Compiladores

Introdução à disciplina

Introdução

Este material contém as notas de aula iniciais da disciplina *Compiladores*. O objetivo é apresentar a disciplina, resgatar o histórico dos compiladores, os principais conceitos e tipos de compiladores.

Compilador

Compilador é uma ferramenta que transforma o programa escrito em uma linguagem fonte em um código de uma linguagem alvo (Figura 1), semanticamente equivalente.

Figura 1. Esquema resumido de um compilador.

O Compilador atua como um tradutor e normalmente (mas não sempre – ver tradutor fonte a fonte) é utilizado para converter um código escrito em uma linguagem de alto nível para um código de máquina para um processador.

Segundo Cooper (2014), para que a tradução ocorra, o compilador deve entender tanto a forma (ou sintaxe) quanto o conteúdo (ou significado/semântica) da linguagem de entrada, e, ainda,

as regras que controlam a sintaxe e o significado na linguagem de saída.

Um compilador é um sistema de software de grande porte, com muitos componentes e algoritmos internos e interações complexas entre eles. Assim, o estudo da construção de compiladores é uma introdução às técnicas para a tradução e o aperfeiçoamento de programas, além de um exercício prático em engenharia de software (COOPER, 2014).

> "*Um bom compilador contém um microcosmo da ciência da computação. Faz uso prático de algoritmos gulosos (alocação de registradores), técnicas de busca heurística (agendamento de lista), algoritmos de grafos (eliminação de código morto), programação dinâmica (seleção de instruções), autômatos finitos e autômatos de pilha (análises léxica e sintática) e algoritmos de ponto fixo (análise de fluxo de dados). Lida com problemas, como alocação dinâmica, sincronização, nomeação, localidade, gerenciamento da hierarquia de memória e escalonamento de pipeline. Poucos sistemas de software reúnem tantos componentes complexos e diversificados. Trabalhar dentro de um compilador fornece experiência em engenharia de software, difícil de se obter com sistemas menores, menos complicados*" (COOPER, 2014).

Exemplos:
Código C → Compilador C → Linguagem de Máquina
Texto LaTeX → Compilador LaTeX → Postscript[1] ou PDF
Java → Javac → Java Bytecode → JVM → Resultado
Haskell → GHC → C

Em resumo, o que consta nos exemplos acima poderia ser escrito genericamente como:
Fonte → Compilador → Alvo

[1] Um programa de composição tipográfica que produz PostScript toma como entrada uma especificação de como o documento aparece na página impressa e produz como saída um arquivo PostScript. PostScript é uma linguagem para descrever imagens. Como este programa assume uma especificação executável e produz outra também executável, ele é um compilador. O código que transforma PostScript em pixels é normalmente um interpretador, não um compilador (adaptado de COOPER, 2014).

Linguagens fonte típicas podem ser C, C++, C#, Python ou Java. Linguagem alvo geralmente é o conjunto de instruções de algum processador.

Exemplos de compiladores conhecidos: `gcc`, `javac`, `ghc`, `pypy`.

Alguns compiladores para C e outras linguagens:

- https://ideone.com/Hiq1Or (clang)
- https://rextester.com/l/c_online_compiler_visual
- https://www.onlinegdb.com/online_c_compiler (gcc? e turbo c)

Compiladores de uma mesma linguagem geram códigos iguais?

- Criar um código de Hello World simples.
- Compilar o código com o GCC:

  `gcc hello-world.c -o hello-world-gcc.out`
- Linha necessária para criar um *link*, pois `clang-6.0`, `clang-12` ou outro (verifique a versão instalada) era o comando reconhecido:

  `sudo ln -s /usr/bin/clang-6.0 /usr/bin/clang`

  `sudo ln -s /usr/bin/clang-12 /usr/bin/clang`
- Compilar o código com o CLang:

  `clang hello-world.c -o hello-world-clang.out`
- Verificar se os arquivos gerados são diferentes:

  `diff hello-world-gcc.out hello-world-clang.out`
- Verificar alguns caracteres imprimíveis nos executáveis:

  `strings hello-world-gcc.out | tail`

  `strings hello-world-clang.out | tail`

  `strings hello-world-gcc.out | head`

  `strings hello-world-clang.out | head`

  `objdump -D hello-world-clang.out`

  `objdump -D hello-world-gcc.out`

As saídas dos dois comandos `objdump` acima estão disponíveis nos arquivos `objdump-D-gcc.txt` e `objdump-D-clang.txt`.

`objdump` é uma ferramenta de linha de comando utilizada para exibir informações sobre arquivos objeto em sistemas operacionais *Unix-like*. Um arquivo objeto é geralmente gerado durante o processo de compilação e contém código de máquina, dados e informações de cabeçalho associadas ao programa ou à biblioteca. Entre as ações realizadas por esta ferramenta estão a desmontagem de código ou *disassembly* (**`objdump -d arquivo.obj`**), a listagem de seções (parâmetro **`-h`**), a exibição de tabela de símbolos (parâmetro **`-t`**) entre outros.

Breve Histórico

- Os primeiros compiladores surgiram na década de 50.
- Vários compiladores foram construídos nas décadas de 60 e 70. O foco era a otimização do código com base na arquitetura CISC;
- Na década de 80, com o surgimento da arquitetura RISC, surgiram compiladores com forte foco em otimização;
- Na década de 90, surgiram novos desafios para a construção de compiladores, como controle de *pipelines*, latência de memória e geração de código paralelo.

Compilação e Interpretação

Um interpretador toma como entrada uma especificação executável e produz como saída o resultado da execução da especificação:
Fonte → Interpretador → Resultados

Um interpretador possui o mesmo *front-end* que um compilador, a diferença dos dois está no *back-end*. Os conceitos de *front-end* e *back-end* serão vistos ao longo da disciplina e estarão presentes nas notas de aula.

O *back-end* é um grande componente que será detalhado na disciplina.

O *back-end* do compilador recebe uma árvore sintática e transforma esta árvore em código alvo.

O *back-end* do interpretador não gera código, mas executa as instruções presentes em um código intermediário. No interpretador, muitas vezes esse código intermediário é a própria árvore sintática.

O processo de tradução é mais rápido no interpretador, pois ele tem menos fases que o compilador. No entanto, o código gerado pelo compilador é mais rápido do que a execução de um interpretador. Outro aspecto a ser destacado é a vantagem do interpretador geralmente fornecer mensagens de erro melhores em tempo de execução, já que ele possui acesso ao código-fonte.

Alguns interpretadores analisam e executam os programas linha a linha. Um exemplo é o interpretador da linguagem BASIC.

Há diversas linguagens com implementações interpretadas. Entre elas, pode-se citar:

- JavaScript
- Perl
- Python
- R
- Ruby

Semelhanças entre Compiladores e Interpretadores (Cooper, 2014):

- Analisam o programa de entrada e determinam se é ou não um programa válido;
- Constroem um modelo interno da estrutura e significado do programa;
- Determinam onde armazenar valores durante a execução.

Existem ainda linguagens cujas implementações adotam esquemas de tradução que incluem tanto compilação quanto interpretação, como ocorre com Java.

A linguagem Java é compilada do código-fonte para um formato denominado *bytecode*, uma representação intermediária compacta, que visa diminuir tempos de *download* para aplicativos Máquina Virtual Java. Aplicativos Java são executados usando o *bytecode* na Máquina Virtual Java (*Java Virtual Machine* – JVM), um interpretador para *bytecode*. Para complicar ainda mais

as coisas, algumas implementações da JVM incluem um compilador, às vezes chamado de compilador *just-in-time* – JIT, que, em tempo de execução, traduz sequências de *bytecode* muito usadas em código nativo para o computador subjacente.

Além de Java, também existem os compiladores JIT Python (PyPy), o .NET da Microsoft, entre outros.

Em geral, diz-se que compiladores JIT trazem benefícios de desempenho e otimizam o código. Por outro lado, podem trazer problemas de segurança relativos ao acesso aos dados e também um custo inicial elevado, especialmente sentido em aplicações pequenas.

Tradutores fonte a fonte

> "*Alguns compiladores produzem um programa-alvo escrito em uma linguagem de programação orientada a humanos, em vez da assembly de algum computador. Os programas que esses compiladores produzem requerem ainda tradução antes que possam ser executados diretamente em um computador. Muitos compiladores de pesquisa produzem programas C como saída. Como existem compiladores para C na maioria dos computadores, isto torna o programa-alvo executável em todos estes sistemas ao custo de uma compilação extra para o alvo final. Compiladores que visam linguagens de programação, em vez de conjunto de instruções de um computador, frequentemente são chamados tradutores fonte a fonte*" (COOPER, 2014).

Tradutores fonte a fonte também são ditos compiladores fonte a fonte, transpiladores ou transcompiladores.

Tradutores fonte a fonte podem ser utilizados para tradução de paralelismo automático (por *automatic parallelization compiler*) e de código legado, por exemplo, quando é necessário substituir instruções antigas, obsoletas e não aceitas em uma *Application Programming Interface* – API ou em novas versões da linguagem por outras instruções atualizadas. Outra possibilidade é a inversa: escrevemos um código considerando instruções atualizadas, mas o código pode ser executado por quem usa ferramentas desatualizadas. Nesse caso, deseja-se

garantir a retrocompatibilidade.

Um exemplo é o uso de pré-processadores de CSS, por exemplo, o SASS. O SASS dispõe de diversos recursos que facilitam a escrita das regras de estilização. Ao compilar o código, gera-se um código CSS equivalente e interpretável por navegadores *web*. A ferramenta de linha de comando sass também pode ser considerada um transpilador.

Ofuscadores também são considerados tradutores fonte a fonte. Conforme Goldwasser & Rothblum (2007), um ofuscador é um compilador que transforma qualquer programa em um programa ofuscado que possui a mesma funcionalidade de entrada-saída do programa original, mas é "ininteligível". Ofuscações têm aplicações para criptografia e proteção de software.

Um exemplo de código de entrada está disponível no quadro abaixo:

```
function NewObject(prefix) {
      var count=0;
      this.SayHello=function(msg) {
                  count++;
                  alert(prefix+msg);
      }
      this.GetCount=function() {
                  return count;
      }
}
var obj=new NewObject("Message : ");
obj.SayHello("You are welcome.");
```

Um exemplo de código de saída após a ação de um ofuscador (JavaScript Obfuscator[2]) está disponível no quadro a seguir:

```
var
_0xa034=["\x53\x61\x79\x48\x65\x6C\x6C\x6F","\x47\x65\x74\x43\x6F\x75
```

[2] Disponível em https://javascriptobfuscator.com/Javascript-Obfuscator.aspx.

```
\x6E\x74","\x4D\x65\x73\x73\x61\x67\x65\x20\x3A\x20","\x59\x6F\x75\x2
0\x61\x72\x65\x20\x77\x65\x6C\x63\x6F\x6D\x65\x2E"];
function NewObject(_0x296bx2){
var _0x296bx3=0;
this[_0xa034[0]]=function(_0x296bx4){
_0x296bx3++;
alert(_0x296bx2+ _0x296bx4)};
this[_0xa034[1]]=function(){return _0x296bx3}}
var obj=new NewObject(_0xa034[2]);
obj.SayHello(_0xa034[3]);
```

Outro exemplo conhecido, também geralmente utilizado no universo do *front-end* para *web*, é o Babel, um *transpiler* para JavaScript.

Os passos a seguir reproduzem um exemplo:

1. Abrir o console do navegador, geralmente pressionando a tecla F12, e digitar:

```
const aula = disciplina => `Neste momento nós estamos na aula de
${disciplina.toUpperCase()}.`
```

O comando acima criou uma *arrow function* com identificador `aula`, que recebe o parâmetro disciplina e exibe uma mensagem.

2. Ainda no console, pode-se usar a função diretamente como parâmetro para a instrução `console.log`, da seguinte forma:

```
console.log(aula('Compiladores'))
```

A função funcionou bem e havia outras formas de criá-la. Uma vez que ela foi criada desta forma, será que navegadores antigos são capazes de entender esta instrução?

A resposta pode ser obtida entrando em https://jstool.gitlab.io/babel-es6-to-es5/.

No campo da esquerda, basta digitar o código:

```
const aula = disciplina => `Neste momento nós estamos na aula de
${disciplina.toUpperCase()}.`
```

```
console.log(aula('Compiladores'))
```

No campo da direita aparecerá o código correspondente retrocompatível:

```
"use strict";

var aula = function aula(disciplina) {
	return "Neste momento n\xF3s estamos na aula de ".concat(disciplina.toUpperCase(), ".");
};

console.log(aula('Compiladores'));
```

Princípios da Compilação (Cooper, 2014)

1. O compilador deve preservar o significado do programa a ser compilado. Ou seja, a tradução não pode alterar o que o programa faz. É necessário traduzir com exatidão.
2. O compilador deve melhorar o programa de entrada de alguma forma perceptível. Como? Tornando o programa executável na máquina alvo.

O babel (*transpiler*), por exemplo, cumpre esses princípios?

Fases da Compilação

Inicialmente o compilador foi resumido da seguinte maneira:
Fonte → Compilador → Alvo

Observando a compilação gradativamente em mais detalhes, pode-se dividi-la em duas (ou três) fases. A divisão da compilação em duas fases envolve ***análise*** (***front-end***) e **síntese** (***back-end***), como mostra a Figura 2. A fase de análise é a primeira, seguida pela fase de síntese. A fase de análise possui como saída o código da linguagem fonte convertido para uma representação intermediária (RI na Figura 2).

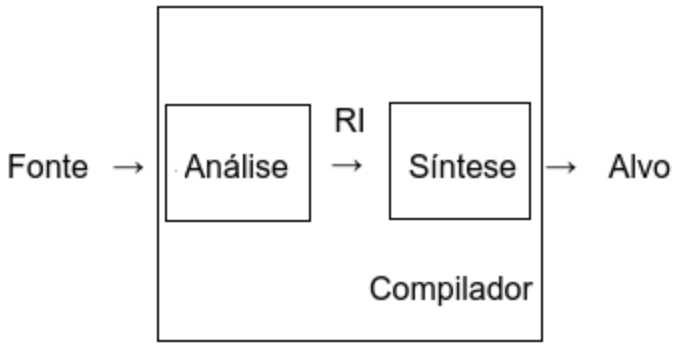

Figura 2. Compilador de duas fases.

A compilação em três fases envolve análise (*front-end*), **otimização** e síntese (*back-end*), como mostra a Figura 3.

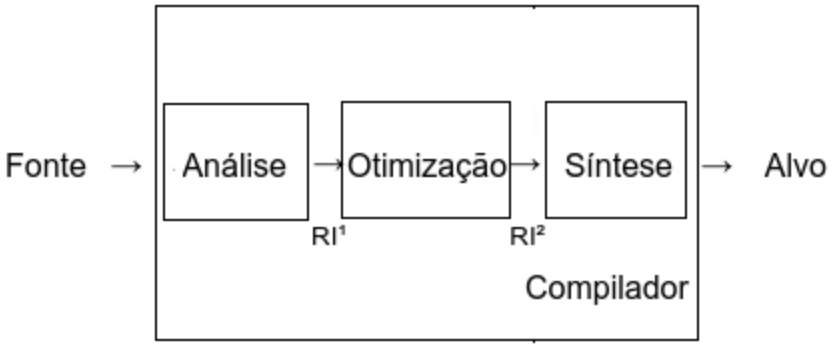

Figura 3. Compilador de três fases.

Conclusão

Na disciplina de Compiladores, explora-se os componentes deste software que converte código escrito em uma linguagem fonte para um código de uma linguagem alvo, preservando sua semântica.

Exercícios

1) O que é um compilador?
2) Por que o compilador deve preservar o significado do programa a ser compilado?

Referências

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

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

GOLDWASSER, Shafi; ROTHBLUM, Guy N. "On best-possible obfuscation." TCC. Vol. 4392. 2007.