# Sumário

[Sumário 1](#_Toc239179908)

[Capítulo 1 Introdução 2](#_Toc239179909)

[1.1 Objetivos 4](#_Toc239179910)

[1.1.1 Objetivos Gerais 4](#_Toc239179911)

[1.1.2 Objetivos Específicos 4](#_Toc239179912)

[1.2 Motivação 5](#_Toc239179913)

[1.3 Fases de Compilação 5](#_Toc239179914)

[1.4 Limitações 7](#_Toc239179915)

[1.5 História 7](#_Toc239179916)

[1.6 Desafios na construção de um compilador para uma liguagem de montagem alvo 8](#_Toc239179917)

# Capítulo 1 Introdução

Os compiladores surgiram com o objetivo de transformar um programa escrito em uma linguagem de alto nível, ou seja, uma linguagem próxima da linguagem natural em um programa escrito em linguagem de máquina. A linguagem de máquina é uma linguagem que os computadores entendem, ou seja, computadores só executam instruções que estirem nessa linguagem. Entretanto, para nós humanos, escrever um programa nessa linguagem  
é uma tarefa tediosa, propensa a erros e improdutiva. Desta forma, passou-se  a criar linguagens de programação de alto nível, que são mais produtivas e mais intuitivas para os programadores. Com o surgimento, das linguagens  
de alto nível surgiu também à necessidade da construção de compiladores.

Um compilador é um programa que aceita como entrada um código de programa numa certa linguagem e produz como saída um texto de programa em outra linguagem, preservando o significado do código de entrada. Esse processo seria chamado de tradução se os textos estivessem em linguagens naturais. Quase todos os compiladores fazem a tradução de uma linguagem de entrada, a linguagem-fonte, para apenas uma linguagem de saída, a linguagem de destino. Nesse projeto, a linguagem de origem é uma Linguagem Imperativa, como C, e a linguagem de destino é código de máquina para microprocessadores Intel 8088. A linguagem na qual o próprio compilador é escrito é chamada linguagem de implementação, a linguagem de implementação adotada é C, com a utilização das ferramentas Lex (analisador léxico) e Yacc (analisador sintático).

O objetivo da construção de um compilador cuja linguagem de destino seja código de máquina é o fato de haver hardware no qual é possível “executar” o programa traduzido ou, mais precisamente, fazer o hardware executar as ações descritas pela semântica do programa. Pois, o hardware é a única fonte real de poder de computação. A execução de um programa traduzido muitas vezes envolve alimentá-lo com dados de entrada em algum formato, e possivelmente resultará em dados de saída em algum outro formato.

Para obter o programa traduzido executamos o compilador, que é simplesmente outro programa cujos dados de entrada são constituídos por um arquivo com o formato de um texto de origem de programa cujos dados de entrada são constituídos por um arquivo com o formato de um texto de origem de programa e cujos dados de saída constituem um arquivo com o formato de código executável.

Um aspecto claro da compilação é que a entrada tem uma propriedade chamada semântica – seu “significado” - que deve ser preservado pelo processo e com freqüência é menos claramente identificável em um programa de conversão de arquivos tradicional.

A parte de um compilador que executa a análise do texto da linguagem-fonte é chamada front-end, e a parte que faz a síntese da linguagem de destino é o back-end. Se o compilador tiver um projeto muito limpo, o front-end irá ignorar totalmente a linguagem de destino e o back-end irá ignorar totalmente a linguagem-fonte: o único fator que eles terão em comum será o conhecimento da representação semântica.

Não há nenhuma diferença fundamental entre usar um compilador e usar um interpretador. Em ambos os casos, o texto do programa é processado numa forma intermediária, a qual é então interpretada por algum mecanismo de interpretação.

Na compilação:

* O processamento de programas é considerável;
* A forma intermediária, o código binário executável, é de baixo nível;
* O mecanismo de interpretação é a CPU do hardware;
* A execução do programa é relativamente rápida.

Na interpretação:

* O processamento de programas é de mínimo a moderado;
* A forma intermediária, alguma estrutura de dados, é de nível alto a médio;
* O mecanismo de interpretação é um programa;
* A execução de programas é relativamente lenta.

Os compiladores podem diferir consideravelmente em sua arquitetura. Duas questões arquiteturais são levantadas, qual a largura do compilador, granularidade dos dados que são repassados entre os módulos do compilador, e outra se refere ao fluxo de controle entre os módulos do compilador.

Um compilador consiste em uma série de módulos que transformam, refinam e repassam informações entre eles. Duas escolhas razoáveis para o tamanho dos blocos de informações são a menor unidade que tem significado entre os dois módulos e o programa inteiro, podendo classificá-los em:

* Compilador Estreito, que lê uma pequena parte do programa, em geral alguns símbolos, processa as informações obtidas produzindo alguns bytes de código-objeto e possivelmente descartar a maior parte das informações sobre esses símbolos e repete o processo até ser alcançado o fim do texto de programa.
* Compilador Largo, que lê o programa inteiro e aplica uma série de transformações a ele que resultam eventualmente No código-objeto desejado.

História:

Podemos distinguir três períodos na história da construção de compiladores, como a seguir:  
    - de 1945 a 1960 (geração de código): O principal problema era como gerar código para uma determinada máquina;  
    - de 1960 a 1975 (análise): Esses anos presenciaram a proliferação de diversas línguas e os projetistas de linguagens começaram a acreditar que ter um compilador para a nova linguagem rapidamente era mais importante que ter um compilador que gerasse código eficiente.  
    - de 1975 até a atualidade (otimização de código e paradigmas): Tanto o número de linguagens novas quanto o números de tipos de máquinas diferentes em uso regular diminuíram, o que reduziu a necessidade de compiladores rápidos e simples/rápidos e sujos para novas linguagens e/ou máquinas. Além disso, ao mesmo tempo foram desenvolvidos novos paradigmas de programação, tendo a programação funcional, lógica e distribuída como os exemplos mais proeminentes.

## Objetivos

Abaixo serão descritos os principais objetivos gerais e específicos propostos no nosso trabalho.

### Objetivos Gerais

O objetivo geral deste trabalho é, implementar um compilador, por meio de um projeto de construção de um compilador simples mas funcional, os conceitos aprendidos sobre a Teoria de Compiladores. Além de aplicar os conceitos e técnicas, utilizaremos ferramentas de desenvolvimento de compiladores como o Lex e o Yacc.O projeto é construir um compilador para a linguagem Portugol. O Portugol é uma linguagem de programação imperativa semelhante a linguagem C. O Portugol contém comandos e declarações na liguagem nativa de seus criadores, no caso o português. A linguagem alvo do compilador é a linguagem de montagem das máquinas baseadas na arquitetura Intel 8088.

### Objetivos Específicos

Um compilador após o término da construção deve satisfazer idealmente algumas propriedades. É claro que a propriedade mais importante de um bom compilador é gerar código correto. Também é desejável que um compilador obedeça completamente à especificação de linguagem. Outra propriedade a ser alcançada é que ele deve ser capaz de manipular programas que são de tamanho essencialmente arbitrário. Outro objetivo é a construção limpa de um compilador através de uma descrição semântica da linguagem que será comum ao front-end e ao back-end.

## Motivação

Há uma série de motivos pela qual devemos estudar a construção de compiladores, uma delas é que a construção de compiladores é bem-sucedida na Ciência da Computação. Algumas razões para isso são a estruturação apropriada para o problema, o uso criterioso de formalismos e o uso de ferramentas sempre que possível.

Os compiladores estruturam o sistema analisando sua entrada, construindo uma representação semântica e sintetizando sua saída a partir dela. Para algumas partes da construção do compilador foram desenvolvidos excelentes formalismos padronizados, os quais reduzem bastante o esforço para produzir essas partes, como as expressões regulares e as gramáticas livres de contexto, usadas em análise léxica e sintática. Tendo um formalismo criado podemos gerar um programa usando um gerador de programas

Outro ponto é que as técnicas de construção de compiladores podem ser e são aplicadas fora da construção de compiladores em seu sentido mais estrito. O estudo e a construção de compiladores se concentram nas estruturas de dados e algoritmos que os compiladores contêm. Por exemplo, hash, tabelas pré-calculadas, mecanismos de pilha, coleta de lixo, programação dinâmica e os algoritmos de grafos.

## Fases de Compilação

Podemos pensar no processo de compilação como uma seqüência de fases, onde cada fase recebe como entrada o resultado da fase anterior. A entrada inicial é o programa fonte e primeira fase transforma esse programa em uma representação que possa servir de entrada para a segunda fase. Desta forma, a cada passagem de uma fase para outra existe uma representação do programa fonte. Na prática, entretanto, algumas das fases podem ser agrupadas e a representação intermediária não precisa ser construída explicitamente. A Fig. 1.0 mostra as fases de compilação.

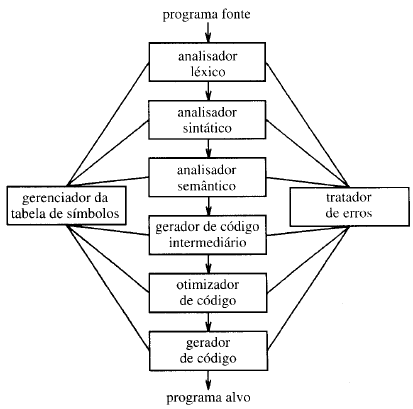


Fig. 1.0 Fases da compilação

As três primeiras fases formam a parte de análise da compilação. A análise quebra o programa fonte em suas partes constituintes e cria uma representação intermediária para ele. A fase de análise léxica lê os caracteres do programa fonte e os agrupa em um fluxo de tokens. Um token representa um seqüência de caracteres, logicamente coesiva como ,por exemplo, um identificador, ou uma palavra chave. Assim, a primeira fase transforma o programe fonte em uma seqüência de tokens. Em seguida, a  fase de análise sintática coloca esses tokens em uma estrutura hierárquica que podemos representar com uma árvore sintática. Por último, a fase de análise semântica verifica os erros semânticos no programa fonte e captura informações de tipo de dados para a fase de geração de código.

Após  a fase de análise semântica, alguns compiladores geram uma representação explícita do programa fonte. A representação intermediária pode ter uma variedade de formas, mas deveria atender a dois requisitos: ser fácil de produzir e ser fácil de traduzir no programa alvo. Desta forma, a representação intermediária funciona como um código gerado para uma máquina abstrata cuja linguagem é mais simples que a linguagem de montagem de uma máquina real, facilitando, assim, a fase seguinte de geração de código.

A fase de otimização tenta melhorar o código intermediário, com o objetivo de tornar o código mais rápido em tempo de execução. É uma fase importante porque tem uma participação significativa sobre o desempenho de um programa. Entretanto, essa fase não é essencial para o processo de tradução. Esta fase simplesmente está presente na compilação porque geralmente desejamos que o código gerado pelo compilador possua um bom desempenho.

A fase final de compilador é a geração do código alvo, normalmente, uma linguagem de montagem. Nesta fase, as localizações de memória são selecionadas para cada uma das variáveis usadas pelo programa. Em seguida, as instruções intermediárias são, cada uma, traduzidas numa seqüência de instruções de máquina que realizam a mesma tarefa.

## Limitações

Como se pode observar, a construção de um compilador não é uma tarefa trivial. O projeto de um compilador necessita do conhecimento de várias áreas como linguagens de programação, teoria da computação, semântica, estrutura de dados,  
algoritmos, arquitetura de computadores  e engenharia de software.  
Dessa forma, a construção de compilador pode exigir especialistas em cada uma dessas áreas e de processo de software bem definido que guie as várias etapas  
do seu desenvolvimento.

## História

Podemos distinguir três períodos na história da construção de compiladores, como a seguir:

- de 1945 a 1960 (geração de código):    O principal problema  
era como gerar código para uma determinada máquina;

- de 1960 a 1975 (análise): Esses anos presenciaram a proliferação de diversas línguas e os projetistas de linguagens começaram a acreditar que ter um compilador para a nova linguagem rapidamente era mais importante que ter um compilador que gerasse código eficiente.

- de 1975 até a atualidade (otimização de código e paradigmas): Tanto o número de linguagens novas quanto o números de tipos de máquinas diferentes em uso regular diminuíram, o que reduziu a necessidade de compiladores rápidos e simples/rápidos e sujos para novas linguagens e/ou máquinas. Além disso, ao mesmo tempo foram desenvolvidos novos paradigmas de programação, tendo a programação funcional, lógica e distribuída como os exemplos mais proeminentes.

## Desafios na construção de um compilador para uma liguagem de montagem alvo

A produção de compiladores que produzem um programa em linguagem assembly como saída torna o processo de geração de código bem mais fácil. Podemos gerar instruções simbólicas e utilizar as facilidades de macro do Montador para auxiliar na geração de código. Contudo, é necessária mais uma etapa no processo de compilação, que consiste em chamar o montador para gerar o código de máquina.

O gerador de código precisa mapear o programa na Representação Intermediária (RI) em uma seqüência de código que possa ser executada pela arquitetura alvo. A complexidade da realização desse mapeamento é determinada por fatores como: O nível da RI, a natureza da arquitetura do conjunto de instruções alvo e a qualidade do código gerado. Se a RI for de alto nível, o gerador de código pode traduzir cada comando da RI em uma seqüência de instruções de máquina usando casamento de códigos. Entretanto, essa geração de código comando por comando produz um código redundante, que exige otimização adicional. No entanto, se a RI refletir alguns dos detalhes de baixo nível da máquina alvo subjacente, o gerador de código pode usar essa informação para gerar seqüências de códigos mais eficientes.

A natureza do conjunto de instruções da máquina alvo tem forte efeito sobre a dificuldade da seleção de instruções. Por exemplo, a uniformidade e a completeza do conjunto de instruções são fatores importantes. Se a máquina alvo não prover cada tipo de dado de maneira uniforme, cada exceção à regra exigirá tratamento especial. Na maioria das máquinas, um programa em dada RI pode ser escrito em diversas seqüências de códigos diferentes, com diferenças significativas de custo entre as diferentes implementações. Uma tradução ingênua do código intermediário pode produzir um código correto, porém ineficiente. Por exemplo, se a máquina alvo possuir uma instrução INC, o comando de três endereços, a = a + 1 pode ser implementado de modo mais eficiente por uma única instrução INC a, em vez de uma seqüência mais óbvia que executa em três passos.

Outro problema importante na geração de código é decidir que valores residirão em registradores e em quais registradores. Os registradores são considerados as estruturas computacionais mais velozes da máquina alvo, mas usualmente arquiteturas, como a do 8088, possuem um número reduzido de registradores,portanto, insuficiente para armazenar todas as variáveis de um programa. Os valores não mantidos nos registradores precisam residir na memória. As instruções envolvendo operandos em registradores são invariavelmente mais rápidas do que aquelas envolvendo operandos na memória, assim, a utilização eficiente de registradores é particularmente importante. O uso de registradores freqüentemente é subdividido em dois subproblemas:

- Alocação de Registradores, etapa na qual decidimos o conjunto de variáveis re residirão nos registradores em cada ponto do programa.

- Atribuição de registradores, etapa na qual determinamos um registrador específico em que uma variável residirá.

É difícil encontrar uma atribuição ótima de registradores para as variáveis, até mesmo com máquinas de um único registrador. Matematicamente, o problema é NP completo.