Chapter 1

Tópicos Avançados: Compilação para WebAssembly

1.1 Objetivos

⋆ title=Objetivos da Aula

Ao final deste capítulo, o estudante deverá ser capaz de:

- Compreender as motivações históricas e arquiteturais que levaram à criação do WebAssembly.
- Descrever o funcionamento conceitual de uma Máquina de Pilha (*Stack Machine*) na execução do Wasm.
- Entender o fluxo do *Pipeline* de compilação utilizando as infraestruturas modernas (como o LLVM).
- Explicar os modelos de isolamento, em especial a **Memória Linear** (*Linear Memory*), garantindo a segurança de execução.
- Avaliar a expansão do Wasm para além dos navegadores (Cloud/Serverless) através do padrão **WASI**.

1.2 Introdução: O Desafio do Desempenho na Web

Por mais de vinte anos, o ecossistema da *World Wide Web* repousou inteiramente sobre uma única linguagem de execução do lado do cliente: o **JavaScript**. Criado originalmente em 1995 para adicionar validações simples a formulários web, a linguagem precisou evoluir para suportar aplicações imensas.

Com a criação dos motores de compilação dinâmicos em níveis (como os JITs V8 do Chrome e SpiderMonkey do Firefox), o JavaScript tornou-se excepcionalmente rápido para a grande maioria das tarefas tradicionais (interface de usuário, requisições de rede). No entanto, à medida que empresas procuravam portar motores complexos de jogos 3D (como a Unity e a Unreal Engine), sistemas de compressão de áudio ou bibliotecas nativas de Visão Computacional massivas escritas em C++ para a Web, gargalos arquiteturais severos surgiram:

- **Custo de Parsing:** O tempo necessário para baixar um código-fonte em formato texto (às vezes 10 MB de JavaScript minimizado), quebrar os *tokens*, montar a Árvore de Sintaxe Abstrata (AST) e passar pela compilação *Just-In-Time* (JIT) era inaceitavelmente alto para inicializar um jogo na web.
- **Coleta de Lixo (Garbage Collection):** O gerenciamento automático de memória em JavaScript resulta em pausas aleatórias do processo inteiro para limpar objetos mortos na RAM. Em jogos a 60 Quadros por Segundo (FPS), essa pausa gera pequenos "engasgos" visuais (*stuttering*).

1.3 A Chegada do WebAssembly (Wasm)

Em resposta a esses limites, em 2015, o consórcio Web (*W3C*) e engenheiros dos principais motores de navegadores criaram o **WebAssembly (Wasm)**. Ao contrário do que o nome pode sugerir, o WebAssembly **não** é uma linguagem que programadores escrevem manualmente. Ele é definido estritamente como um **alvo de compilação portátil e binário**.

Um programa Wasm é um arquivo executável, similar conceitualmente a um `.exe` do Windows ou a um `.jar` do Java, porém projetado com um objetivo central: **segurança draconiana**.

1.3.1 A Máquina de Pilha (Stack Machine)

Enquanto arquiteturas de hardware reais (como a familia x86_64 ou o ARM) são máquinas baseadas em **Registradores** espaciais fixos (como `eax` ou `rbx`), o WebAssembly é especificado como uma **Máquina de Pilha Virtual**.

Nesse modelo, não existem registradores no formato do arquivo. Toda operação ocorre implicitamente empilhando ou desempilhando elementos na Pilha de Controle atual.

```
1 i32.const 5       ;; Faz push do inteiro 5 no topo da pilha
2 i32.const 10      ;; Faz push do inteiro 10 no topo da pilha
3 i32.add           ;; Consome (pop) os 2 valores, soma, e faz push
     do 15
4 local.set 0       ;; Consome (pop) o topo e atribui aa variavel
     local 0
```

Listing 1.1: Exemplo de código no formato de texto (WAT) de uma função de soma.

A escolha por uma máquina de pilha não visa o hardware final, mas sim o **tamanho do código**. Por ser implícita (não sendo necessário codificar na instrução que a adição

pegará o ‘R1‘ e ‘R2‘ e colocará no ‘R3‘), as instruções Wasm são incrivelmente compactas, reduzindo o tempo de *download* ao mínimo necessário. Ademais, validar uma máquina de pilha para impedir execuções maliciosas é algoritmicamente muito mais ágil do que tentar prever os efeitos de dezenas de registradores descontrolados.

1.4 O Pipeline de Compilação com LLVM

Um dos maiores triunfos do Wasm foi a compatibilidade instantânea gerada pela comunidade acadêmica e de desenvolvimento. A tecnologia principal que viabilizou o Wasm foi o compilador infraestrutural **LLVM** que estudamos anteriormente no curso.

Como o LLVM já traduzia a linguagem `C/C++` por meio do Clang para sua *Representação Intermediária (LLVM IR)*, a equipe do WebAssembly apenas precisou escrever um novo Módulo *Back-end* para o LLVM. Isso abriu instantaneamente as portas da Web para montanhas de códigos nativos previamente inacessíveis ou presos ao sistema operacional original.

Hoje, além do C e C++, os desenvolvedores rotineiramente usam o LLVM para compilar **Rust** diretamente para a Web, alcançando um ambiente isento das falhas típicas de vazamento de memória associadas aos ponteiros de C.

1.5 Isolamento de Memória (Linear Memory)

A maior dúvida arquitetural inicial era: *"Se permitirmos que um programa C++ compilado rode dentro do Chrome, e esse programa contém manipulação de ponteiros nulos ou invasivos (como o temido ponteiro solto de C++), o hacker não poderá corromper a memória do próprio navegador para roubar senhas de outras abas?"*

A genialidade por trás da segurança total do Wasm é seu modelo de memória isolada chamado **Memória Linear** (*Linear Memory*).

Quando uma instância de WebAssembly é iniciada pelo navegador, ela não obtém acesso à RAM do sistema operacional nem à RAM inteira do navegador. Em vez disso, o navegador aloca silenciosamente um simples grande vetor contíguo (como um *Array* estático de JavaScript).

Para o módulo Wasm, os endereços começam estritamente no índice 0 até a capacidade máxima desse vetor. Quando o código C++ tenta executar `*(char*)1024 = 'A'`, ele não está gravando na memória virtual do Windows, mas sim escrevendo estritamente no índice `1024` do Vetor seguro e fechado isolado pelo interpretador em formato *Sandbox*. Se o binário tentar ler além do limite final do Array, uma infração de fronteira é facilmente interceptada pela CPU real e o módulo simplesmente é ejetado (morre em um *Trap*), sem afetar nem mesmo a página web que o executa, garantindo 100% de segurança.

1.6 O Futuro: Extrapolando o Navegador (WASI)

Uma vez criado um ambiente virtual hiper-otimizado, extremamente seguro e universal, a indústria percebeu que o Wasm serviria perfeitamente fora da web.

Foi estabelecida a **WebAssembly System Interface (WASI)**. Ela define as permissões que um sistema operacional fornece aos programas isolados em Wasm para ler arquivos do disco ou enviar pacotes de rede. Hoje, módulos em WebAssembly são utilizados comercialmente para competir e até substituir contêineres Docker.

Enquanto um container Docker exige trazer toda uma distribuição Linux na imagem (gerando arquivos imensos de 200 MB), uma instância de WebAssembly pode pesar apenas 1 a 2 MegaBytes, *"bootando"* (iniciando a execução) em microssegundos em qualquer servidor de Computação em Nuvem ou arquiteturas *Edge* (*Serverless*), marcando uma revolução definitiva das técnicas clássicas de compiladores na engenharia moderna.