1.7. Tradutores de Linguagens de Programação: Conceitos Básicos

O objetivo de qualquer linguagem é a comunicação entre duas partes, um emissor e um receptor. Em LP, a comunicação ocorre entre um programador e um programa tradutor. O objetivo deste programa tradutor é aceitar um conjunto de instruções escritas em uma linguagem de programação de alto nível, que é independente da máquina, e fazer com que as atividades especificadas por estas instruções sejam executadas pelo computador. Em outras palavras, como os computadores só podem executar programas escritos em linguagem de máquina, programas escritos em linguagem de alto nível devem ser traduzidos para versões equivalentes em linguagem de máquina, antes de serem executados [DER 90, WAT 90].

Existem dois tipos fundamentais de tradutores: interpretadores e compiladores. No caso de um interpretador, as instruções definidas na linguagem de alto nível são executadas diretamente. Ele traduz um comando de um programa de cada vez e então chama uma rotina para completar a execução do comando, como ilustrado na figura 1.12. Mais precisamente, um interpretador é um programa que executa repetidamente a seguinte seqüência:

- Pega a próxima instrução;
- Determina as ações a serem executadas;
- Executa estas ações [DER 90, GHE 97].

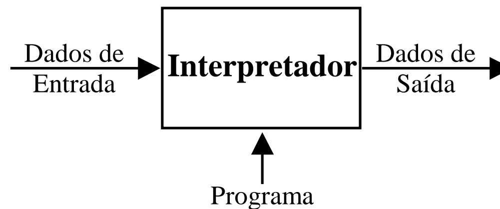

Figura 1.12 – Processo de interpretação [GHE 97]

Já um compilador produz a partir do programa de entrada, outro programa que é equivalente ao original, porém numa linguagem que é executável. Este programa resultante pode ser em uma linguagem que é diretamente executável, tal como linguagem de máquina, ou indiretamente executável, tal como outra linguagem para a qual já existe um tradutor.

Cada um destes processos tem suas vantagens e desvantagens. Interpretação, apesar de ter um tempo maior de execução, tem a vantagem de não traduzir instruções que nunca são executadas e de conseguir voltar à instrução correspondente na LP a partir de qualquer ponto da execução. O compilador, por outro lado, precisa traduzir cada instrução somente uma vez, independente de quantas vezes a instrução é executada. Isto aplica-se tanto no caso de iteração como no caso de execuções repetidas do mesmo programa. As vantagens de um compilador em geral superam as do interpretador na prática, o que faz com que esta forma de tradução seja uma das mais usadas. Por esta razão, e porque a compilação é um processo mais complexo, as atividades de um compilador serão detalhadamente descritas [DER 90].

O objetivo de um compilador é traduzir um programa escrito em uma linguagem fonte em um programa equivalente expresso em uma linguagem que executável diretamente pela máquina. Estes dois programas são chamados **programa fonte** (ou código fonte) e **programa objeto** (ou código objeto). A linguagem do programa objeto é chamada de linguagem *target*. A figura 1.13 mostra uma visão do processo de compilação onde o programa objeto é executado diretamente. O tempo durante o qual o compilador está "trabalhando", isto é, realizando a conversão entre código-fonte e código-objeto, é chamado de tempo de compilação. O tempo durante o qual o programa objeto é executado é chamado tempo de execução.

A compilação pode ser dividida em várias fases. Estas fases são conceituais e especificam atividades que todos os compiladores executam, embora freqüentemente as atividades de várias fases possam ser combinadas e executadas simultaneamente. Estas fases estão resumidamente ilustradas na figura 1.14.

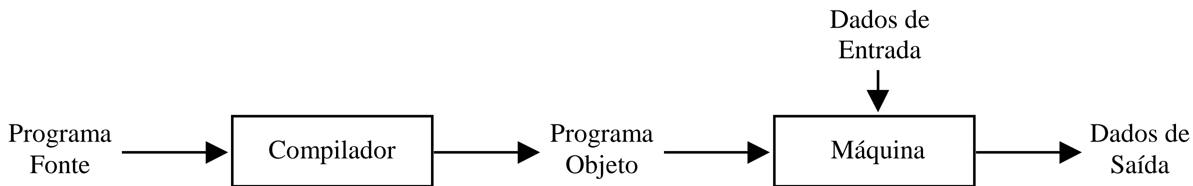

Figura 1.13 - Processo de compilação [DER 90]

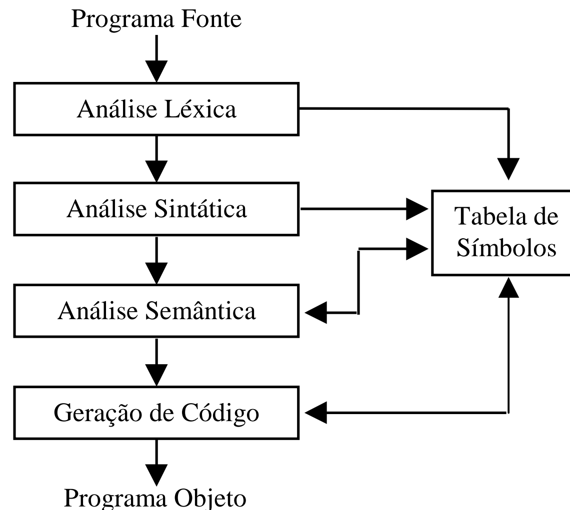

Figura 1.14 - Fases da compilação

As fases apresentadas no modelo da figura 1.15 possuem, resumidamente, as seguintes funções:

- Tabela de Símbolos: contém uma entrada para cada símbolo definido pelo usuário ou identificador incluído no programa fonte. Possui uma importância fundamental, uma vez que é montada durante a análise do programa fonte, com informações sobre declarações de variáveis, declarações dos procedimentos ou sub-rotinas, parâmetros de sub-rotinas, etc., e é usada para comunicação entre fases.
- Análise Léxica: tem como função identificar seqüências de caracteres de entrada e produzir como saída uma seqüência de elementos, os *tokens*. O analisador léxico lê caracter a caracter do texto fonte verificando se os caracteres lidos pertencem ao alfabeto da linguagem, identificando *tokens*, desprezando comentários e brancos, e convertendo caracteres para minúsculas. Os *tokens*, menores elementos que contém informação em uma linguagem, constituem classes de símbolos, tais como, palavras reservadas, delimitadores, identificadores, etc.
- Análise Sintática: consiste no processo de identificar seqüências de símbolos que constituem estruturas sintáticas (por exemplo, expressões, comandos), através de uma varredura, ou *parsing*, da representação interna (cadeia de *tokens*) do programa fonte. O analisador sintático produz (explícita ou implicitamente) uma estrutura em árvore, chamada árvore de derivação, que exibe a estrutura sintática do texto fonte, resultante da aplicação das regras gramaticais da linguagem
- Análise Semântica: tem como principal atividade determinar se as estruturas sintáticas analisadas fazem sentido, ou seja, se um identificador declarado como variável é usado como tal; se existe compatibilidade entre operandos e operadores em expressões; etc. Resumindo, o analisador semântico verifica se o programa não possui erros de significado.
- Geração de Código: utiliza a representação interna produzida pelo analisador sintático e gera como saída uma seqüência de código objeto. Também é responsável pela reserva de memória para dados e variáveis, geração de código para acessar tais posições, seleção de registradores, etc.

Torna-se interessante comentar que o compilador também possui uma fase de otimização, que refere-se ao processo de transformar o programa em um programa equivalente que resultará numa execução mais eficiente. Esta modificação pode ser feita no programa fonte, no programa objeto ou em qualquer outra forma do programa durante o processo de compilação. O objetivo é construir um programa que executará mais rápido ao mesmo tempo que mantém a funcionalidade do programa fonte original [DER 90].

Para geração do código executável final (código que pode ser executado pelo sistema operacional), entretanto, existem outros passos além da compilação. A figura 1.15 ilustra o processo de geração do código executável. No primeiro passo, o **pré-processador** mapea instruções escritas numa linguagem de alto nível estendida, para instruções da linguagem de programação original. Entre as funções que ele pode realizar incluem-se: processamento de macros (as evocações a macro-rotinas são traduzidas para o código original da macro); inclusão de arquivos (referências a arquivos são substituídas pelo próprio arquivo); racionalização (substituição de código não oferecido pelo compilador por código equivalente suportado por ele); e extensão da linguagem (suporte a novos aspectos). O **compilador** analisa o código-fonte e o converte para um código-*assembly* (versão mnemônica da linguagem de máquina). O **montador** traduz o código *assembly* para código de máquina (código objeto). Porém, esta forma é intermediária, não podendo ser lida pelo programador, nem executada pelo computador. Este código pode ser relocável (carregável para a execução em qualquer posição de memória) ou absoluto (carregável a partir de uma posição absoluta). Finalmente, o **carregador/ligador** executa duas tarefas: carregar, que consiste em tomar o código relocável, alterar os endereços necessários e colocar o código e os dados na localização de memória adequada; e ligar, que consiste em "juntar" o código objeto com as bibliotecas necessárias para gerar o programa executável. O tempo após a ativação do programa executável é chamado tempo de execução [HAN 86].

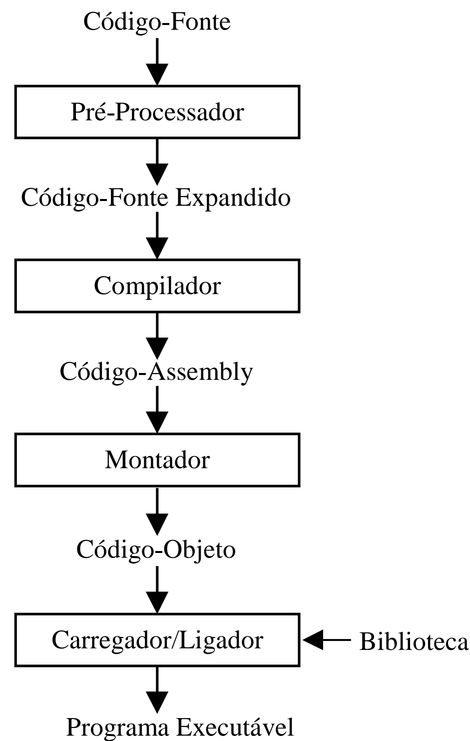

Figura 1.15 - Processo de geração do código executável