Linguagens Formais e Compiladores

Prof^a Jerusa Marchi

jerusa.marchi@ufsc.br

Departamento de Informática e Estatística

Teoria da Computação

- Estudo de máquinas e linguagens:
  - Computável
  - Decidível
  - Tratável
- Tese de Church Turing
- Algoritmos $\times$ Procedimentos

Teoria da Computação

- Classes de Máquinas (AF, AP, MT)
- Classes de Linguagens (Regulares, Livres de Contexto, Sensíveis ao Contexto, Recursivas e Recursivamente Enumeráveis)
  - Representação Finita de Linguagens possivelmente infinitas

Linguagens Formais e Compiladores

- Estudar classes de linguagens, seus mecanismos de representação e suas propriedades
  - Gramáticas, Expressoões Regulares, Máquinas
- Aplicação práticas dos formalismos e propriedades no processo de desenvolvimento e compilação de linguagens

Linguagens de Programação

Evolução

- 1^a Geração: linguagem de máquina (portas lógicas)
- 2^a Geração: linguagem de montagem (Assembly)
  - representação simbólica da linguagem de máquina
  - precisa ser **traduzido** para a linguagem de máquina (assembler ou montador)
- 3^a Geração: linguagens estruturadas orientadas a procedimentos
  - Fortran, Algol, Pascal, C, COBOL, Basic, Ada,...
- 4^a Geração: linguagens orientadas à aplicação
- 5^a Geração: linguagens voltadadas para a inteligência artificial - paradigmas lógico e funcional (PROLOG e LISP)
- 6^a Geração: ??

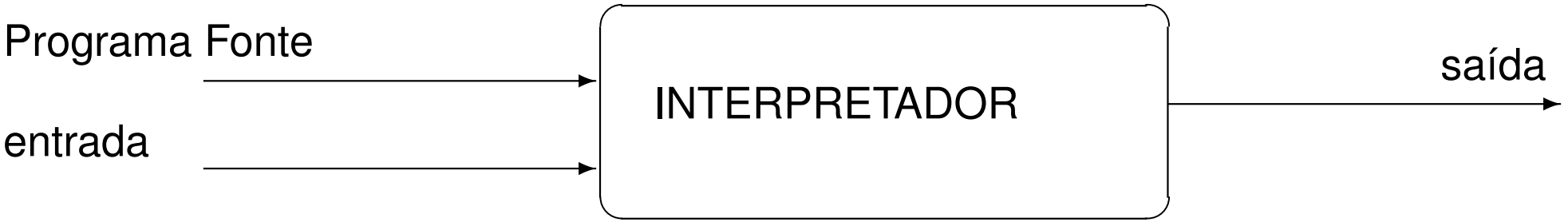
Definições Preliminares

- Todo e qualquer código fonte precisa ser **traduzido** para que possa ser executado pela máquina.
- Um **Compilador** é um programa que recebe como entrada um programa em uma linguagem de programação - a linguagem *fonte* - e o traduz para uma linguagem equivalente - a linguagem *objeto*
  - Um papel importante do compilador é reportar erros no programa fonte durante o processo de tradução.

- Se o programa objeto for um programa em uma linguagem de máquina executável, este poderá então ser chamado pelo usuário

Definições Preliminares

- Um *Interpretador* é outro tipo comum de processador de linguagem
  - Ao invés de produzir um programa objeto, um interpretador executa diretamente operações especificadas no programa fonte sobre as entradas fornecidas pelo usuário
  - Cada linha é interpretada e, estando correta, executada



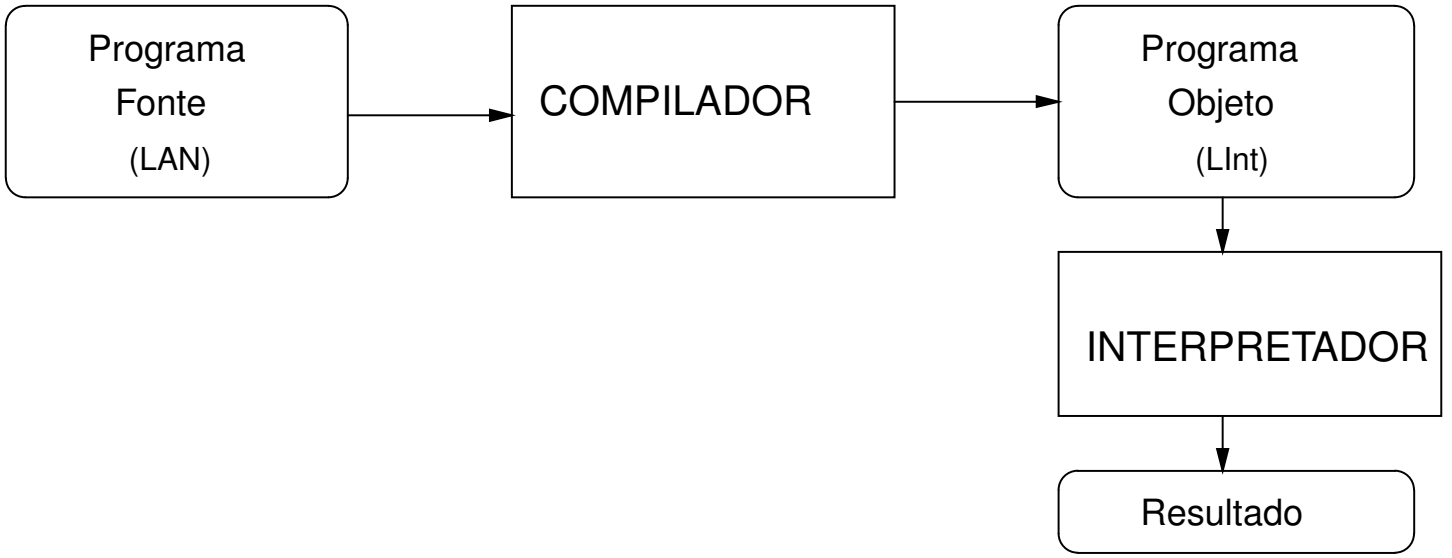
Definições Preliminares

- Compilador $\times$ Interpretador

	Compilador	Interpretador
Tempo compilação	grande	-
Tempo execução	+ rápido	+ lento
Consumo memória	maior	menor
Código fonte	oculto	visível
Otimização	sim	não
Depuração	+ complexa	simples
Execução com erro	não	sim

Definições Preliminares

- Compilador/Interpretador (híbrido)
  - o programa fonte é lido e traduzido em um código intermediário, ao mesmo tempo, é inicializada uma máquina virtual que recebe este código como entrada e o executa, retornando o resultado



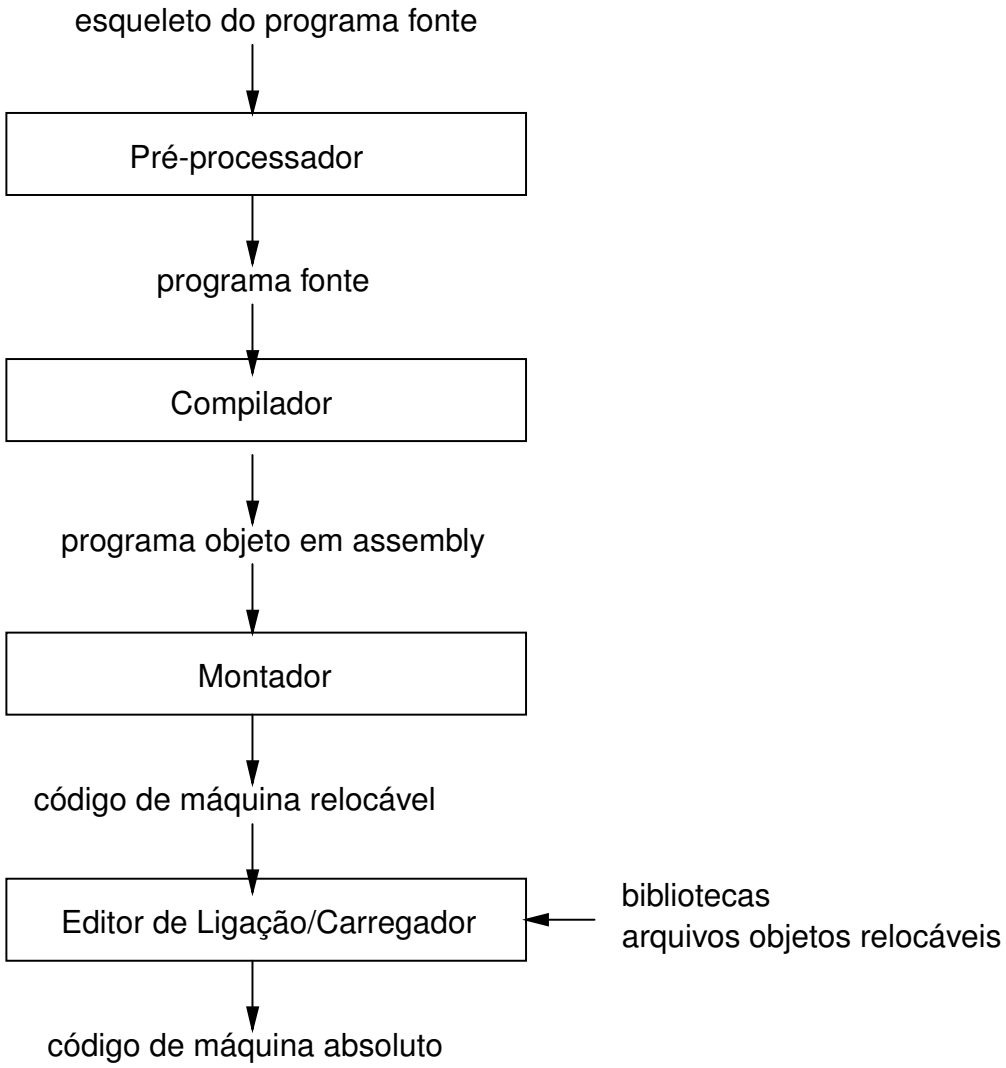
Definições Preliminares

- Os processadores da linguagem Java são híbridos - combinam compilação e interpretação
- O programa fonte é compilado, gerando uma forma intermediária - bytecodes
- O bytecode é interpretado por uma máquina virtual Java - a máquina pode diferir daquela em que o bytecode foi gerado.

Definições Preliminares

- Além do compilador, outros programas podem ser necessários para a criação do programa objeto executável
  - O código fonte pode estar dividido em módulos armazenados em arquivos separados ou conter macros que precisam ser expandidas - *pré-processador*
  - O compilador pode gerar uma linguagem de simbólica (assembly) que precisa ser traduzida para código de máquina relocável - *assembler ou montador*
  - Programas grandes são compilados em partes e seus códigos de máquina relocável precisam ser unidos, além disso, ainda há a necessidade de linkar o código com bibliotecas, resolvendo endereçamentos de memória - *linker ou editor de ligação*
  - Os arquivos objeto executáveis são carregados para a memória - *loader ou carregador*

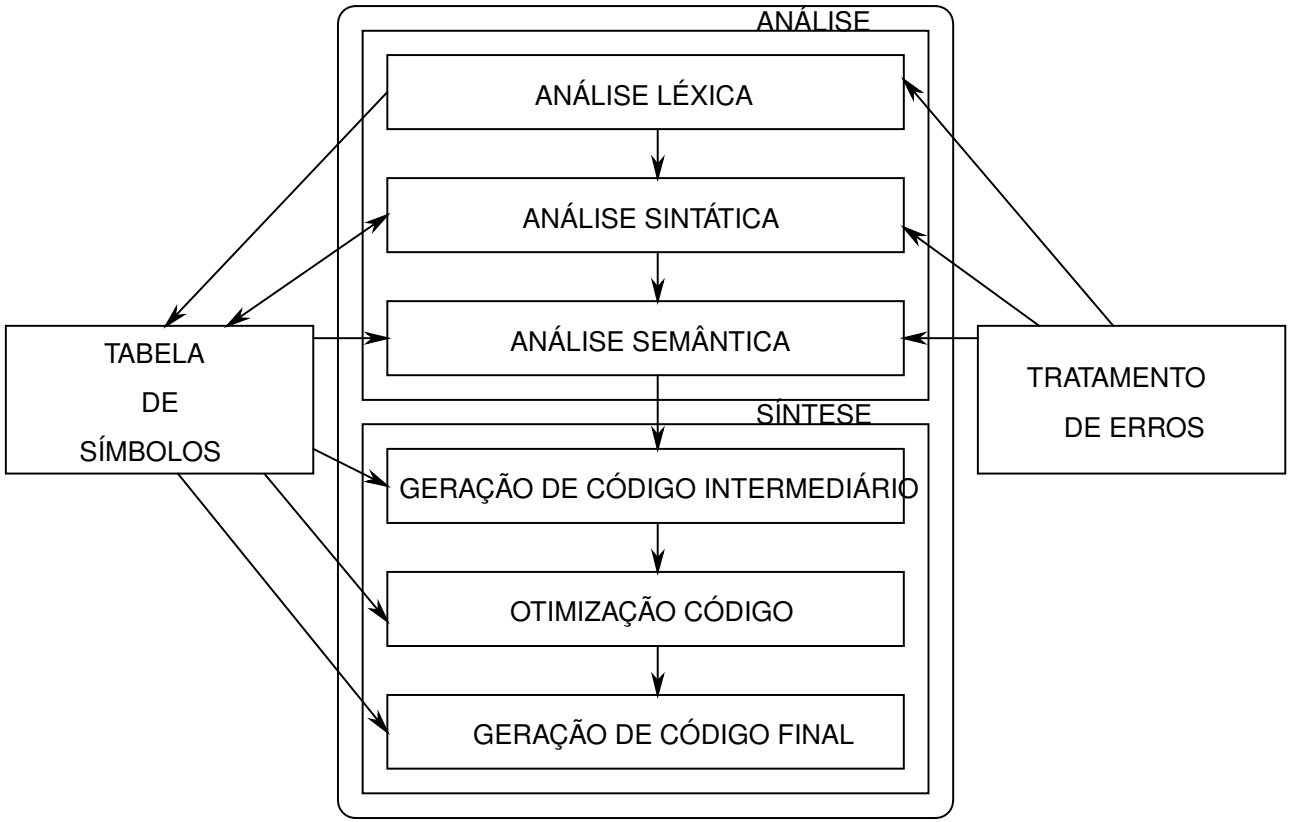
Contexto de um Compilador



Estrutura Geral do Compilador

- O processo de compilação é dividido em duas partes:
  - Análise
    - Divide o programa fonte nas partes que o constituem e cria uma representação intermediária
    - Fases: *análise léxica*, *sintática* e *semântica*
  - Síntese
    - Constrói o programa alvo a partir da representação intermediária
    - Fases: *geração de código intermediário*, *otimização de código* e *geração de código final*

Estrutura Geral do Compilador



Estrutura Geral do Compilador

- Outro modelo possível:
  - *Interface de Vanguarda (front-end)*: fases que dependem primariamente da linguagem e são independentes da máquina alvo (análise léxica, sintática e semântica, juntamente com a criação da tabela de símbolos e a geração de código intermediário)
  - *Interface de Retaguarda (back-end)*: fases que dependem da máquina alvo e não dependem da linguagem fonte (otimização (opcional) e geração de código)

Implementação de um Compilador

- *Passo*: Passagem completa do programa compilador sobre o programa fonte que está sendo compilado, gerando um arquivo de saída
  - Cada fase pode ser implementada em um passo separado, gerando resultados parciais como saída.
  - Também pode ser centrado na análise sintática (compilador de 1 único passo)
    - O analisador sintático solicita ao analisador léxico que lhe forneça os tokens
    - Ao receber uma estrutura sintática completa, e estando esta correta, o analisador sintático chama o analisador semântico
    - Estando a estrutura correta semanticamente, o analisador sintático solicita a geração de código intermediário
    - Traduzida a estrutura, o controle volta para o analisador sintático

Implementação de um Compilador

- quanto maior o número de passos:
  - maior a complexidade do compilador
  - maior o tempo de compilação
  - maior a possibilidade de executar otimizações
  - menor consumo de memória

Analisador Léxico

- Também chamado de *scanner*
- interface entre o programa fonte e o compilador
- funções:
  - ler o programa fonte
  - agrupar caracteres em sequências significativas - *lexemas*
  - produz como saída uma sequência de *tokens* - par $\langle nome - token, valor - atributo \rangle$
    - $nome - token$ - identifica o tipo de item léxico encontrado:
      - identificadores, palavras reservadas, constantes, símbolos especiais
  - ignorar elementos sem valor sintático
    - espaços em branco, comentários e caracteres de controle
  - detectar erros léxicos
    - símbolos inválidos

Analisador Léxico

- Exemplo:

```
montante = dep_inicial + t_juros * 60
```

  - `montante` é um lexema mapeado para o token $\langle id, 1\rangle$ onde id é um símbolo abstrado que significa identificador e 1 aponta para a entrada da tabela de símbolos (TS) onde se encontra `montante`
  - O símbolo de atribuição $=$ é mapeado para $\langle = \rangle$ (sem valor de atributo)
  - `dep_inicial` é mapeado para o token $\langle id, 2\rangle$ onde 2 é onde se encontra `dep_inicial` na TS
  - $+$ é mapeado para $\langle + \rangle$
  - `t_juros` é mapeado para $\langle id, 3\rangle$ onde 3 aponta `t_juros` na TS
  - $*$ é um lexema mapeado para o token $\langle * \rangle$
  - 60 é mapeado para $\langle num, 4\rangle$, sendo o valor 60 armazenado na TS

Analisador Léxico

- A fase de análise é terminada ainda que erros léxicos sejam identificados, contudo não faz sentido prosseguir na análise sintática
- Implementação de analisadores léxicos:
  - Geradores Automáticos de Analisadores Léxicos (LEX)
  - Entrada: especificação léxica da linguagem
  - Formalismo: Gramáticas Regulares e Autômatos Finitos

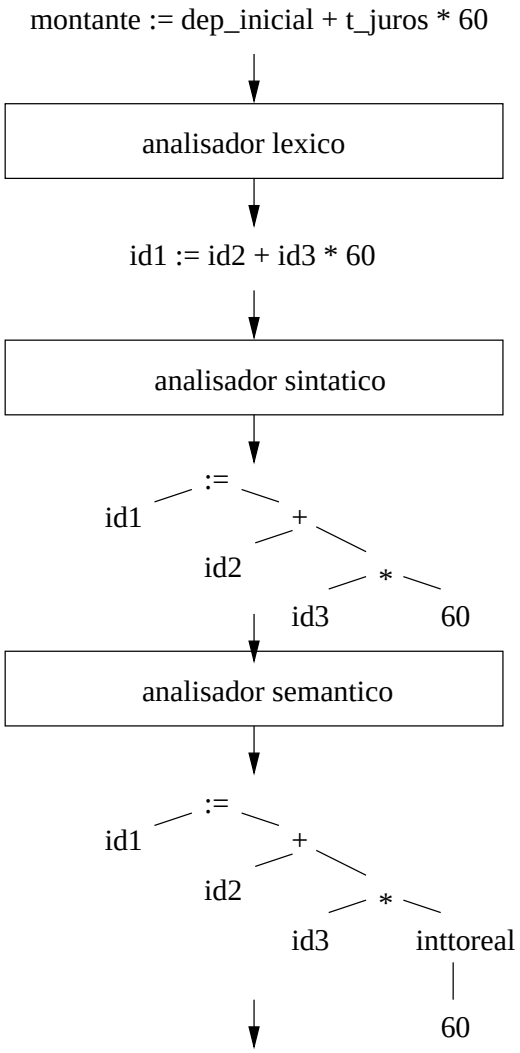
Analisador Sintático

- funções:
  - agrupar os **tokens** em estruturas sintáticas (expressões, comandos, declarações, etc.) na forma de uma árvore de derivação
  - verificar se a sintaxe da linguagem na qual o programa foi escrito está sendo respeitada
    - detectar/diagnosticar erros sintáticos - lexema que desrespeita a estrutura sintática esperada

Exemplo

tabela de simbolos

1	montante	. . .
2	dep_inicial	. . .
3	t_juros	. . .
4		



Analisador Sintático

- Implementação de analisadores sintáticos:
  - Geradores Automáticos de Analisadores Sintáticos (YACC)
  - Entrada: especificação sintática da linguagem (gramática)
  - Formalismo: Gramáticas Livres de Contexto e Autômatos de Pilha

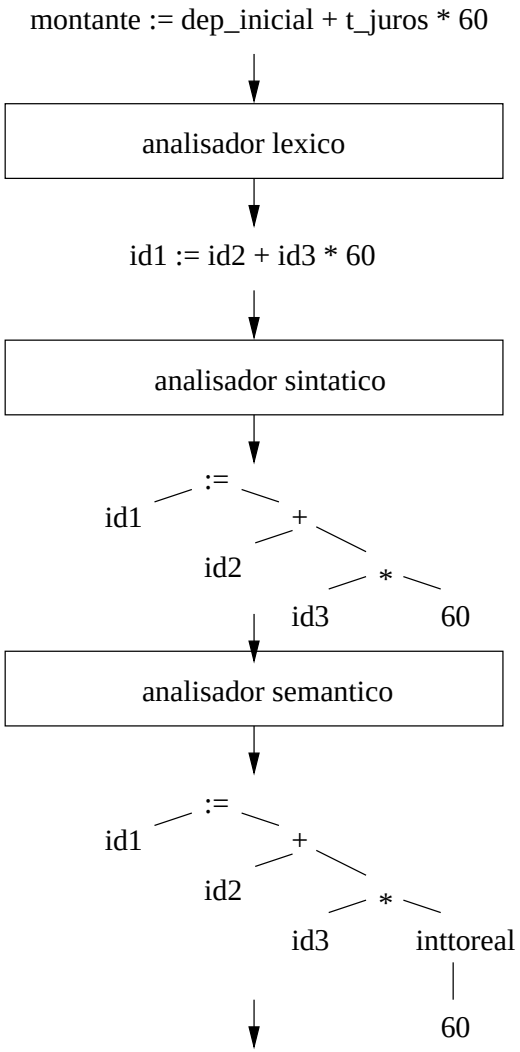
Analisador Semântico

- Utiliza a árvore sintática e as informações na tabela de símbolos para verificar a consistência semântica do programa fonte com a definição da linguagem
  - verificação da compatibilidade de tipos (coerção), declaração de variáveis (ausência ou múltiplas declarações), coerência entre declaração e uso de identificadores
- Reúne informações sobre os tipos e as salva na árvore sintática (árvore sintática anotada) para uso na geração de código intermediário
- Detecta erros semânticos

Exemplo

tabela de simbolos

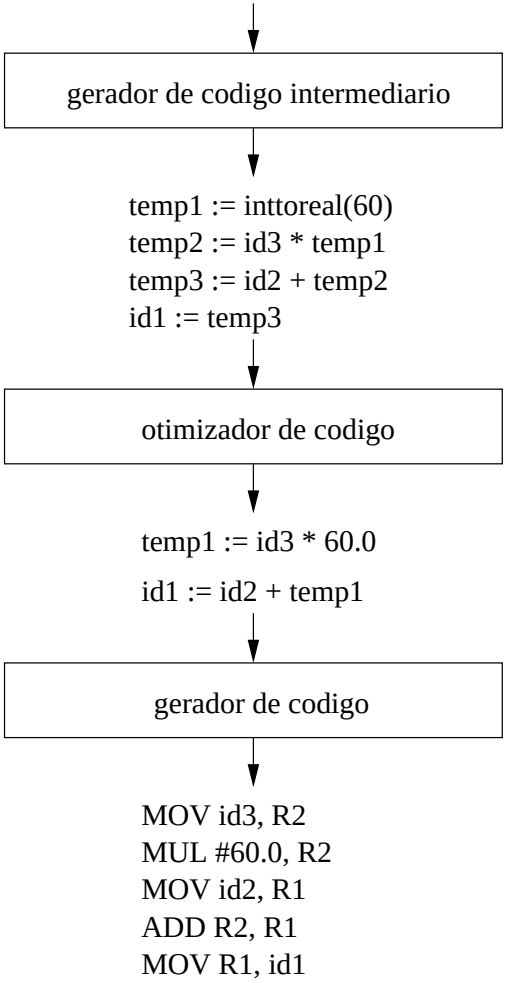
1	montante	. . .
2	dep_inicial	. . .
3	t_juros	. . .
4		



Gerador de código intermediário

- consiste na geração de um conjunto de instruções equivalentes ao programa fonte para uma máquina hipotética (virtual)
- a representação intermediária deve ser facilmente produzida e ser facilmente traduzida para a máquina alvo
- objetivos:
  - facilitar a tradução para a linguagem objeto
  - permitir a otimização de código
  - portabilidade

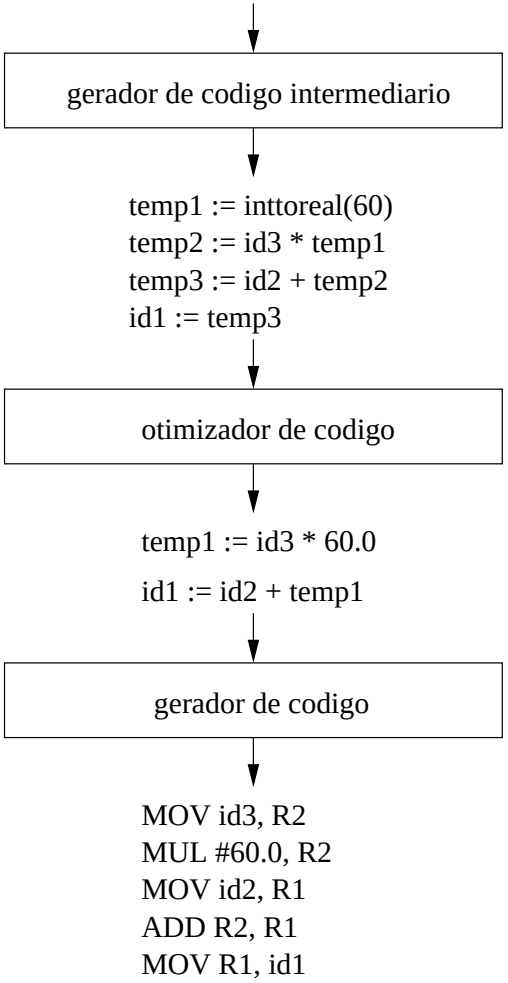
Exemplo


gerador de codigo intermediario
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
otimizador de codigo
temp1 := id3 * 60.0
id1 := id2 + temp1
gerador de codigo
MOV id3, R2
MUL #60.0, R2
MOV id2, R1
ADD R2, R1
MOV R1, id1

Otimizador de Código

- Melhorar o código, de forma que a execução seja mais eficiente quanto ao tempo e/ou espaço ocupado
- otimizações mais frequentes:
  - agrupamentos de sub-expressões comuns
  - retirada de comandos invariantes ao LOOP
  - eliminação de código inalcançável
  - alocação ótima de registradores
  - Redução em força (linearização de código)
- Compromisso entre tempo de compilação (gasto em otimização) $\times$ tempo de vida do código, número de execuções do código

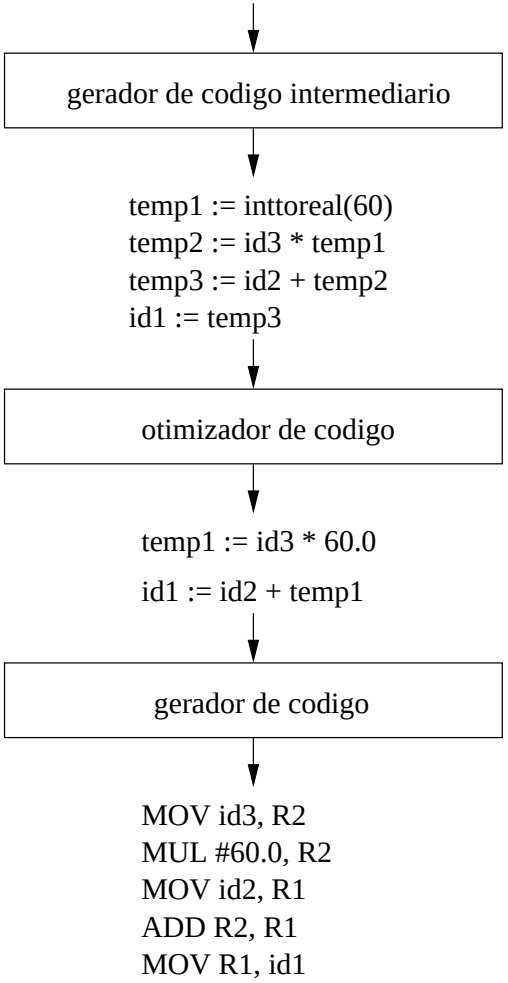
Exemplo


gerador de codigo intermediario
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
otimizador de codigo
temp1 := id3 * 60.0
id1 := id2 + temp1
gerador de codigo
MOV id3, R2
MUL #60.0, R2
MOV id2, R1
ADD R2, R1
MOV R1, id1

Gerador de Código Objeto

- converte o programa fonte (a partir de sua representação em código intermediário) para uma sequência de instruções (assembly ou máquina) de uma máquina real
  - uma nova otimização de código pode ser executada sobre o código final - otimização de código dependente da máquina

Exemplo


gerador de codigo intermediario
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
otimizador de codigo
temp1 := id3 * 60.0
id1 := id2 + temp1
gerador de codigo
MOV id3, R2
MUL #60.0, R2
MOV id2, R1
ADD R2, R1
MOV R1, id1

Bibliografia

- AHO, A.V.; LAM, M.S.; SETHI, R. ULLMAN, J.D., **Compiladores - Princípios, Técnicas e Ferramentas**, 2^a edição, Ed. Addison Wesley, 2008 (capítulo 1 - Introdução)