('mul_type', left, right, target)    # target = left * right
('div_type', left, right, target)    # target = left / right (integer truncation)
('mod_type', left, right, target)    # target = left % right
```

```
('not_type', expr, target)          # target = !expr
```

```
('oper_type', left, right, target)  # target = left `oper` right, where `oper` is:
                                     #          lt, le, ge, gt, eq, ne, and, or, not
```

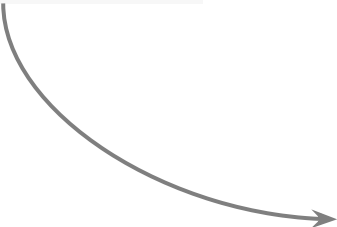
```
('label:', ) # Label definition  
( 'jump', target) # Jump to a target label  
( 'cbranch', expr_test, true_target, false_target) # Conditional Branch
```

```
('define_type', source, args)    # Function definition. Source=function label, args=list of pairs  
                                  # (type, name) of formal arguments.  
( 'call_type', source, target)   # Call a function. target is an optional return value  
( 'return_type', target)         # Return from function. target is an optional return value  
( 'param_type', source)         # source is an actual parameter  
( 'read_type', source)          # Read value to source  
( 'print_type', source)         # Print value of source
```



```
int n = 10;
```

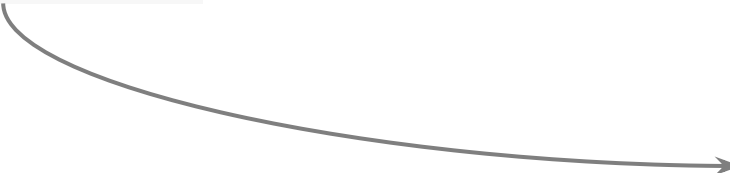
```
int foo(int a, int b) {  
    return n * (a + b);  
}
```



```
('global_int', '@n', 10)  
('define_int', '@foo', [('int', '%1'), ('int', '%2')])  
# function arguments: the value for "a" is passed in register %1, for "b" in register %2  
# & register %3 is reserved to hold the return value (note that %0 is reserved by the Interpreter)  
('entry:',)  
('alloc_int', '%3')  
('alloc_int', '%a')  
('alloc_int', '%b')  
('store_int', '%1', '%a')  
('store_int', '%2', '%b')  
('load_int', '%a', '%4')  
('load_int', '%b', '%5')  
('add_int', '%4', '%5', '%6')  
('load_int', '@n', '%7')  
('mul_int', '%7', '%6', '%8')  
('store_int', '%8', '%3')  
('jump', '%exit')  
('exit:',)  
('load_int', '%3', '%9')  
('return_int', '%9')
```

Exemplo com Pretty Print

```
int n = 10;  
  
int foo(int a, int b) {  
    return n * (a + b);  
}
```



```
@n = global int 10  
  
define int @foo (int %1, int %2)  
entry:  
    %3 = alloc int  
    %a = alloc int  
    %b = alloc int  
    store int %1 %a  
    store int %2 %b  
    %4 = load int %a  
    %5 = load int %b  
    %6 = add int %4 %5  
    %7 = load int @n  
    %8 = mul int %7 %6  
    store int %8 %3  
    jump label %exit  
exit:  
    %9 = load int %3  
    return int %9
```

O que faz esse código?

```
@n = global int 3
@.str.0 = global string 'assertion_fail on 10:12'

define int @function (int %1)
entry:
    %2 = alloc int
    %x = alloc int
    store int %1 %x
    %3 = load int %x
    %4 = load int %x
    %5 = mul int %3 %4
    store int %5 %2
    jump label %exit
exit:
    %6 = load int %2
    return int %6
```

```
define void @main ()
entry:
    %v = alloc int
    %1 = load int @n
    store int %1 %v
    %2 = load int %v
    param int %2
    %3 = call int @function
    store int %3 %v
    %4 = load int @n
    %5 = load int @n
    %6 = mul int %4 %5
    %7 = load int %v
    %8 = eq int %7 %6
    cbranch %8 label %assert.true label %assert.false
assert.false:
    print string @.str.0
    jump label %exit
assert.true:
    jump label %exit
exit:
    return
```

Resumo

- Visão Geral do Front-end
- Representação Intermediária
- Tipos de IRs
- Exemplo
- IRs de Compiladores
- uC IR

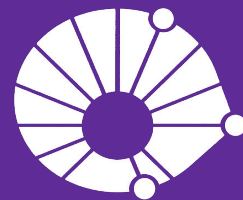
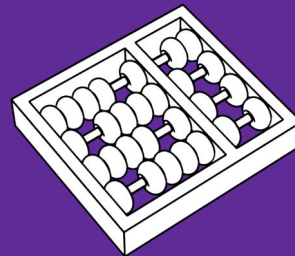
Leitura Recomendada

- Capítulo 8 do livro do Aho
- Capítulo 7 e 8 do livro do Appel

Próxima Aula

- Seleção de Instruções

Obrigado!
Merci!



UNICAMP

Pallette



BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

DRACULA

Table Title	
Column 1	Column 2
One	Two
Three	Four

Table Title	
Column 1	Column 2
One	Two
Three	Four

Table Title	
Column 1	Column 2
One	Two
Three	Four

Table Title	
Column 1	Column 2
One	Two
Three	Four

