Compiladores

Fases da Compilação

Introdução

Este material contém as notas de aula sobre *Fases da Compilação*, da disciplina *Compiladores*. O objetivo é aprofundar os conhecimentos sobre compiladores detalhando ainda mais este software cujos conceitos introdutórios já foram apresentados nas notas de aula de Introdução. Para o aproveitamento completo do conteúdo sugerido neste material, sugere-se que o leitor tenha acesso à obra de Cooper (2014).

Fases da Compilação

Observando a compilação em detalhes, observa-se que o processo possui duas ou três fases. No modelo de duas fases, a compilação se divide em fases de **análise** (*front-end*) e **síntese** (*back-end*), como mostra a Figura 1.

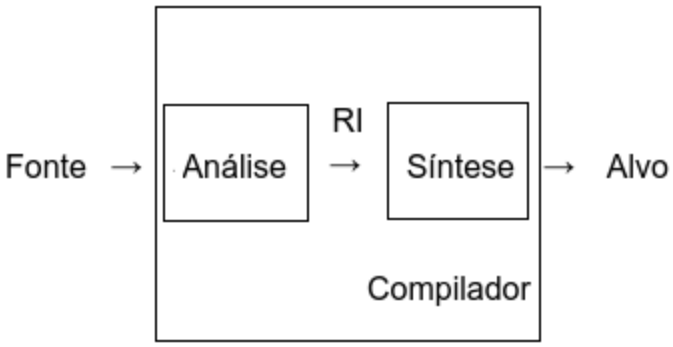

Figura 1. Compilador de duas fases.

No modelo de três fases, a compilação é composta, em ordem, por **análise** (*front-end*),

otimização e **síntese** (*back-end*), como mostra a Figura 2.

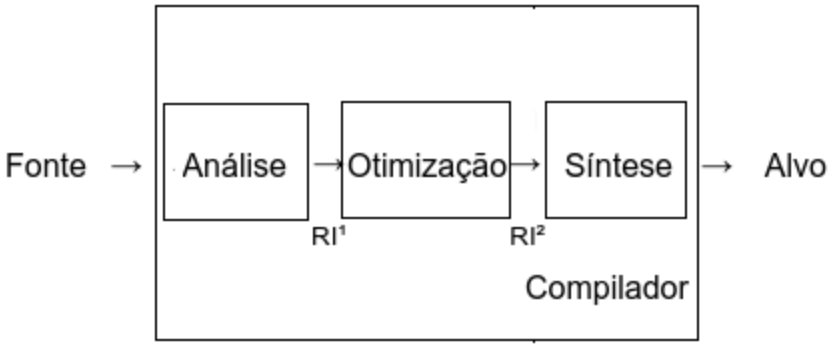

Figura 2. Compilador de três fases.

Mas em que consistem essas três fases? A sequência do texto as resume!

Análise

A fase de **análise**:

- É considerada o *front-end* (a linguagem interface com a qual se interage).
- Lida com a linguagem de origem.
- Determina se o código de entrada está bem formado em termos de sintaxe e significado (semântica).
- Em termos de sintaxe, verifica se o conjunto de *strings* constantes no código fonte segue as regras de definição da gramática formal desta linguagem. Isso é feito de forma mais geral atribuindo os termos do programa a classes gramaticais/categorias. Se descobrir que o código é válido, cria uma representação deste código na representação intermediária do compilador. Se descobrir que o código é inválido, informa ao usuário com mensagens de erro de diagnóstico para identificar os problemas com o código.
- Possui os componentes *scanner* e *parser*, que determinam se o código de entrada é ou não, de fato, um membro do conjunto de programas válidos definidos pela gramática (COOPER, 2014). O scanner é o analisador léxico, o parser é o analisador sintático.
- É composta pelas análises léxica, sintática e semântica.
- Retorna um mapeamento do código fonte em uma Representação Intermediária – RI, que pode ser:
    - um grafo (uma árvore sintática abstrata).

- um código de montagem (*assembly*) sequencial
- uma notação pós-fixa de instruções de máquina
- um código de três endereços
- (...)

O exemplo a seguir demonstra a RI com código de três endereços para a expressão **`x := y + tmp * 30`**:

```
t1 := intToFloat(30);
t2 := tmp * t1;
t3 := y + t2;
x := t3;
```

O exemplo a seguir demonstra a RI com notação pós-fixa para a expressão **`2 + x`**:

```
2 x +
PUSH 2
PUSH x
ADD
```

Além do que já foi dito, representações intermediárias costumam ser fáceis de gerar, otimizar e converter para código alvo. Elas também são *portáveis*, uma vez que o código ainda é independente de arquitetura e pode, portanto, ser traduzido para diferentes arquiteturas.

Otimização

Sobre a fase de **otimização, destaca-se**:

- Esta fase não está presente em compiladores de duas fases, embora possa haver, internamente, algum mecanismo de aprimoramento em alguma fase da compilação.
- Caso exista a fase de otimização, a representação intermediária gerada na análise pode ser refinada *n* vezes, até gerar outra representação intermediária otimizada, utilizada como entrada para a etapa de síntese. Portanto, o otimizador é um tradutor de uma RI para outra RI (ou para a mesma, mas com alguma alteração relevante), ou seja, o próprio otimizador é um compilador (COOPER, 2014).

- O otimizador analisa o formato da RI para descobrir fatos sobre esse contexto, e, a partir disso, usa esse conhecimento contextual para reescrever o código de modo que calcule a mesma resposta de maneira mais eficiente[1] (Cooper, 2014).

Um exemplo de otimização em linguagem C, adaptado de Cooper (2014), está disponível no quadro abaixo:

```
#include <stdio.h>

int main() {

    long int a = 1;
    long int b = 3;
    long int c = 10;
    long int d;
    long int n;
    long int i;

    scanf("%ld", &n);

    for (i = 1; i < n; i++) {
        scanf("%ld", &d);

        a = a * 2 * b * c * d;
        printf("O valor de a é %ld\n", a);
    }

    return 0;
}
```

É possível notar, observando o código, que os valores das variáveis `b` e `c` não são alterados no laço de repetição. Além disso, há um valor constante, o número `2`. Todos esses valores podem

[1] Eficiência, nesse contexto, pode significar menor custo energético, menor código compilado, menor tempo de execução entre outros.

ser multiplicados antes do laço e reaproveitados. Se o otimizador perceber isso, o código pode ser melhorado.

Uma versão equivalente, mas otimizada, está disponível no quadro a seguir:

```
#include <stdio.h>

int main() {

    long int a = 1;
    long int b = 3;
    long int c = 10;
    long int d;
    long int n;

    scanf("%ld", &n);

    long int t = 2 * b * c;

    while (--n) {
        scanf("%ld", &d);

        a = a * t * d;
        printf("O valor de a é %ld\n", a);
    }

    return 0;
}
```

A remoção da variável `i` se deu apenas porque o controle poderia ser feito com a variável `n`.

O primeiro código, não otimizado, precisaria realizar quatro multiplicações durante *n* iterações, totalizando *4n* multiplicações. Já o segundo código, otimizado, faz duas multiplicações antes do

loop, depois mais duas em cada uma das *n* iterações, totalizando *2n + 2* multiplicações. Assim, para *n=1* os códigos fazem o mesmo número de multiplicações, mas para *n=2000*, por exemplo, o primeiro código faz *8000* multiplicações e o segundo faz *4002*.

PERGUNTAS

Seria possível

1. reaproveitar a variável `b`, evitando o uso da variável `t`? Como?
2. substituir diretamente o valor `2*b*c` por `60` na fórmula? E se os valores de `b` e `c` fossem informados pelo usuário?
3. substituir a atribuição da variável `t` por `t=b*c; t+=t;`? Qual é a diferença?

Retornando à expressão **`x := y + tmp * 30`**, utilizada em exemplo anterior para conversão em código de três endereços, poder-se-ia otimizar o código anteriormente apresentado da seguinte forma:

```
t1 := tmp * 30.0;
x := y + t1;
```

Diferentemente do primeiro exemplo apresentado, ainda não otimizado, neste exemplo somente uma variável temporária foi declarada, assim como somente duas operações de atribuição foram necessárias, contra quatro do primeiro exemplo.

Síntese

A fase de **síntese**:

- É considerada o *back-end*.
- Lida com a linguagem alvo.
- Recebe um código intermediário (otimizado, se houver esta etapa) e gera o código alvo (geralmente um conjunto de instruções de um processador específico).
- Seleciona as operações da máquina-alvo para implementar cada operação da representação intermediária (COOPER, 2014).
- Escolhe uma ordem em que as operações serão executadas de modo mais eficiente e

decide quais valores serão armazenados nos registradores e quais serão armazenados na memória, inserindo código para impor essas decisões.

Um dos grandes desafios da etapa de síntese reside na **alocação de registradores**, pois eles são finitos e, tipicamente, um conjunto pequeno de registradores de propósito geral é fornecido pelos processadores. Assim, é fundamental que um compilador, para ser eficiente, faça bom uso desses registradores.

SAIBA MAIS

No Capítulo 1 do livro Construindo Compiladores (COOPER, 2014), páginas 13-15 (25-27 no arquivo PDF) há um exemplo interessante sobre como otimizar as instruções de máquina e a alocação de registradores para atribuir a `a` o resultado da expressão `a × 2 × b × c × d`.

É importante observar que essas otimizações não são possíveis na fase de otimização. Por quê? Porque elas são realizadas com o conjunto de instruções do processador, não em uma linguagem intermediária. A seleção das instruções do processador já faz parte da etapa de síntese.

Estrutura de um Compilador

Com base no que foi exposto, a estrutura de um compilador típico pode ser vista em mais detalhes na Figura 3, extraída de Cooper (2014). Nela a fase de *front-end* contém explicitamente as etapas de análise léxica, sintática e de elaboração. O otimizador pode conter diversas iterações de otimização, o que é ilustrado na Figura 3 por meio de *n* otimizações. Já a síntese contém a seleção de instruções, seguida pelo escalonamento de instruções e conclui suas ações com a alocação de registradores.

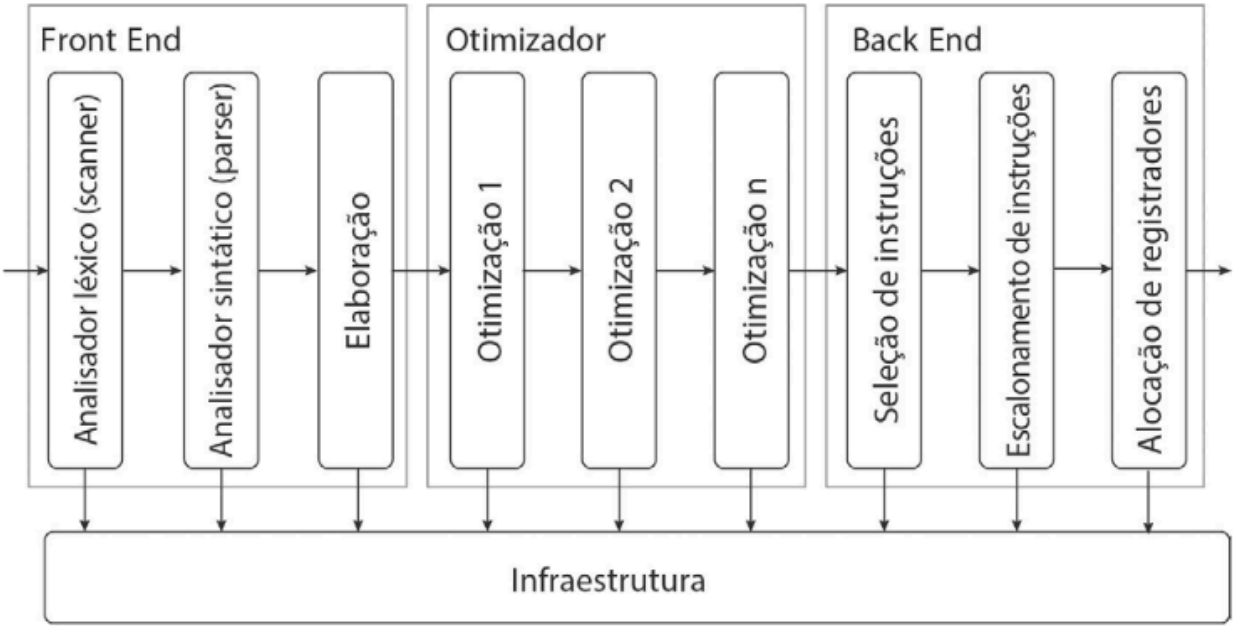

Figura 3. Estrutura de um compilador típico. Fonte: Cooper, 2014.

Há diversos autores que separam as etapas de geração de código intermediário e sua otimização das ações de síntese, como visto na Figura 3. No entanto, outros autores tratam a geração de código intermediário e a sua otimização como parte da síntese também (ver Figura 4), e outros tratam a síntese a partir da otimização (ver Figura 5).

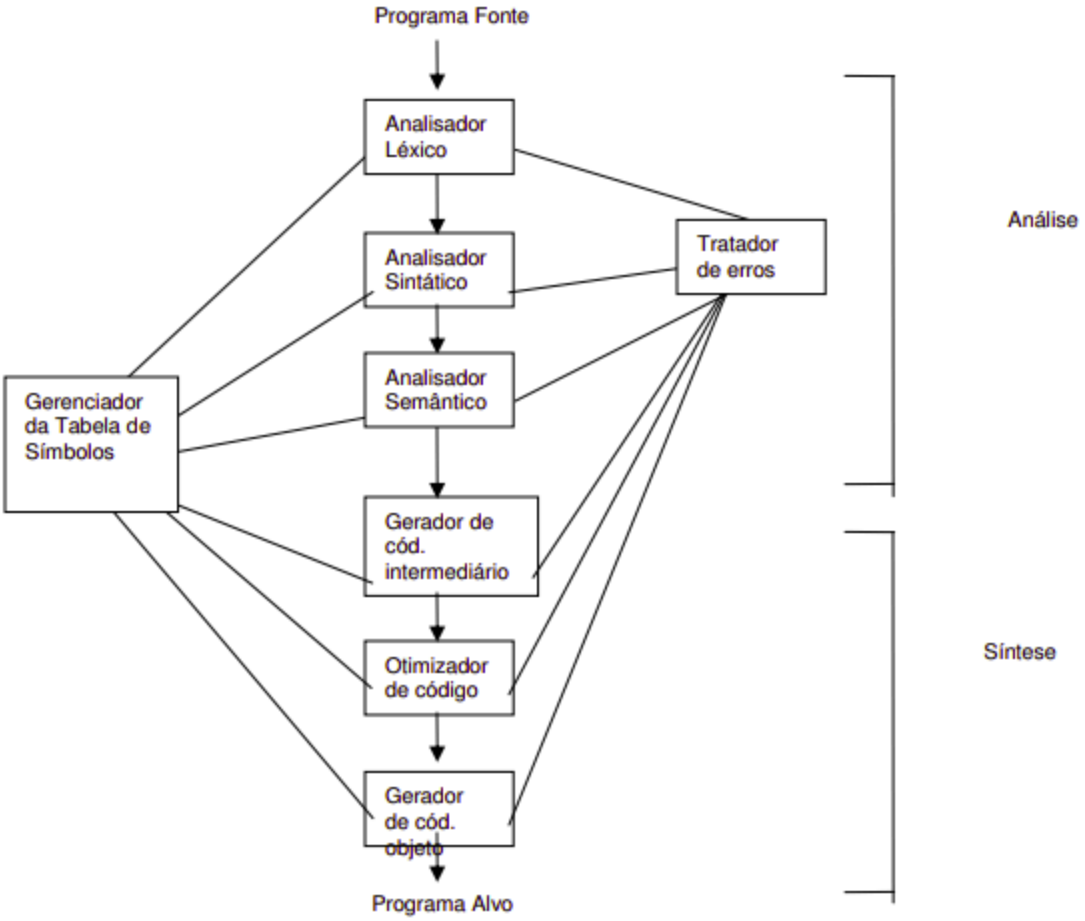

Figura 4. Componentes do compilador. Fonte:

http://professor.pucgoias.edu.br/SiteDocente/admin/arquivosUpload/17389/material/Aulao.pdf.

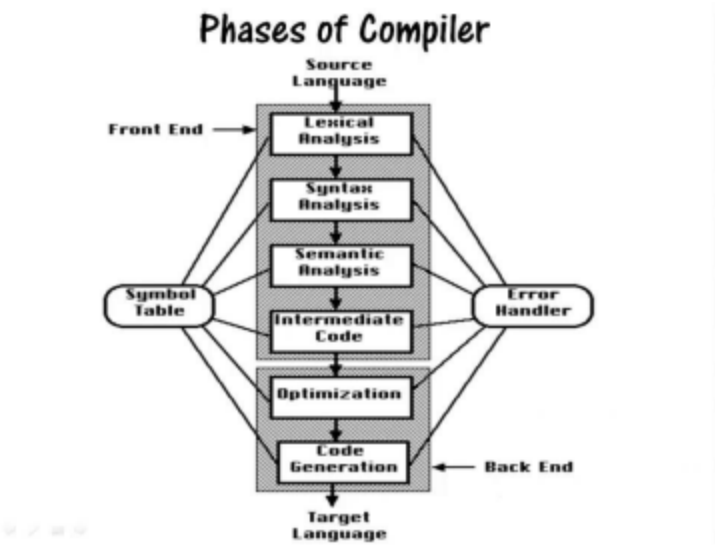

Figura 5. As fases de um compilador. Fonte: ALI, Shaukat, 2020.

Compiladores possuem ainda outros elementos importantes. Entre eles, podem ser destacados a **tabela de símbolos** e o **tratador de erros** (presentes na Figura 4), detalhados em materiais futuros. Estes elementos são utilizados em todas as fases da compilação.

Conclusão

O compilador pode ser dividido em grandes grupos de componentes, que, tipicamente, são dois ou três: análise[, otimização] e síntese. A análise (*front-end*), verifica aspectos léxicos, sintáticos e semânticos do código, gerando uma representação intermediária. A otimização, quando presente, aprimora essa representação para melhorar algum(ns) aspecto(s) do programa. A síntese (*back-end*), por fim, traduz a representação intermediária em código alvo, selecionando as operações adequadas e otimizando o uso de registradores.

Para avançar nos estudos, é essencial compreender as etapas mencionadas, assim é possível partir para o detalhamento dos componentes do compilador, presentes nas Figuras 3-5.

Exercícios

1) Explique como uma otimização bem-sucedida pode impactar um código gerado.

2) Quais são as diferenças entre os modelos de compilação de duas e três fases? Quais são as vantagens e desvantagens de cada abordagem?
3) Assista aos vídeos seguintes vídeos do professor Tiago Fernandes Tavares:
    3.1) Etapas de Compilação
    3.2) Etapas de Compilação no GCC
    3.3) Compilando Programas por Etapas

Referências

AHO, Alfred V. et al. Compilers: principles, techniques, & tools. Pearson Education India, 2007.

DU BOIS, André Rauber. Notas de Aula sobre Compiladores. 09 ago. 2011.

COOPER, Keith; TORCZON, Linda. Construindo Compiladores. Vol. 1. Elsevier Brasil, 2014.