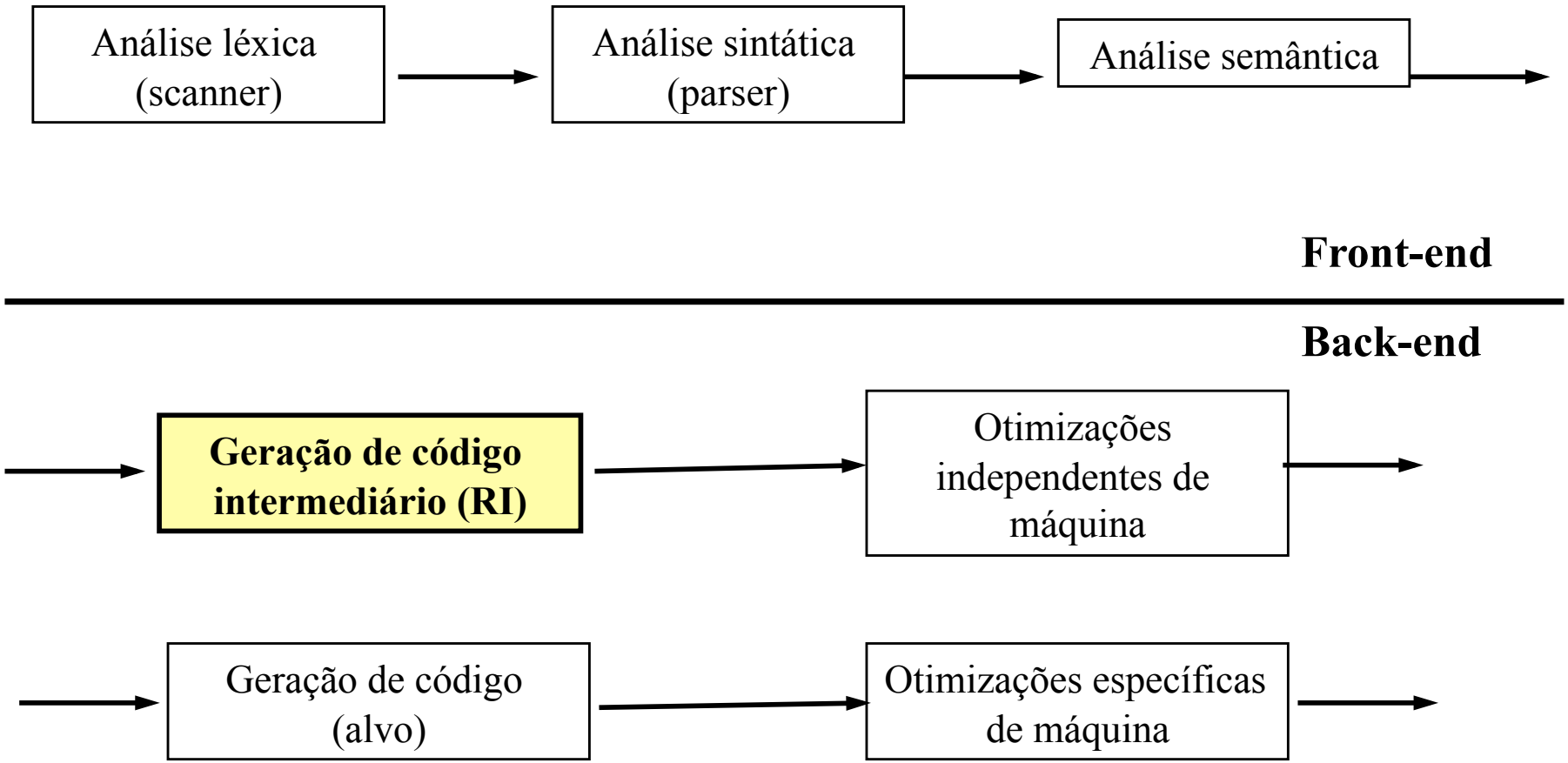
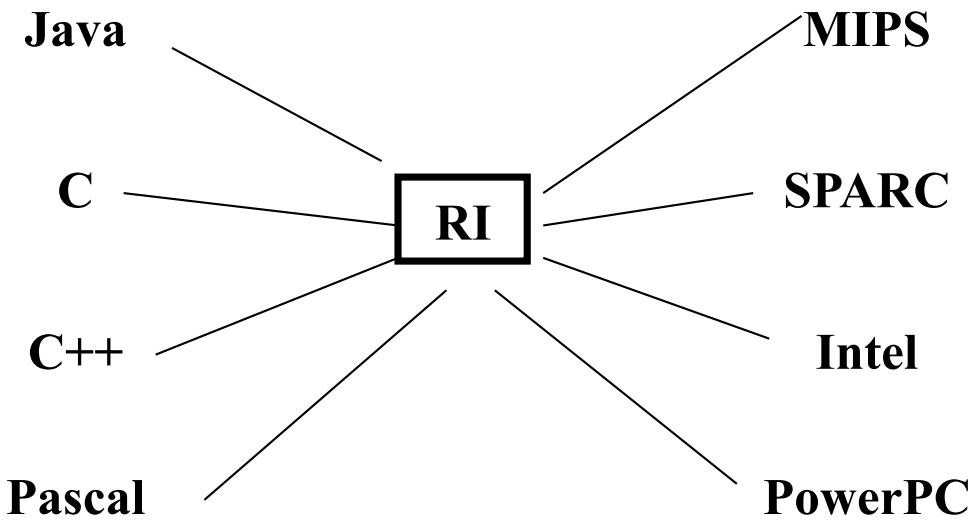
Representação Intermediária

Guido Araújo
guido@ic.unicamp.br

Representação Intermediária (RI)



Representação Intermediária (RI)



Queremos compiladores para N linguagens, direcionados para M máquinas diferentes.

RI nos possibilita elaborar N front-ends e M back-ends, ao invéz de N.M compiladores.

Requisitos para uma RI

- 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

Escolha de uma RI

- Reuso: adequação à linguagem e à arquitetura alvo, custos.
- Projeto: nível, estrutura, expressividade
- “Intermediate-language design is largely an art, not a science”, Steven Muchnick
- Pode-se adotar mais de uma RI

Características de uma RI

- Alto nível: acesso a arrays, chamada de procedimentos
- Nível Médio: composta por descrições de operações simples: busca/armazenamento de valores, soma, movimentação, saltos, etc.

Tipos de RI

- Representação gráfica:
  - Árvores ou DAGs
  - Manipulação pode ser feita através de gramáticas
  - Pode ser tanto de alto-nível como nível médio

Directed Acyclic Graphs

- Construção:
  - Folhas são identificadores únicos
    - variáveis, constantes
    - são os valores iniciais das variáveis
    - usa-se índices para não confundir com valor atual

Directed Acyclic Graphs

- Construção:
  - Nós internos são operadores:
    - valores computados
    - O rótulo é o operador associado
    - podem ter uma lista de variáveis associados
      - é o último valor computado para cada uma delas
    - um nó associado a cada sentença
    - filhos representam a última definição dos operandos

Exemplo - DAGs

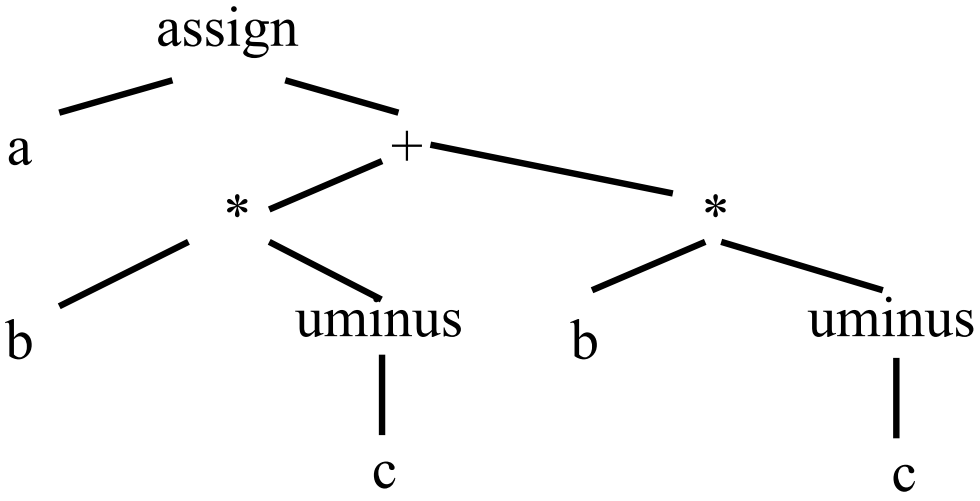
```
(1)t1 := 4 * i
(2)t2 := a [ t1]
(3)t3 := 4 * i
(4)t4 := b [t3]
(5)t5 := t2 * t4
(6)t6 := prod + t5
(7)prod := t6
(8) t7 := i + 1
(9) i := t7
(10) if i <= 20 goto (1)
```

Aplicações

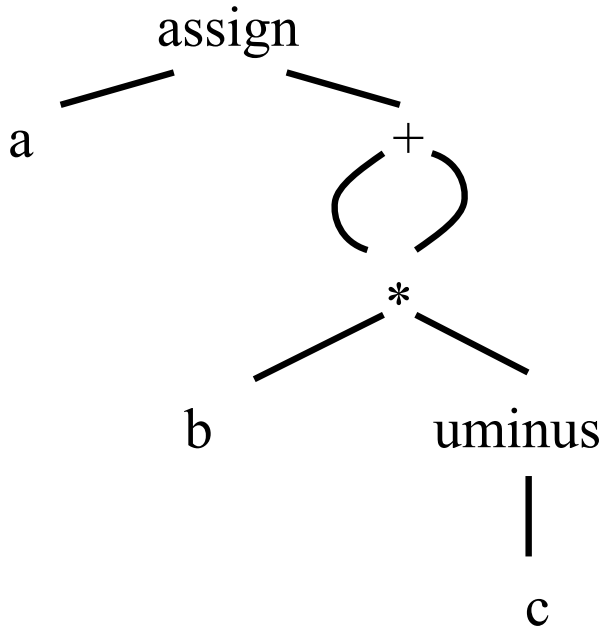
- Detectar sub-expressões comuns
  - automaticamente durante a construção
- Detectar os identificadores cujos valores são usados no bloco
  - São as folhas
- Detectar sentenças que geram valores que podem ser usados fora do bloco
  - São aquelas sentenças que geraram nó(x) = n durante a construção e ainda temos nó(x) =n ao final

Árvore vs DAG

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



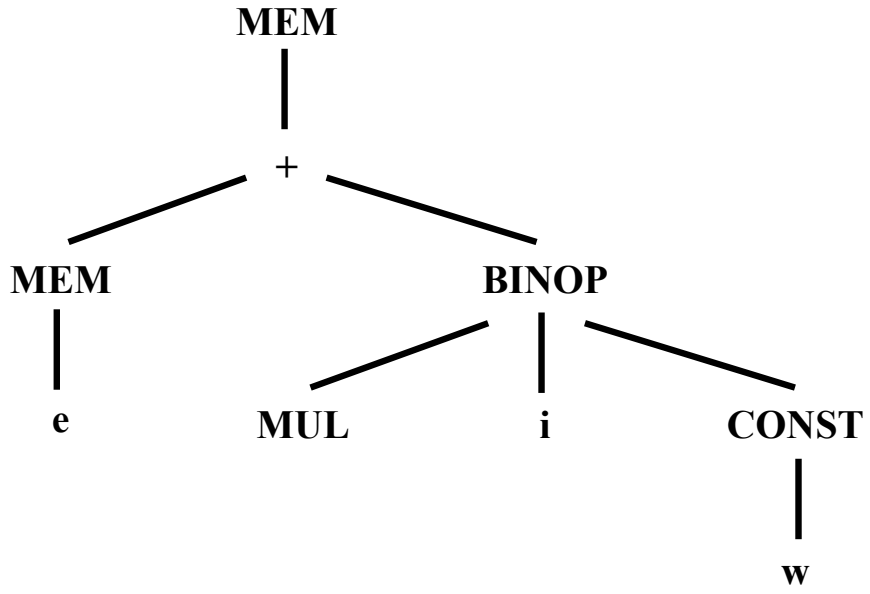
Árvore



DAG

Representação em Árvore

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



Ref.: Appel, A

Compilador do Livro do Appel

- O parser produz abstract syntax trees
- Estas árvores refletem a estrutura sintática do programa
- São a interface entre o parser e as próximas fases da compilação
- Já abstraem detalhes irrelevantes para as fases seguintes. Ex:pontuação

AST no compilador do Appel

- Geradas a partir do SableCC
- Precisam ainda ser transformadas em uma IR mais apropriada para geração de código
- Vejamos exemplos no livro ...

Tradução para IR

- Feita pelo pacote Translate
  - Capítulo 7
- A AST pode conter coisas complicadas de representar diretamente nas instruções da máquina
  - Acesso arrays, procedure calls, etc

Tradução para IR

- A IR deveria ter componentes descrevendo operações simples
  - MOVE, um simples acesso, um JUMP, etc
- A idéia é quebrar pedaços complicados da AST em um conjunto de operações de máquina
- Cada operação ainda pode gerar mais de uma instrução assembly no final

IR no compilador do Appel

```
package Tree;
abstract class Exp
 CONST(int value)
 NAME(Label label)
 TEMP(Temp.Temp temp)
 BINOP(int binop, Exp left, Exp right)
 MEM(Exp exp)
 CALL(Exp func, ExpList args)
 ESEQ(Stm stm, Exp exp)
abstract class Stm
 MOVE(Exp dst, Exp src)
 EXP(Exp exp)
 JUMP(Exp exp, Temp.LabelList targets)
 CJUMP(int relop, Exp left, Exp right, Label iftrue, Label
   iffalse)
 SEQ(Stm left, Stm right)
 LABEL(Label label)
```

IR no compilador do Appel

other classes:

ExpList(Exp head, ExpList tail)

StmList(Stm head, StmList tail)

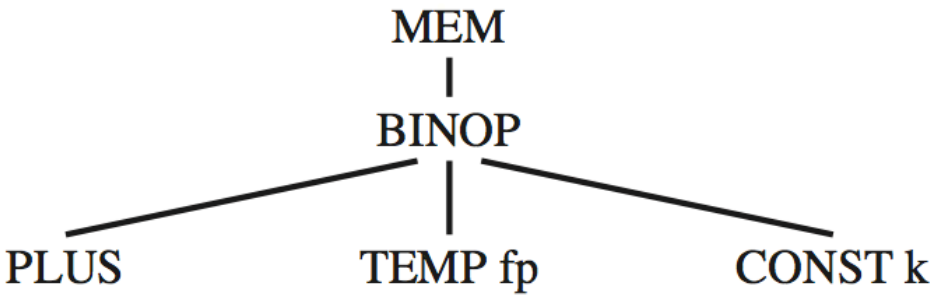
other constants:

```
final static int BINOP.PLUS, BINOP.MINUS, BINOP.MUL, BINOP.DIV,
  BINOP.AND, BINOP.OR, BINOP.LSHIFT, BINOP.RSHIFT, BINOP.ARSHIFT,
  BINOP.XOR;

final static int CJUMP.EQ, CJUMP.NE, CJUMP.LT, CJUMP.GT,
  CJUMP.LE, CJUMP.GE, CJUMP.ULT, CJUMP.ULE, CJUMP.UGT, CJUMP.UGE;
```

Tradução para IR

- Acesso a variáveis:



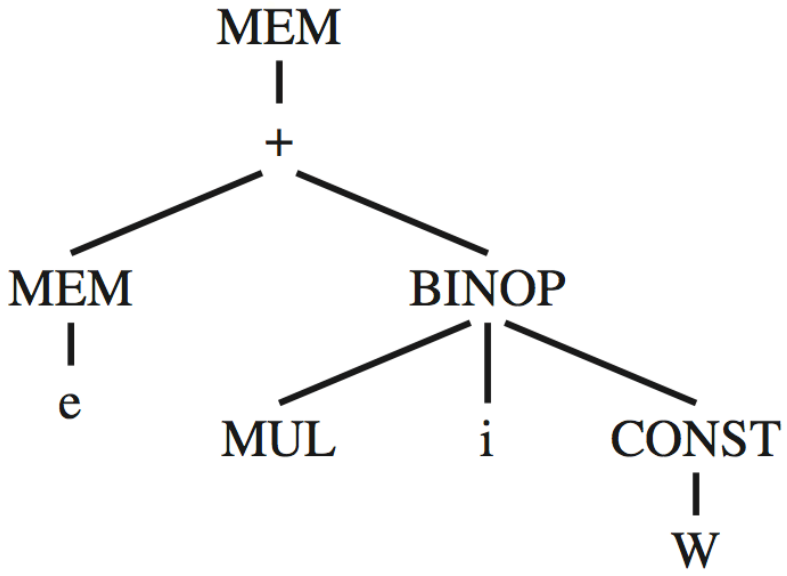
$$\text{MEM}(\text{BINOP}(\text{PLUS}, \text{TEMP fp}, \text{CONST } k))$$

- **if** $x < 5$

$$s_1(t, f) = \text{CJUMP}(\text{LT}, x, \text{CONST}(5), t, f)$$

Tradução para IR

- **Registro**: Acesso a[i] a array iniciando na posição de memória *e*



$$\text{MEM}(+(\text{MEM}(e),\ \text{BINOP}(\text{MUL},\ i,\ \text{CONST}\ W)))$$

Leitura Complementar

- Aho, et al; “Compilers – Principles, Techniques, and Tools”; Cap 8
- Appel, Andrew; ”Modern Compiler Implementation in Java”; Cap 7
- Attrot, Wesley; “Xingo – Compilação para uma Representação Intermediária”.
- The Gnu Compiler Collection Reference Manual.

Representação Linear

- RI's se assemelham a um pseudo-código para alguma máquina abstrata
  - Java: Bytecode (interpretado ou traduzido)
  - GCC RTL
  - Código de três endereços

Caso Real – Gnu Compiler Collection

- Várias linguagens: pascal, fortran, C, C++
- Várias arquiteturas alvo: MIPS, SPARC, Intel, Motorola, PowerPC, etc
- Utiliza mais de uma representação intermediária
  - árvore sintática: construída por ações na gramática
  - RTL: tradução de trechos da árvore

Caso Real – Gnu Compiler Collection

```
                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)))
```

Caso Real – Xingó

- Representação linear (XIR)
- Utiliza operações que se aproximam das disponíveis na linguagem C
- Possibilita a geração de código fonte a partir da RI

Caso Real – Xingó

```
int fat (int n) {
   if(n==0) return 1;
   else return n*fat(n-1);}
```

```
MOVE .t3 n              t3 = n;
MOVEI .t4 0             t4 = 0;
JNE .L2 .t3 .t4         if(t3!=t4) goto L2;
MOVEI .t5 1             t5 = 1;
RET .t5                 return t5;
JUMP .L1                goto L1;
LABEL .L2               L2:
MOVE .t7 n              t7 = n;
MOVEI .t8 1             t8 = 1;
SUB .t9 .t7 .t8         t9 = t7 - t8;
```

Caso Real – Xingó

```
CALL [.t10] fat
ARG .t9
MOVE .l1 .t10
MOVE .t13 n
MOVE .t15 .l1
MUL .t16 .t13 .t15
MOVE .l2 .t16
MOVE .t19 .l2
RET .t19
LABEL .L1
```

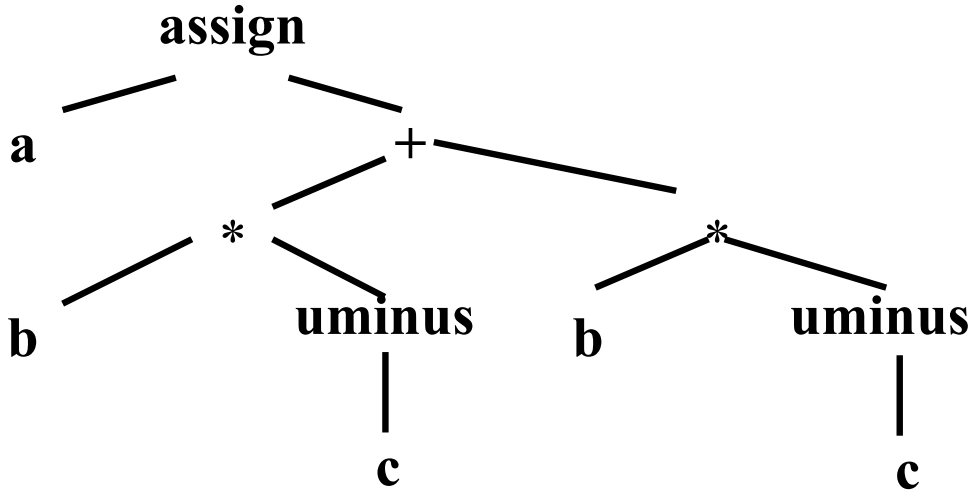
XIR

```
t10 = fat(t9);
//argumento
l1 = t10;
t13 = n;
t15 = l1;
t16 = t13 * t15
l2 = t16;
t19 = l2;
return t19;
L1;
```

Linguagem C

Código de três endereços

- Forma geral: x := y *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
```

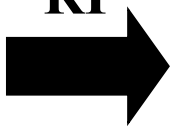
a := b * -c + b * - c

Código de três endereços

- Tipos de sentenças:
  - atribuição: x := y op z ou x := y ou x:= 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

Exemplo: Produto Interno

```
{
  ...
  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)
```

Cuidados

- Arrays
  - `x := a[i]`
  - `a[j] := y`
  - `z := a[i]`
- Pode virar
  - `x := a[i]`
  - `z := x`
  - `a[j] := y`
- Ponteiros, problema similar
  - `*p=w`

Representação

- quádruplas: (op, arg1, arg2, result)
  - (1) → (*, b, t1, t2)

- triplas:
  - (0) → (-, c,)
  - (1) → (*, b, (0))
  - (2) → (-,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

Geração de código de 3 endereços

- A partir da árvore sintática
- Emissão a partir de ações na gramática com atributos

Geração de código de 3 endereços

S→ id := E	S.code := E.code || *gen*(id.place ':=' E.place)
E → E1 + E2	E.place := *newtemp*; E.code := E1.code || E2.code || *gen*((E.place ':=' E1.place '+' E2.place)
S -> if E then S1 else S2	E. true := *newlabel*; E.false := S.next; S1.next := S.next; S.code := E.code || *gen*(E.true ':') || S1.code || *gen*('goto' S.next) || *gen*(E.false ':') || S2.code

Representação Híbrida

- Combina-se elementos tanto das RI's gráficas (estrutura) como das lineares.
  - Usar RI linear para blocos de código seqüencial e uma representação gráfica (grafo CFG) para representar o fluxo de controle entre esses blocos

Introdução

- Representação gráfica do código de 3 endereços é útil para entender os algoritmos de otimização
- Nós: computação
- Arestas: fluxo de controle
- Muito usado em coletas de informações sobre o programa

Blocos Básicos

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

```
t1 = a * a
t2 = a * b
t3 = b * 3
t4 = t2 - t3
```

Algoritmo para Quebrar em BBs

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

Quick Sort

```
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.

Dividindo em Blocos Básicos

```
(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.

Quick Sort

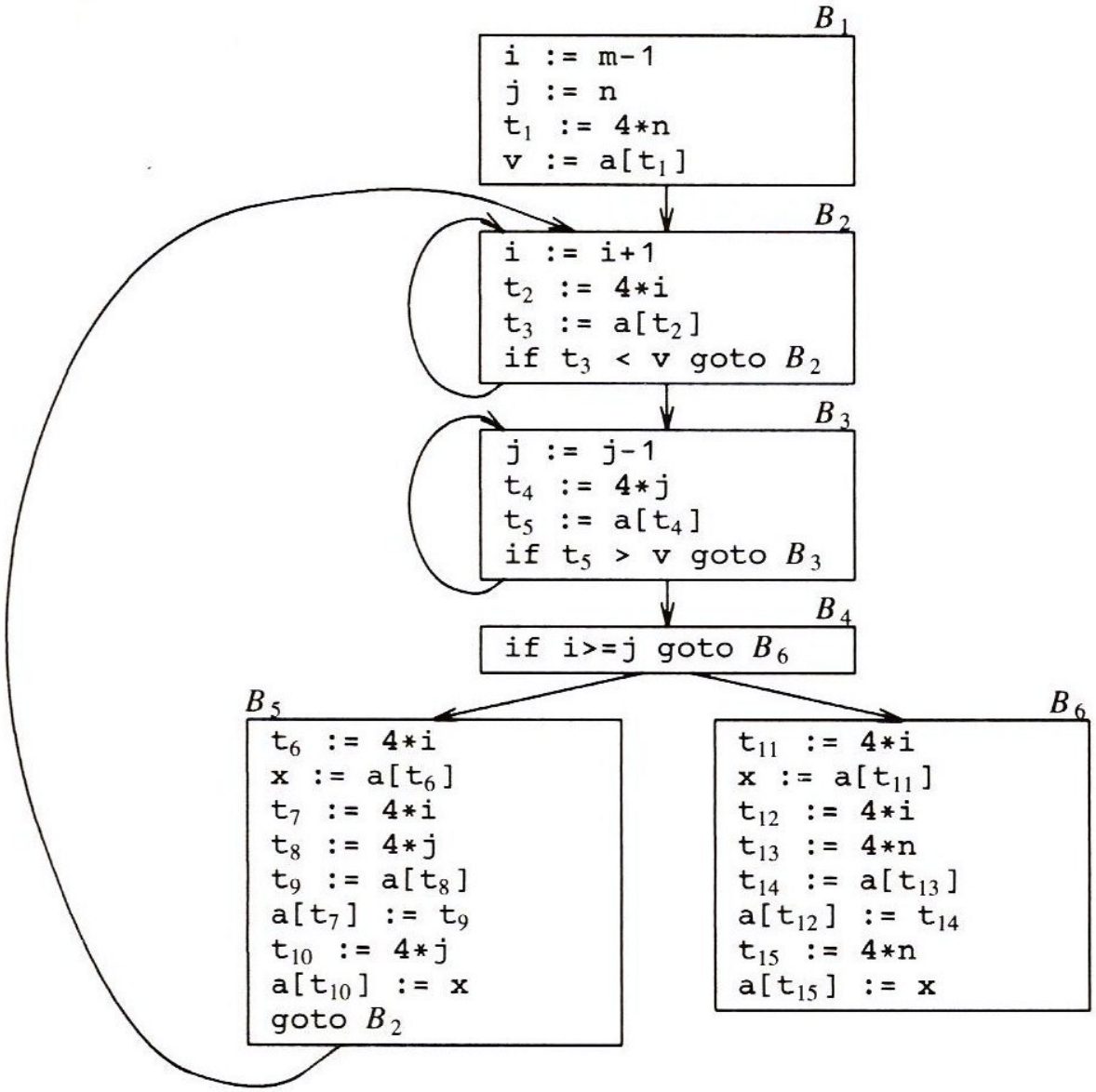
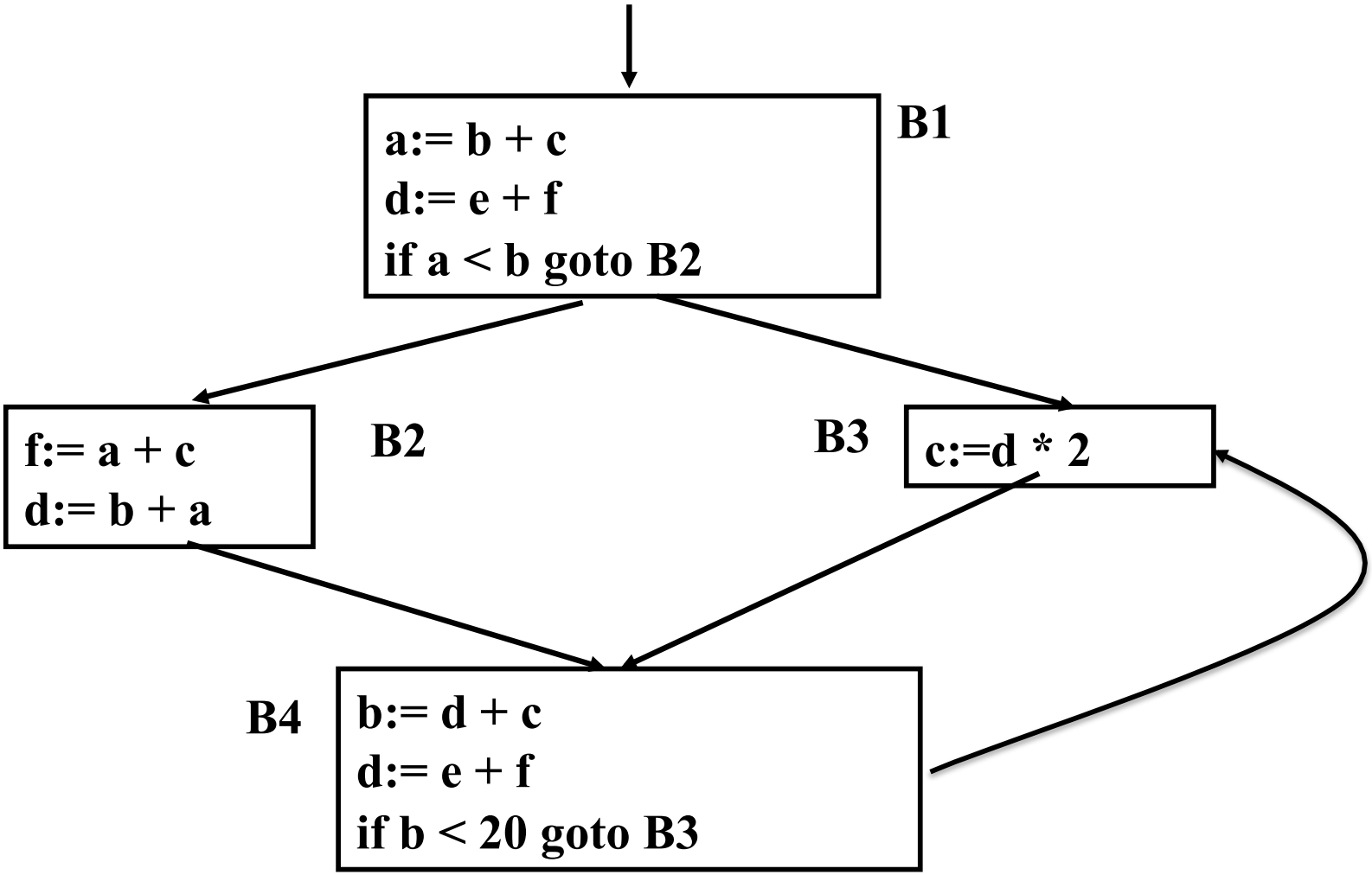


Fig. 10.5. Flow graph.

Grafo de Fluxo de Controle (CFG)



Exemplo – CFG com código 3-addr