Compiladores – Introdução

PROFESSOR: DR. DIEGO KASUO NAKATA DA SILVA

Introdução

- Linguagens de programação são notações para se descrever computações para pessoas e máquinas.

- Mas, antes que possa rodar, um programa primeiro precisa ser traduzido para um formato que lhe permita ser executado por um computador.

- Os sistemas de software que fazem essa tradução são denominados ***compiladores.***

1.1 – Processadores de Linguagem

- Um compilador é um programa que recebe como entrada um programa em uma linguagem de programação – a linguagem *fonte* – e o traduz para um programa equivalente em outra linguagem – a linguagem *objeto*. Figura 1.1.

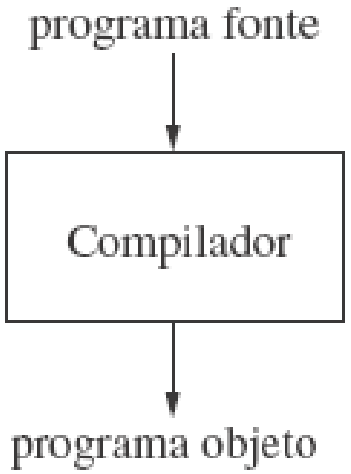


FIGURA 1.1 Um compilador.

- Um papel importante do compilador é relatar quaisquer erros no programa fonte detectados durante esse processo de tradução.

1.1 – Processadores de Linguagem

- Se o programa objeto for um programa em uma linguagem de máquina executável, poderá ser chamado pelo usuário para processar entradas e produzir saída, Figura 1.2.

FIGURA 1.2 Executando o programa objeto.

1.1 – Processadores de Linguagem

- Um *interpretador* é outro tipo comum de processador de linguagem. Em vez de produzir um programa objeto como resultado da tradução, um interpretador executa diretamente as operações especificadas no programa fonte sobre as entradas fornecidas pelo usuário, Figura 1.3.

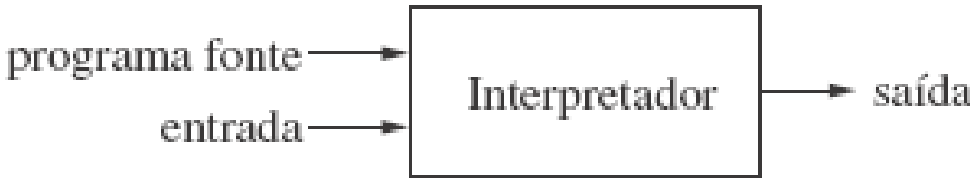


FIGURA 1.3 Um interpretador.

1.1 – Processadores de Linguagem

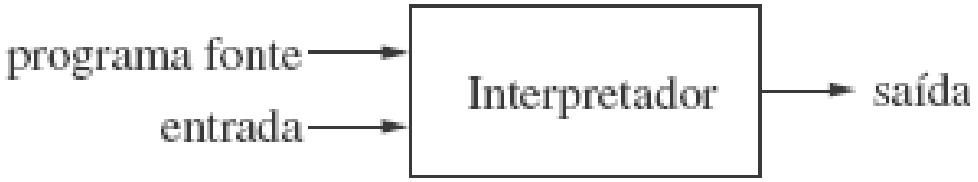


FIGURA 1.3 Um interpretador.

- O programa objeto em linguagem de máquina produzido por um compilador normalmente é muito mais rápido no mapeamento de entradas para saídas do que um interpretador. Porém, um interpretador frequentemente oferece um melhor diagnóstico de erro do que um compilador, pois executa o programa fonte instrução por instrução.

1.1 – Processadores de Linguagem

- Além de um compilador, vários outros programas podem ser necessários para a criação de um programa objeto executável, Figura 1.5.

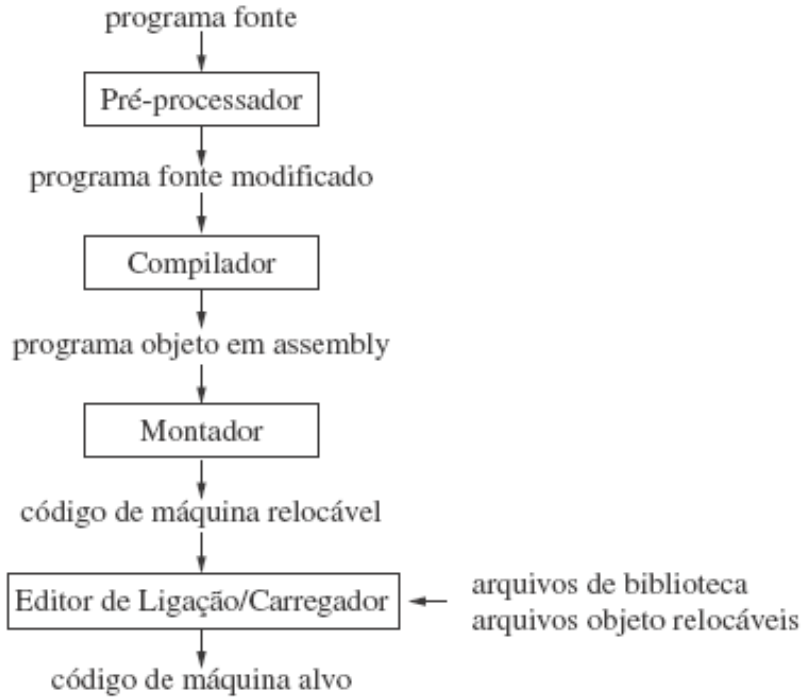


FIGURA 1.5 Um sistema de processamento de linguagem.

1.1 – Processadores de Linguagem

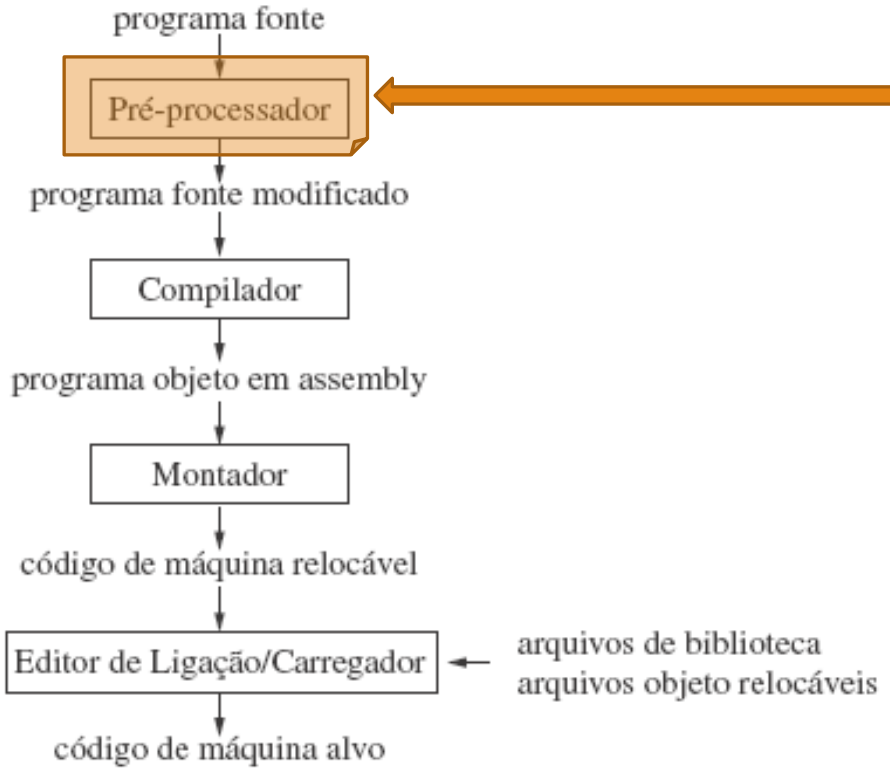


FIGURA 1.5 Um sistema de processamento de linguagem.

- A tarefa de coletar o programa fonte às vezes é confiada a um programa separado, chamado *pré-processador*.

1.1 – Processadores de Linguagem

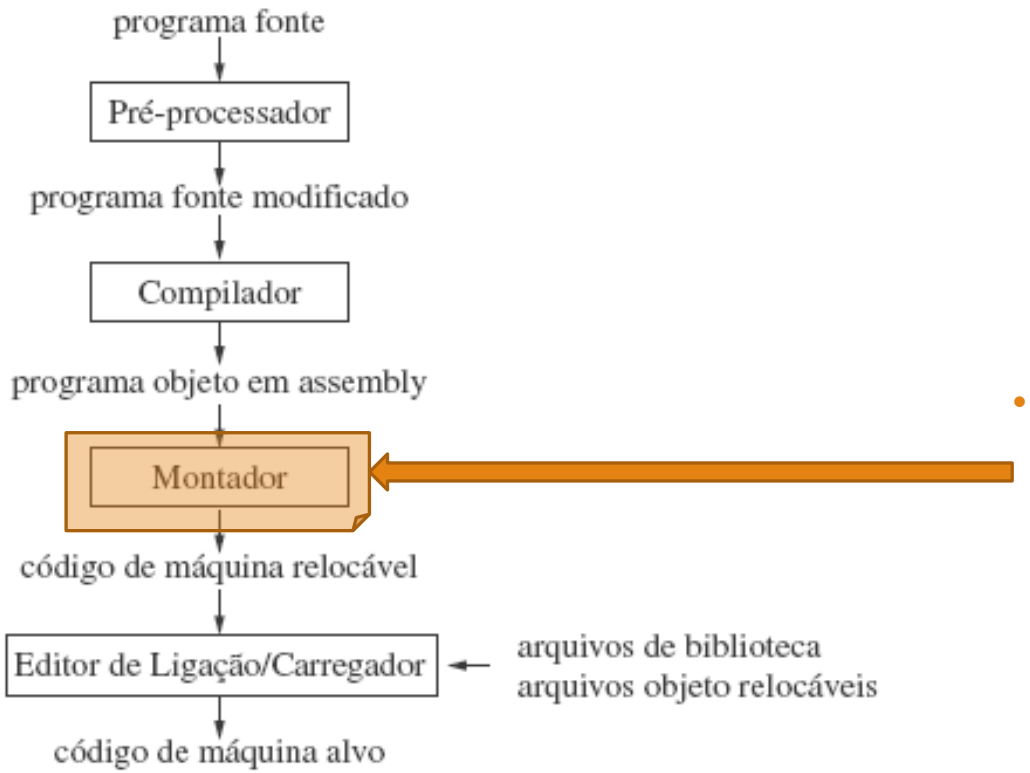


FIGURA 1.5 Um sistema de processamento de linguagem.

- A linguagem simbólica produzida pelo compilador é processada pelo *Montador*, que produz código de máquina realocável como sua saída.

1.1 – Processadores de Linguagem

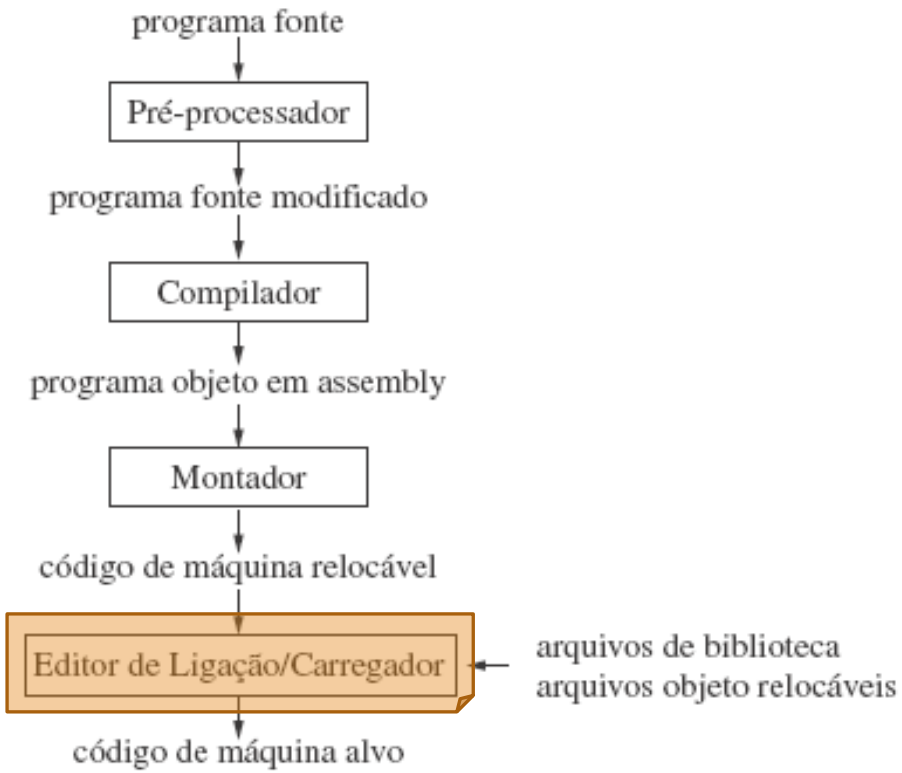


FIGURA 1.5 Um sistema de processamento de linguagem.

- O editor de ligação (*linker*) resolve os endereços de memória externos.
- O carregador (*loader*) reúne todos os arquivos objeto executáveis na memória para a execução.

1.2 – A Estrutura de um Compilador

- Existem duas partes: *Análise* e *Síntese*.
- A parte de *análise* subdivide o programa fonte em partes constituintes e impõe uma estrutura gramatical sobre elas.
- Usa essa estrutura para criar uma representação intermediária do programa fonte.
- Se a parte de análise detectar que o programa fonte está sintaticamente mal formado ou semanticamente incorreto, então ele precisa oferecer mensagens esclarecedoras, de modo que o usuário possa tomar a ação corretiva.
- A parte de análise também coleta informações sobre o programa fonte e as armazena em uma estrutura de dados chamada *tabela de símbolos*.

1.2 – A Estrutura de um Compilador

- Existem duas partes: *Análise* e *Síntese*.
- A parte de *síntese* constrói o programa objeto desejado a partir da representação intermediária e das informações na tabela de símbolos.
- A parte de análise normalmente é chamada de *front-end* do compilador; a parte de síntese é o *back-end.*

- A Figura 1.6, a seguir, exibe a decomposição típica de um compilador em fases.

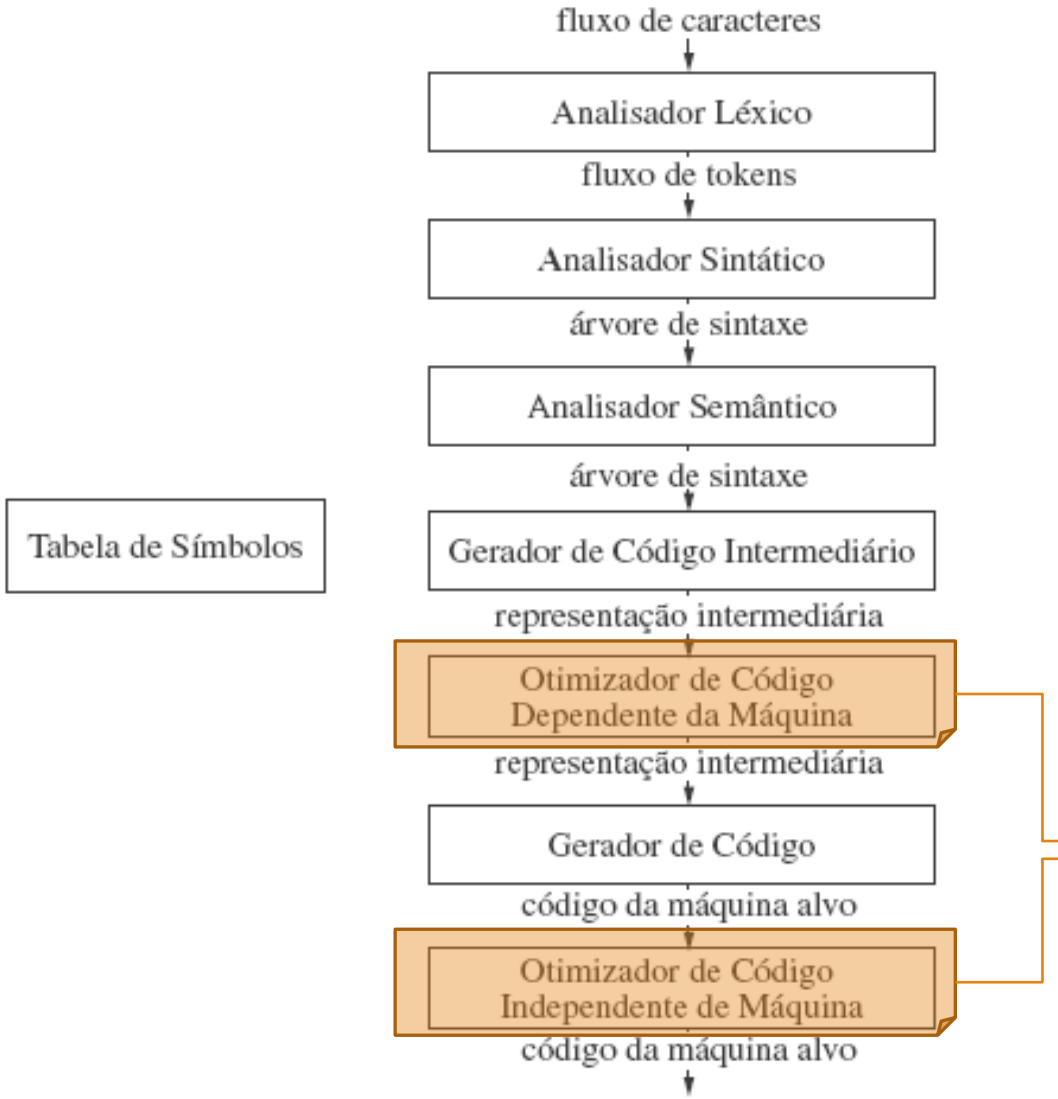


FIGURA 1.6 Fases de um compilador.

- São opcionais.

1.3 – Análise Léxica

- A primeira fase de um compilador é chamada de *análise léxica* ou *leitura* (*scanning*).
- O analisador léxico lê o fluxo de caracteres que compõem o programa fonte e os agrupa em sequências significativas, chamadas *lexemas*.
- Para cada lexema, o analisador léxico produz como saída um token no formato:

- é um símbolo abstrato que é usado durante a análise sintática.
- aponta para uma entrada na tabela de símbolos referente a esse token.

1.3 – Análise Léxica

- Por exemplo, suponha que um programa fonte contenha o comando de atribuição

- Os caracteres nessa atribuição poderiam ser agrupados nos seguintes lexemas e mapeados para os seguintes tokens passados ao analisador sintático:

1. é lexema mapeado em um token , onde é um símbolo abstrato que significa *identificador e* aponta para a entrada da tabela de símbolos onde se encontra . A entrada da tabela de símbolos para um identificador mantém informações sobre o identificador, como seu nome e tipo.

1.3 – Análise Léxica

- Por exemplo, suponha que um programa fonte contenha o comando de atribuição

2. O símbolo de atribuição é um lexema mapeado para o token Como esse token não precisa de um valor de atributo, omitimos o segundo. Poderíamos ter usado qualquer símbolo abstrato, como **atribuir** para o nome do token.

3. é um lexema mapeado para o token onde aponta para a entrada da tabela de símbolos onde se encontra .

1.3 – Análise Léxica

- Por exemplo, suponha que um programa fonte contenha o comando de atribuição

4. é um lexema mapeado para o token

5. é um lexema mapeado para o token onde aponta para a entrada da tabela de símbolos onde se encontra .

6. é um lexema mapeado para o token .

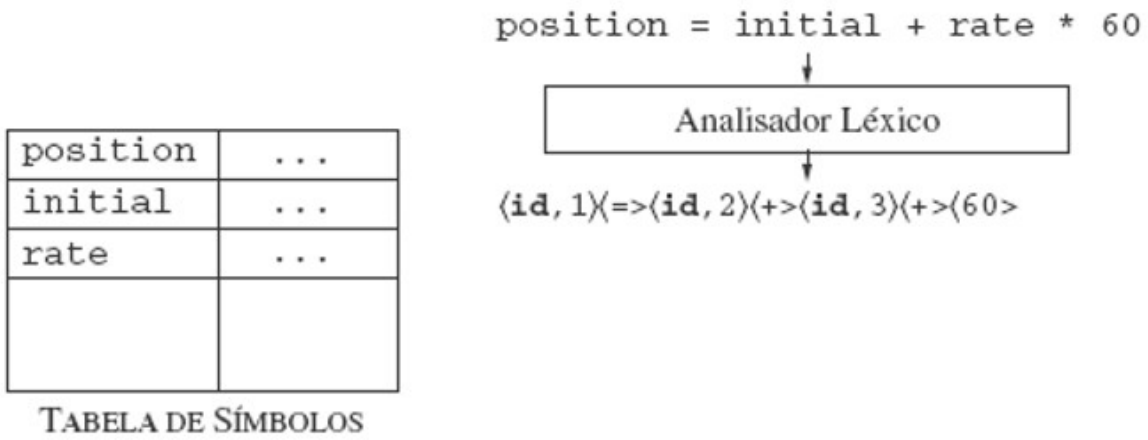
7. é um lexema mapeado para o token .

Os espaços que separam os lexemas são descartados pelo analisador léxico.

1.3 – Análise Léxica

- A Figura a seguir mostra a representação do comando de atribuição após a análise léxica como uma sequência de tokens

- Nessa representação, os nomes de token =, + e são símbolos abstratos para os operadores de atribuição, adição e multiplicação.

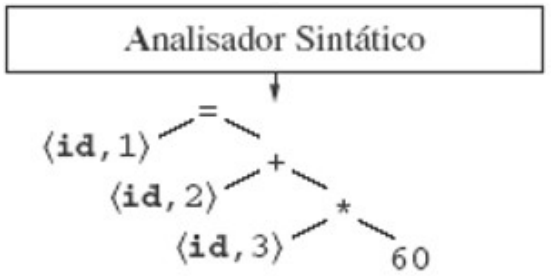


1.4 – Análise Sintática

- A segunda fase do compilador é a *análise sintática.* O analisador sintático utiliza os primeiros componentes dos tokens produzidos pelo analisador léxico para criar uma representação intermediária.
- A uma representação típica é uma *árvore de sintaxe* em que cada nó interior representa uma operação, e os filhos do nó representam os argumentos da operação.

position	...
initial	...
rate	...

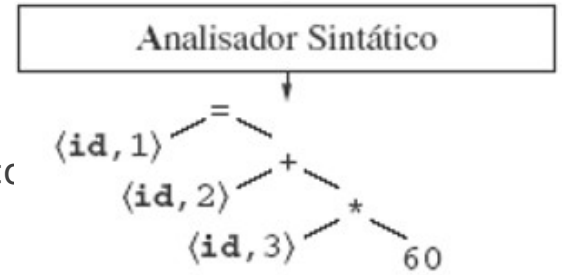
TABELA DE SÍMBOLOS



1.4 – Análise Sintática

position	...
initial	...
rate	...

TABELA DE SÍMBOLOS



- A árvore possu :ulado cc esquerda e o inteiro 60 como seu filho da direita.
- O nó represent
- O nó rotulado com torna explícito que devemos primeiro multiplicar o valor de por 60.
- O nó rotulado com + indica que devemos somar o resultado dessa multiplicação com o valor de .
- A raiz da árvore, rotulada com =, indica que devemos armazenar o resultado dessa adição em uma localização associada ao identificador .

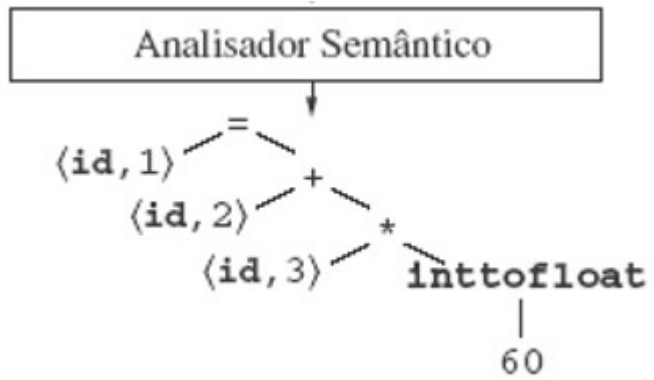
1.5 – Análise Semântica

- O *analisador semântico* utiliza a árvore de sintaxe e as informações na tabela de símbolos para verificar a consistência semântica do programa fonte com a definição da linguagem.
- Uma parte importante da análise semântica é a *verificação de tipo*, em que o compilador verifica se cada operador possui operandos compatíveis.
- Suponha que , e tenham sido declarados como números de ponto flutuante, e que o lexema 60 tenha a forma de um inteiro. O verificador de tipos no analisador semântico descobre que o operador é aplicado a um número de ponto flutuante e a um inteiro 60. Nesse caso, o inteiro pode ser convertido em um numero de ponto flutuante.

1.5 – Análise Semântica

position	...
initial	...
rate	...

TABELA DE SÍMBOLOS



1.6 – Geração de Código Intermediário

- As árvores de sintaxe denotam uma forma de representação intermediária; elas normalmente são usadas durante as análises sintática e semântica.
- Depois das análises sintáticas e semântica do programa fonte, muitos compiladores geram uma representação intermediária explícita de baixo nível ou do tipo linguagem de máquina.
- Essa representação intermediária deve ter duas propriedades importantes: *ser facilmente produzida* e *ser facilmente traduzida para a máquina alvo.*
- Vamos considerar uma forma intermediária, chamada *código de três* endereços, que consiste em uma sequência de instruções do tipo assembler com três operandos por instrução. Cada operando pode atuar como um registrador.

1.6 – Geração de Código Intermediário

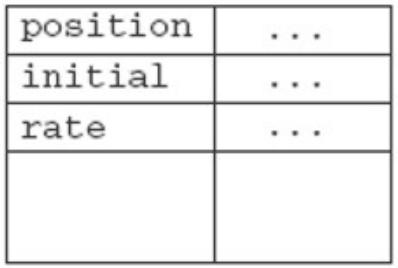
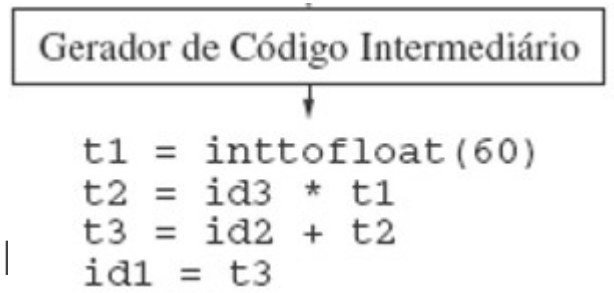


TABELA DE SÍMBOLOS



- Primeiro, [illegible] atri[illegible] ossui no máximo um operador do lado direito. Essas instruções determinam a ordem em que as operações devem ser realizadas.
- Segundo, o compilador precisa gerar um nome temporário para guardar o valor computado por uma instrução de três endereços.
- Terceiro, algumas “instruções de três endereços”, como a primeira e última na sequência da figura, possuem menos de três operandos.

1.7 – Otimização de Código

- A fase de otimização de código faz algumas transformações no código intermediário com o objetivo de produzir um código objeto melhor.
- Melhor significa mais rápido , um código menor ou um código objeto que consuma m

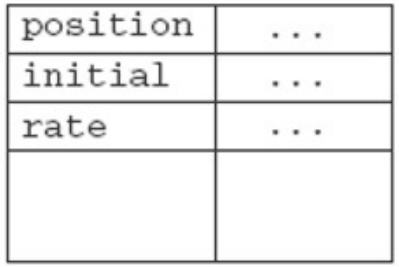
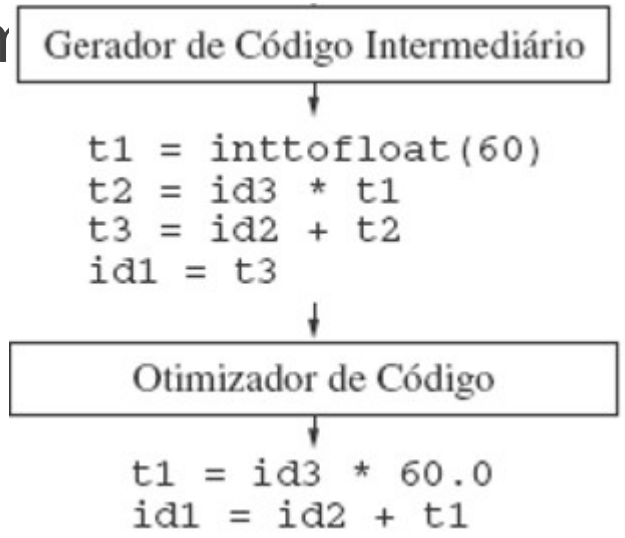
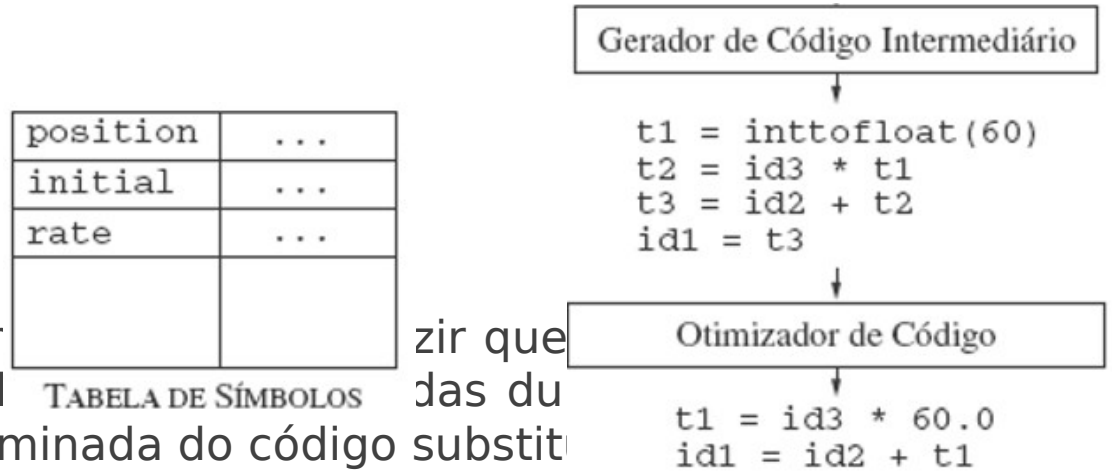


TABELA DE SÍMBOLOS



1.7 – Otimização de Código



- O otim zir que nteiro 60 para ponto flutuante pode ser feita d das du modo que a operação **inttofloat** pode ser eliminada do código substit o número de ponto flutuante 60.
- Além do mais, é usado apenas uma vez na atribuição de seu valor para , portanto o otimizador pode eliminá-lo também.

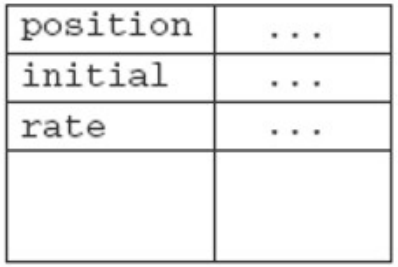
1.8 – Geração de Código

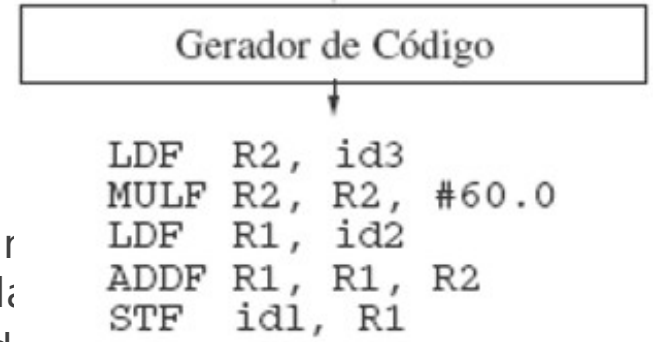
- O gerador de código recebe como entrada uma representação intermediária do programa fonte e o mapeia em uma linguagem objeto.
- Se a linguagem objeto for código de máquina de alguma arquitetura, devem-se selecionar os registradores ou localizações de memória para cada uma das variáveis u[illegible].

position	...
initial	...
rate	...

TABELA DE SÍMBOLOS

1.8 – Geração de Código





- O prime ada ur um destino. O em cada uma das instruçõ anipul
- O código acima carrega o conteúdo do endereço no registrador , depois o multiplica pela constante de ponto flutuante **60.0**. O **#** significa que o valor de **60.0** deve ser tratado como uma constante imediata.
- A terceira instrução move para o registrador , e a quarta o soma com o valor previamente calculado no registrador .
- Finalmente, o valor no registrador é armazenado no endereço de .

1.9 – Gerenciamento da Tabela de Símbolos

- A tabela de símbolos é uma estrutura de dados contendo um registro para cada nome de variável, com campos para os atributos do nome.
- A estrutura de dados deve ser projetada para permitir que o compilador encontre rapidamente o registro para cada nome e armazene ou recupere dados desse registro também rapidamente.

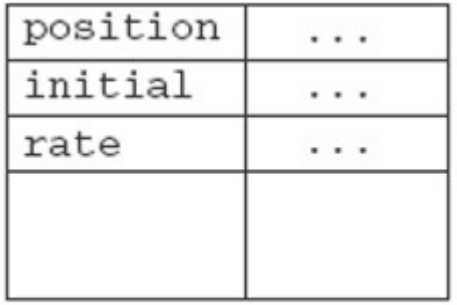
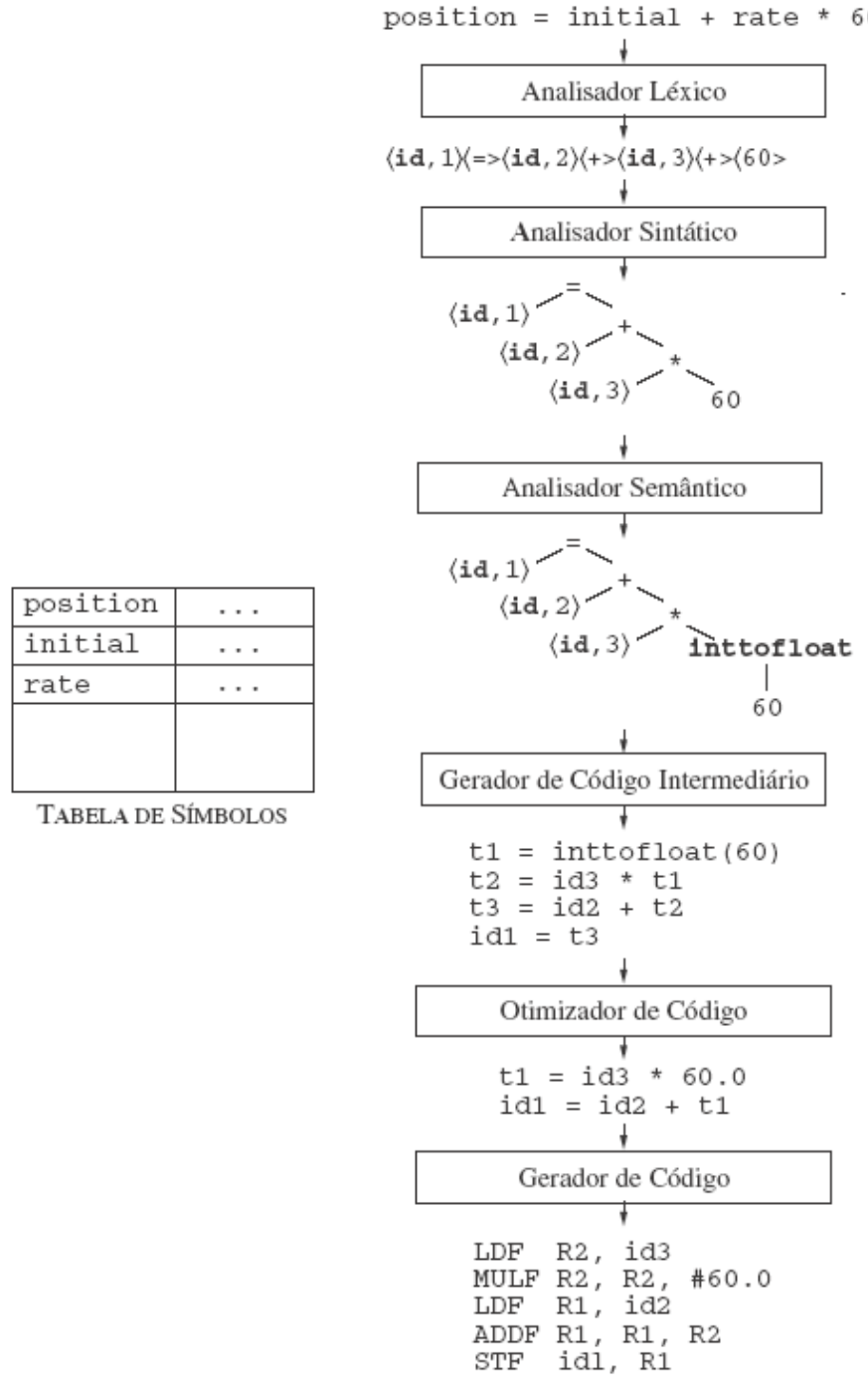


TABELA DE SÍMBOLOS



position	...
initial	...
rate	...

TABELA DE SÍMBOLOS