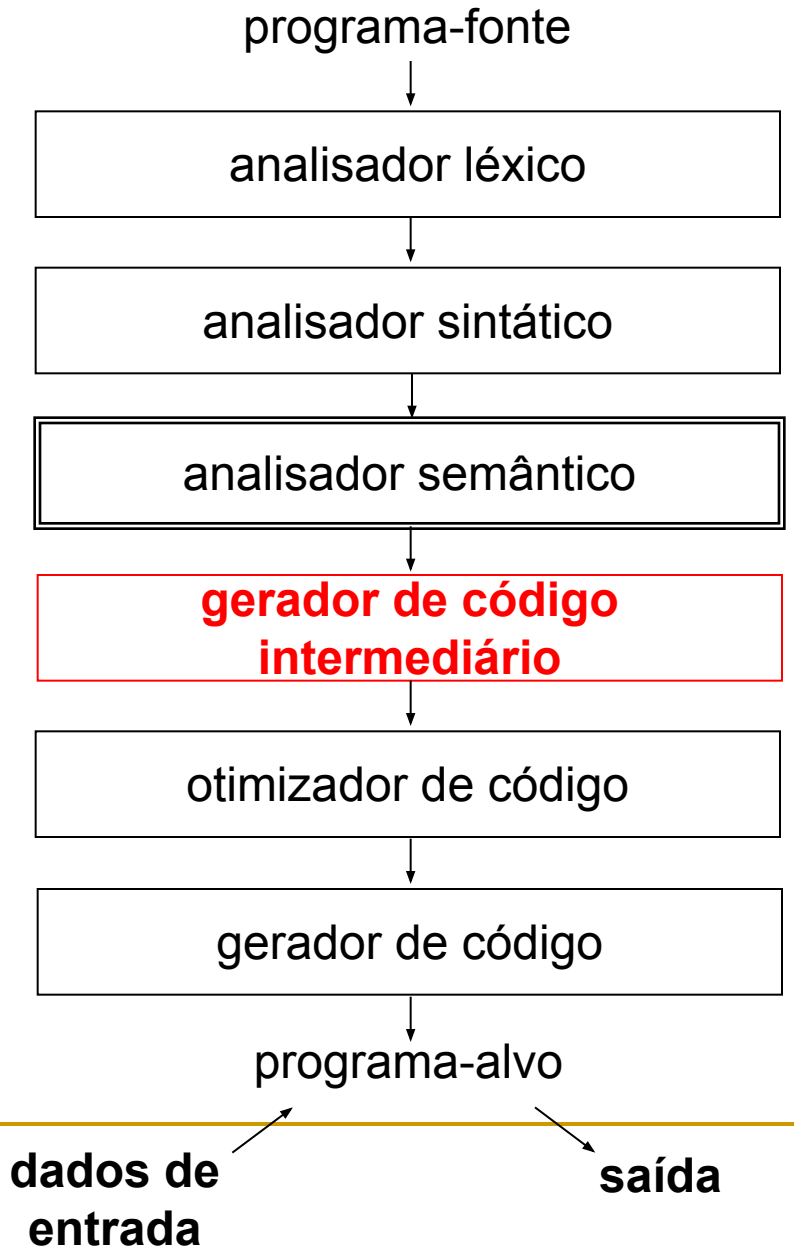
Síntese

- Geração de Código Intermediário
- Otimização de Código
- Gerador de Código

Estrutura geral de um compilador



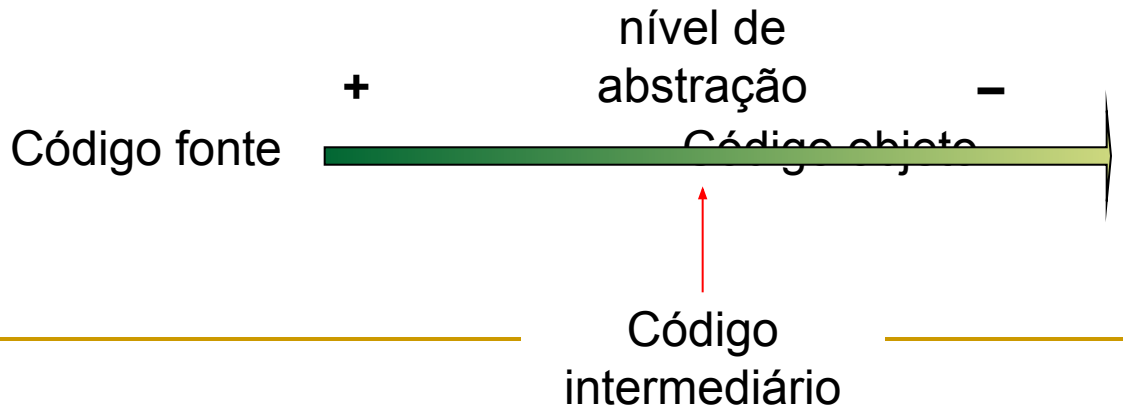
Analise

Síntese

- Na síntese, encontram-se as fases de geração de código intermediário, otimização e geração de código.

Código intermediário

- A partir da árvore de derivação pode ser gerado o código objeto final
  - Resolução de muita abstração em um passo
  - Complexidade
- Normalmente é feito um passo intermediário
  - Geração de uma representação intermediária do código

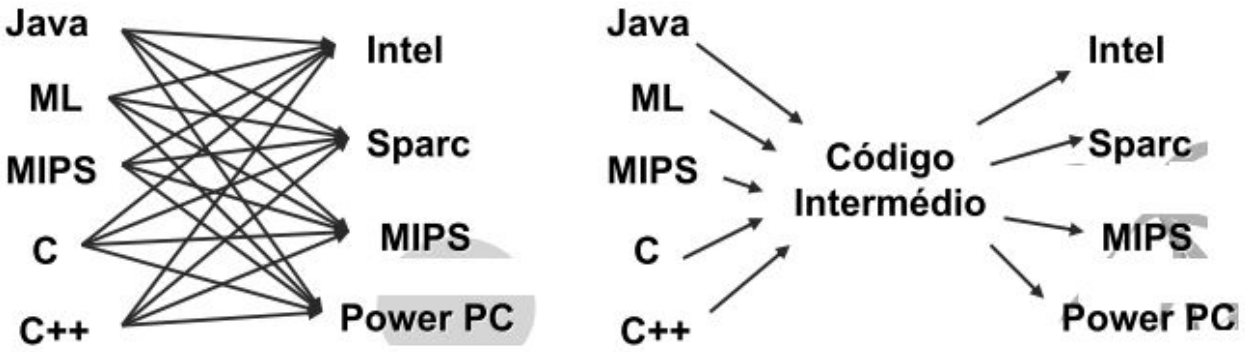


Código intermediário

- O gerador de código intermediário usa as estruturas produzidas pelo analisador sintático e verificadas pelo analisador semântico para criar uma seqüência de instruções simples, denominada código intermediário
  - está entre a linguagem de alto nível e a linguagem de baixo nível
- O código intermediário é o que mais se aproxima do programa executável, porém, ele passa por mais etapas até virar o executável final.

Código intermediário

- Vantagens
  - Possibilita a otimização do código intermediário
    - Código objeto final mais eficiente
  - Simplifica a implementação do compilador
    - Resolução gradativa da abstração das operações
      - O código intermediário abstrai detalhes da máquina alvo
  - Possibilita a tradução do código intermediário para diversas máquinas



Código intermediário

- **Desvantagem**
  - O compilador precisa realizar um passo a mais, logo a tradução do código fonte para o objeto leva a uma compilação mais lenta.
- **Intermediário X objeto final**
  - O intermediário não especifica detalhes da maquina alvo, tais como
    - quais registradores serão usados,
    - quais endereços de memória serão referenciados, etc.

Código intermediário

- Representações intermediárias
  - representação gráfica: árvore sintática ou grafo
  - Notação pós-fixada e pré-fixada
  - Código de três endereços

- A representação intermediária pode ser construída paralelamente à análise sintática
  - Tradução dirigida pela sintaxe
    - Compilação em um passo

Código de três endereços

- O código de três endereços é formado por uma seqüência de comandos com o seguinte formato geral:

  A := B op C

  A := op B

  A := B

  goto L

  if A oprel B goto L

  - onde:
    - **A**, **B** e **C** são nomes, constantes ou objetos de dados temporários criados pelo compilador
    - **op** está no lugar de qualquer operador aritmético de ponto fixo ou flutuante
    - **oprel** é um operador relacional
    - **L** é um rótulo simbólico

Código de três endereços

- Outros enunciados que serão usados para chamadas de procedimentos

  param X

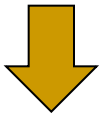
  call P, N

  return Y

  - onde:
    - **X** é um parâmetro do procedimento
    - **P** é o nome do procedimento
    - **N** é número de parâmetros do procedimento
    - **Y** é o valor retornado (opcional)

Exemplo

- Código de três endereços para o comando de atribuição A := X + Y * Z



```
t1 := Y * Z
t2 := X + t1
A := t2
```

t1 e t2 são variáveis temporárias criadas pelo compilador

- É chamado assim porque utiliza no máximo três posições de memória, duas para os operandos e uma para o resultado.

Exercício

- Escreva o código de três endereços para o comando de atribuição **a := b * (- c) + b * (- c)**

Exercício

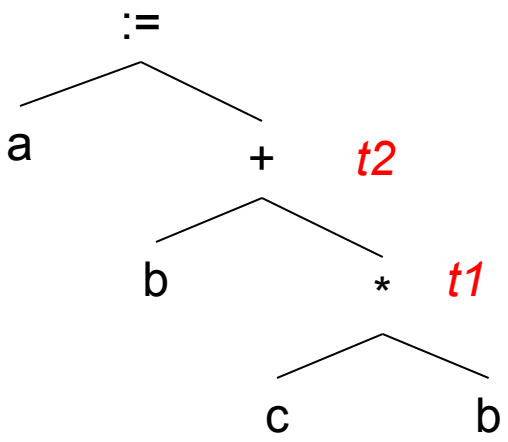
- Escreva o código de três endereços para o comando de atribuição **a := b * (- c) + b * (- c)**

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

Exercício

- Escreva o código de três endereços para o comando de atribuição **a := b + c * b**

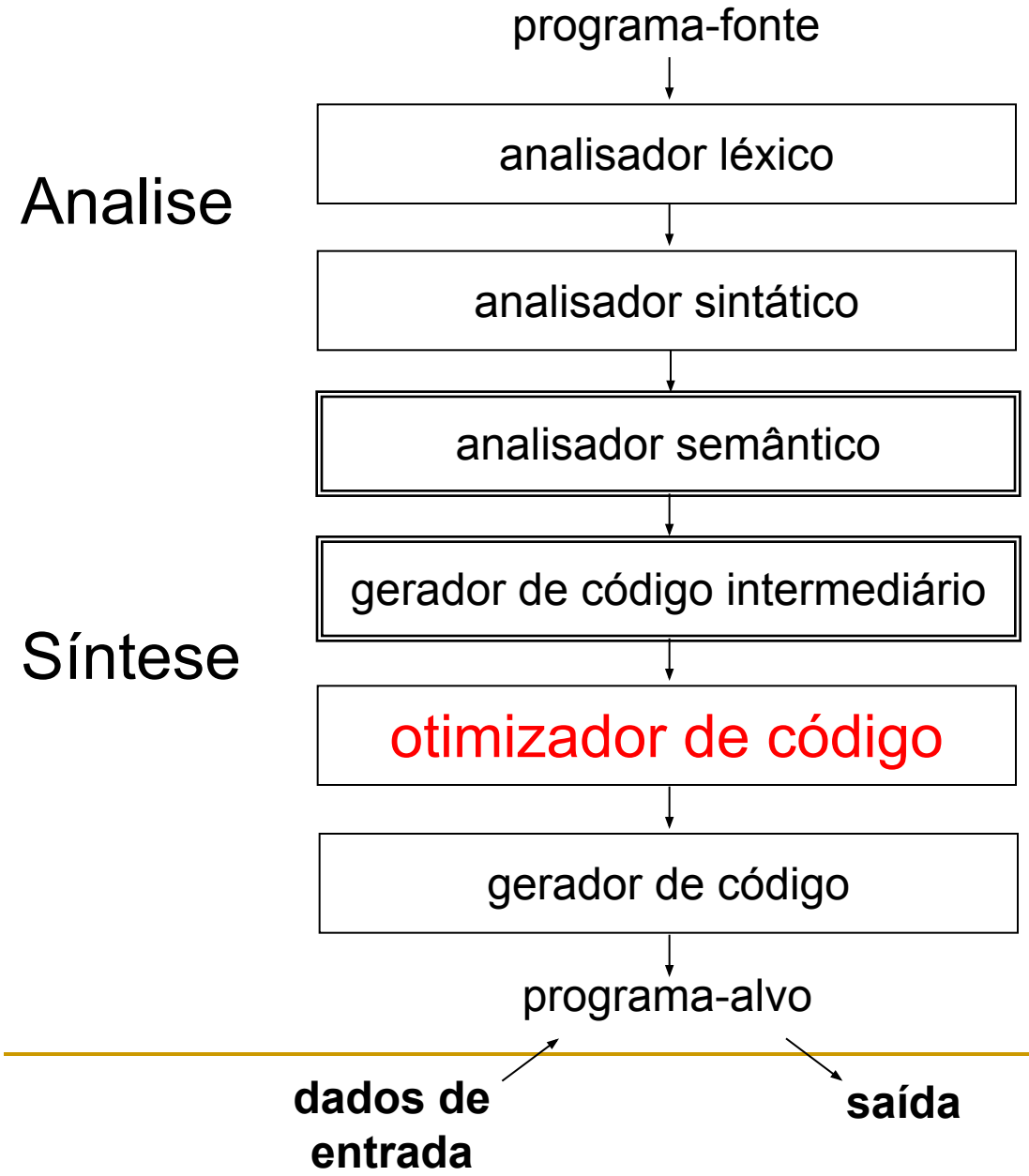
```
t1 := c * b
t2 := b + t1
a := t2
```



Código intermediário

- Revendo....
  - Código de três endereços
    - Cada instrução faz referência, no máximo, a três variáveis (endereços de memória)
      - A := B + C
  - A representação intermediária pode ser construída paralelamente à análise sintática
    - Tradução dirigida pela sintaxe
  - O código intermediário não especifica detalhes da máquina alvo

Estrutura geral de um compilador



Otimizador de Código

Referência: http://www-di.inf.puc-rio.br/~rangel/comp/otim.pdf

Otimizador de Código

- A otimização do código é a inferência de um conjunto de mudanças que melhoram a seqüência de instruções de máquina (tempo de execução, memória ocupada, etc) sem que seja modificada a semântica.
- Apesar do termo otimização, são poucas às vezes que se pode garantir que o código obtido seja o melhor possível.
- A otimização de código faz com que o compilador gaste muito tempo de compilação
  - Deve ser implementada se o uso do compilador realmente necessite de um código objeto (código de máquina) eficiente.

Otimizador de Código

- O difícil na otimização é não modificar em nenhum caso o funcionamento do programa.
- Por exemplo, considere um trecho de programa em que aparece um comando x=a+b; Este programa pode ser melhorado (torna-se mais rápido e menor) se este comando for retirado.
- Mas para que o funcionamento do programa não seja alterado deve-se verificar algumas propriedades, por ex.:
  - o comando é inútil, porque nenhum dos comandos executados posteriormente, usa o valor da variável x
  - o comando x=a+b; nunca é executado. Por exemplo, está em seguida a um if cuja condição nunca é satisfeita:

```
if(0)
x=a+b;
```

Otimizador de Código

- Oportunidades de otimização
  - Suponha que a mesma expressão ocorre mais de uma vez em um trecho de programa.
  - Se as variáveis que ocorrem na expressão não tem seus valores alterados entre as duas ocorrências, é possível calcular seu valor apenas uma vez. Exemplo:

```
x=a+b;
...
y=a+b;
...
```

  - Se os valores de a e de b não são alterados, é possível guardar o valor da expressão a+b em uma temporária (t1) e usá-lo posteriormente.

```
t1=a+b;
x=t1;
...
y=t1;
```

Otimizador de Código

Eliminação de subexpressões comuns

Operações que se repetem sem que seus argumentos sejam alterados podem ser realizadas uma única vez

Exemplo

Sem atribuição a `a` ou `b` entre as duas instruções:

```
x = a + b + c;
...
y = a + b + d;
```

Sem otimização:

```
_t1 := a + b
x := _t1 + c
...
_t2 := a + b
y := _t2 + d
```

Com otimização:

```
_t1 := a + b
x := _t1 + c
...
y := _t1 + d
```

Otimizador de Código

Eliminação de código redundante

Instruções sem efeito podem ser eliminadas

Exemplo

Sem nenhuma atribuição a `x` ou a `y` entre as duas instruções,

```
x := y
...
x := y
```

```
x := y
...
y := x
```

a segunda instrução pode ser seguramente eliminada

Otimizador de Código

Propagação de cópias

Variáveis que só mantêm cópia de um valor, sem outros usos, podem ser eliminadas

Exemplo

Sem outras atribuições a `y` e sem outros usos de `x`

```
x := y
...
z := x
```

pode ser reduzido a

```
...
z := y
```

Otimizador de Código

Eliminação de desvios desnecessários

Desvio incondicional para a próxima instrução pode ser eliminado

Exemplo

```
        a := _t2
        goto _L6
_L6:    c := a + b
```

equivale a

```
        a := _t2
        c := a + b
```

Otimizador de Código

Uso de propriedades algébricas

Substituição de expressões aritméticas por formas equivalentes

Original	Equivalente
x + y	y + x
x + 0	x
x - 0	x
x * y	y * x
x * 1	x
x / 1	x
2 * x	x + x
x^2	x * x

Otimizador de Código

- Outro exemplo de otimização é a retirada de comandos de laço de repetição (um *loop*). Por exemplo:

```
for (i=0; i<N; i++) {
a=j+5;
f(a*i);    }
```

- poderia ser melhorado retirando-se o comando a=j+5; do for.

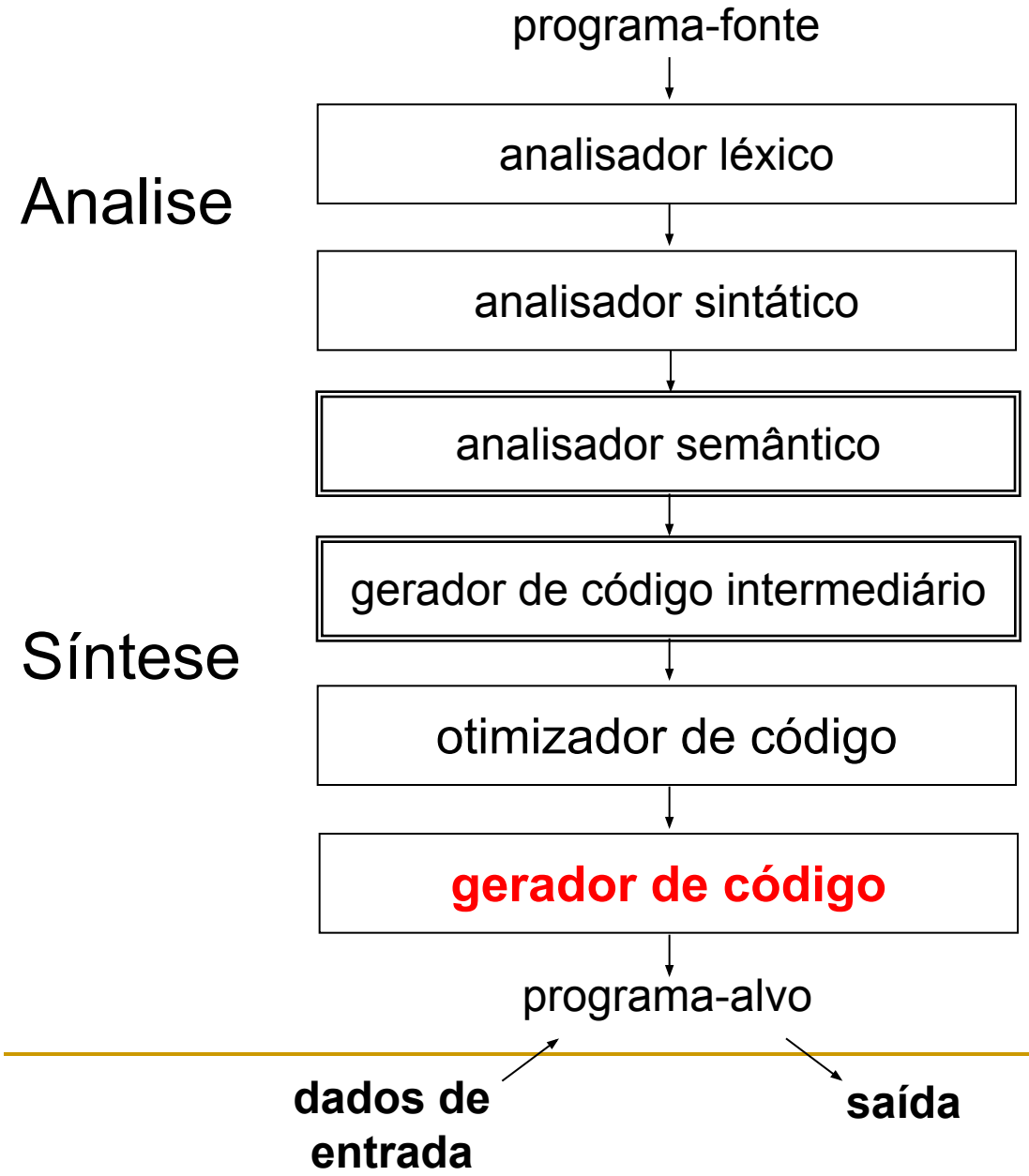
```
a=j+5;
for (i=0; i<N; i++)
f(a*i);
```

- Por outro lado, se N=0, o programa foi "piorado", porque o comando a=j+5; que era executado 0 vezes, passou a ser executado 1 vez.
- Pode haver um problema maior: se a variável a é usada após o loop,em vez de seu valor original, seu valor será, incorretamente, o resultado dado pela atribuição.

Otimizador de Código

- Um outro problema é a quantidade de informação que se deseja manipular:
  - examinar otimizações locais (em trechos pequenos de programas, por exemplo trechos sem desvios, ou seja, trechos em linha reta),
  - otimizações em um nível intermediário (as otimizações são consideradas apenas em funções, módulos, ou classes, dependendo da linguagem) e
  - otimizações globais (que consideram as inter-relações de todas as partes de um programa).
- A maioria dos compiladores oferece algumas otimizações do primeiro

Estrutura geral de um compilador



Geração de Código Objeto

Referências: http://www-di.inf.puc-rio.br/~rangel/comp/gercod.pdf

Geração de Código Objeto

- Enquanto a fase de análise é essencialmente dependente da **linguagem** de programação, a fase de geração de código é dependente principalmente da **máquina alvo**
- Sua principal função é gerar o código equivalente ao programa fonte para uma máquina real
- A fase de tradução converte o código fonte para um código objeto, que pode ser:
  - um ASSEMBLY de uma determinada máquina
  - um pseudo-código de uma máquina hipotética
    - interpretado posteriormente,
    - pode ser executado em qualquer máquina

Geração de Código Objeto

Exemplo de código em linguagem simbólica

```
        .file   "hello.cpp"
        .text
        .align 2
.LCFI2:
        movl    %eax, -4(%ebp)
        movl    %edx, -8(%ebp)
        cmpl    $1, -4(%ebp)
        jne     .L5
```

Geração de Código Objeto

- Os principais requisitos impostos a geradores de código objeto são:
  - O código gerado deve ser correto e de alta qualidade
  - O código gerado deve fazer uso efetivo dos recursos da máquina; e
  - O código gerado deve executar eficientemente.
- O problema de gerar código ótimo é insolúvel (indecidível) como tantos outros. Na prática, devemos usar heurísticas que geram um “bom” código.

Geração de Código Objeto

- A última etapa do compilador propriamente dito é a geração do código em linguagem simbólica
- Uma vez que esse código seja gerado, outro programa– o montador – será responsável pela tradução para o código em formato de linguagem de máquina.

Geração de Código Objeto

- A abordagem mais simples da etapa de geração de código objeto é:
  - para cada instrução (do código intermediário) ter um gabarito com a correspondente seqüência de instruções em linguagem simbólica do processador-alvo
  - Por exemplo:

    le := ld1 + ld2

    - A seqüência de instruções em linguagem simbólica que corresponde a essa instrução depende da arquitetura do processador para o qual o programa é gerado.

Geração de Código Objeto

- Diferentes processadores podem ter distintos formatos de instruções
- Classificação pelo número de endereços na instrução:
  - 3 : dois operandos e o resultado
  - 2 : dois operandos (resultado sobrescreve primeiro operando)
  - 1 : só segundo operando, primeiro operando implícito (registrador acumulador), resultado sobrescreve primeiro operando
  - 0 operandos e resultado numa pilha, sem endereço explícitos

Geração de Código Objeto

Tradução para linguagem simbólica

Tradução ocorre segundo gabaritos definidos de acordo com o tipo de máquina

Exemplo: x := y + z

3-end

```
ADD y, z, x
```

2-end

```
MOVE Ri, y
ADD  Ri, z
MOVE x, Ri
```

1-end

```
LOAD  y
ADD   z
STORE x
```

0-end

```
PUSH y
PUSH z
ADD
POP  x
```

Produção do Código Executável

- O resultado da compilação é um arquivo em linguagem simbólica
- Montagem
  - Processo em que o programa em linguagem simbólica é transformado em formato binário, em código de máquina.
  - O programa responsável por essa transformação é o montador
- Montadores
  - Traduzem código em linguagem simbólica para linguagem de máquina

Exercício

- Sobre o processo de compilação, assinale V para verdadeiro e F para falso.
  - O resultado da compilação é um arquivo em linguagem simbólica
  - A última etapa do compilador propriamente dito é a geração do código em linguagem executável
  - A fase de análise é essencialmente dependente da máquina alvo
  - A fase de geração de código objeto é dependente principalmente da linguagem de programação
  - A fase de tradução converte o código fonte para um código objeto, que pode ser um ASSEMBLY de uma determinada máquina