

Compiladores (CC3001) — Aula 1: Apresentação

Mário Florido

DCC/FCUP

2024



FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Objetivos e funcionamento

Compiladores e interpretadores

Extras

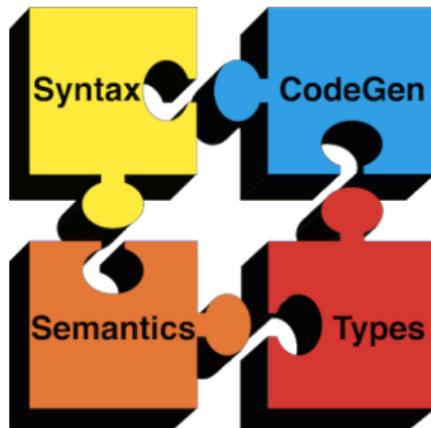
Objetivos e funcionamento

Compiladores e interpretadores

Extras

1. Princípios de conceção e construção de compiladores

- ▶ análise léxical
- ▶ análise sintática
- ▶ árvore sintática abstrata
- ▶ geradores de analisadores lexicais e sintáticos
- ▶ tabelas de símbolos
- ▶ invocação de funções e registos de ativação
- ▶ geração de código intermédio
- ▶ alocação de registos e geração de código máquina



2. Trabalho laboratorial: implementação de um compilador para uma linguagem imperativa simples

Aulas teóricas

- ▶ Presenciais no anfiteatro
- ▶ Exposição de conceitos e exemplos

Aulas laboratoriais

- ▶ Presenciais nos laboratórios
- ▶ Resolução de exercícios
- ▶ Acompanhamento do trabalho
- ▶ *Moodle* e *GitHub Classroom* para exercícios e trabalho

- ▶ Recomendado sistema operativo GNU/Linux
- ▶ Windows: use o WSL (*Windows Subsystem Linux*)
- ▶ *haskell-platform*: GHC, alex, happy, bibliotecas padrão
- ▶ Pode também usar Java ou C
- ▶ O seu editor de texto/IDE predileto
- ▶ *Git* (linha de comando)

Trabalho prático: 15% + 15% (2 fases)

Exame final: 70%

- ▶ Os exercícios de aulas são individuais
- ▶ Trabalho prático:
 - ▶ **componente obrigatória**
 - ▶ realizado em grupo de 2 estudantes
 - ▶ entregas e apresentações: 1^a fase em novembro, 2^a fase em dezembro
- ▶ Classificação mínima no exame final: 40% (8 valores em 20)

1. *Basics of Compiler Design*, Torben Aegidus Mogensen:
http://hjemmesider.diku.dk/~torbenm/Basics/basics_lulu2.pdf
2. *Modern Compiler construction in ML*, Andrew W. Appel, Cambridge University Press (existem também versões em C e Java.)
3. *Compilers: Principles, Techniques, and Tools*, Alfred Aho, Jeffrey Ullman, Ravi Sethi, Monica Lam

Objetivos e funcionamento

Compiladores e interpretadores

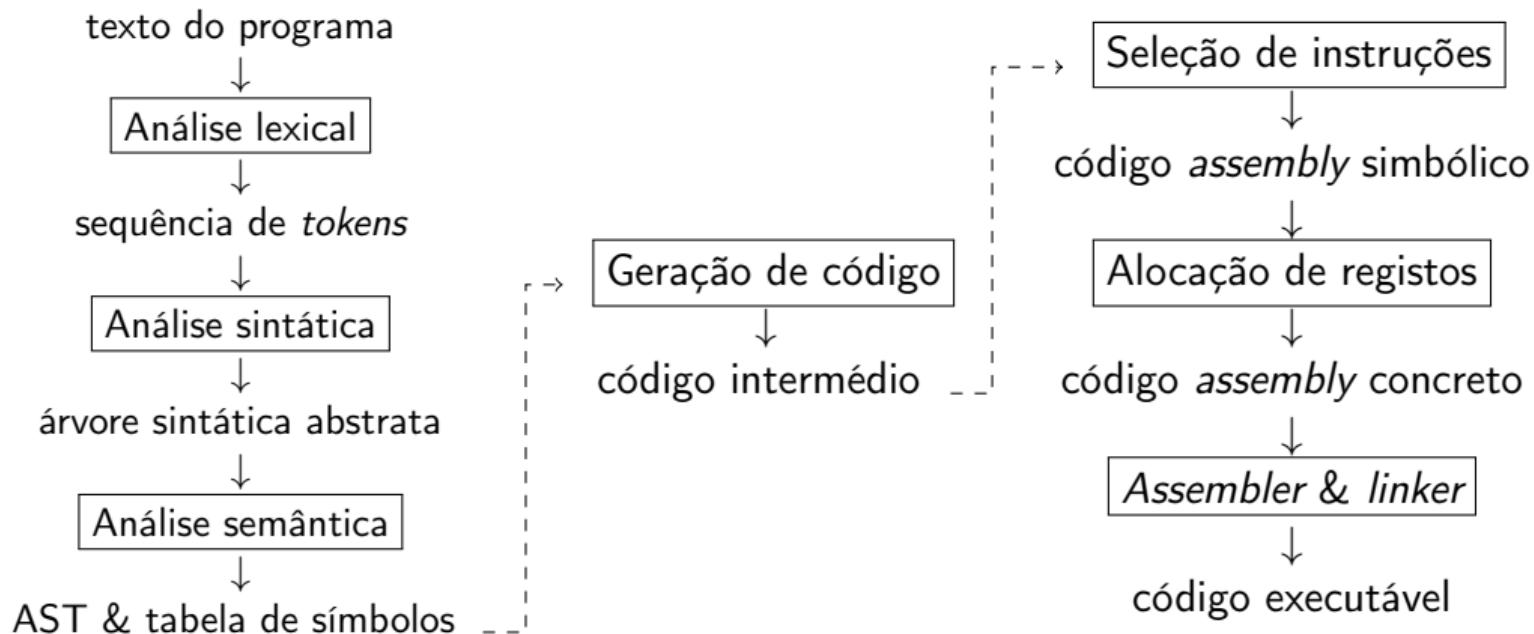
Extras

- ▶ Tradutor de programas numa linguagem de programação para outra
- ▶ Usualmente: traduz uma linguagem de *alto nível* numa de
- ▶ Principal técnica para implementação de linguagens

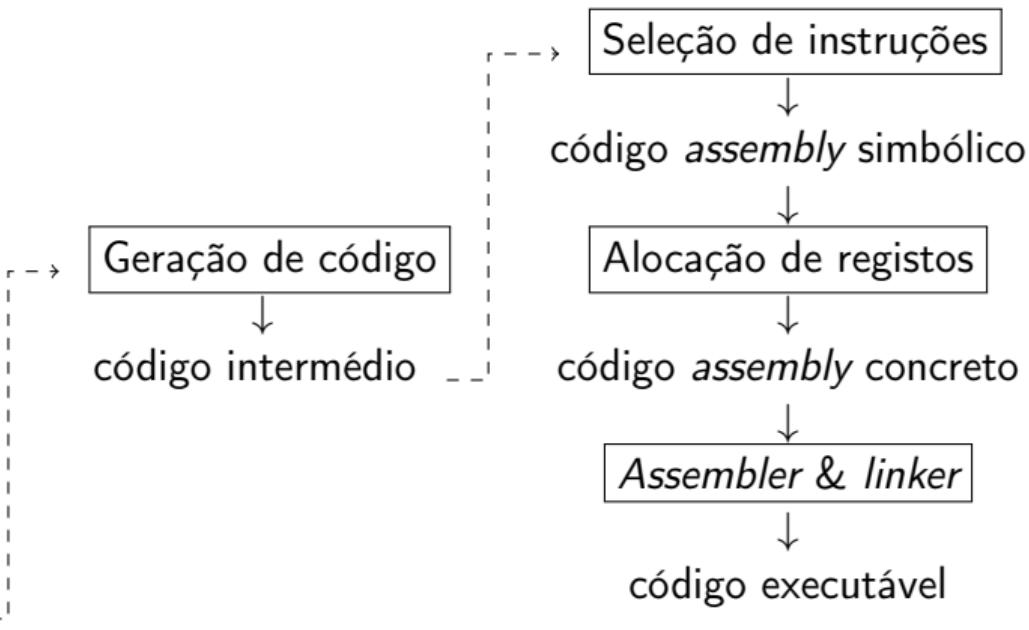
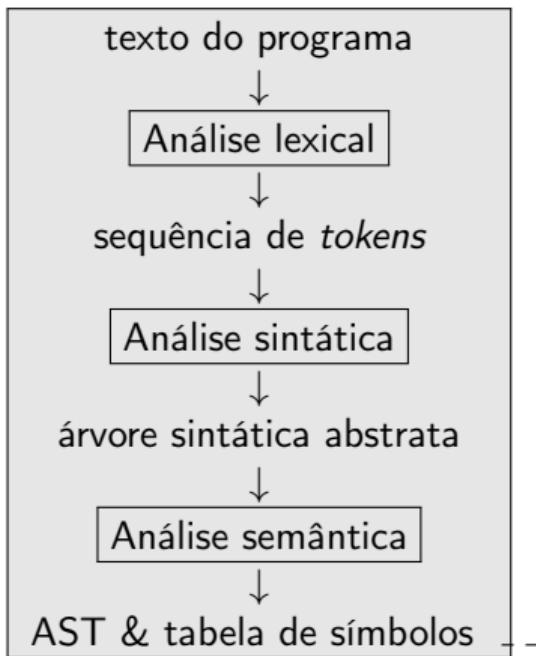
	linguagem fonte	linguagem destino
GCC	C/C++/...	código máquina ¹
Clang	C/C++/...	código máquina
Fpc	Pascal	código máquina
GHC	Haskell	código máquina
Rustc	Rust	código máquina
Javac	Java	código JVM ²
Scalac	Scala	código JVM
elm	Elm	JavaScript

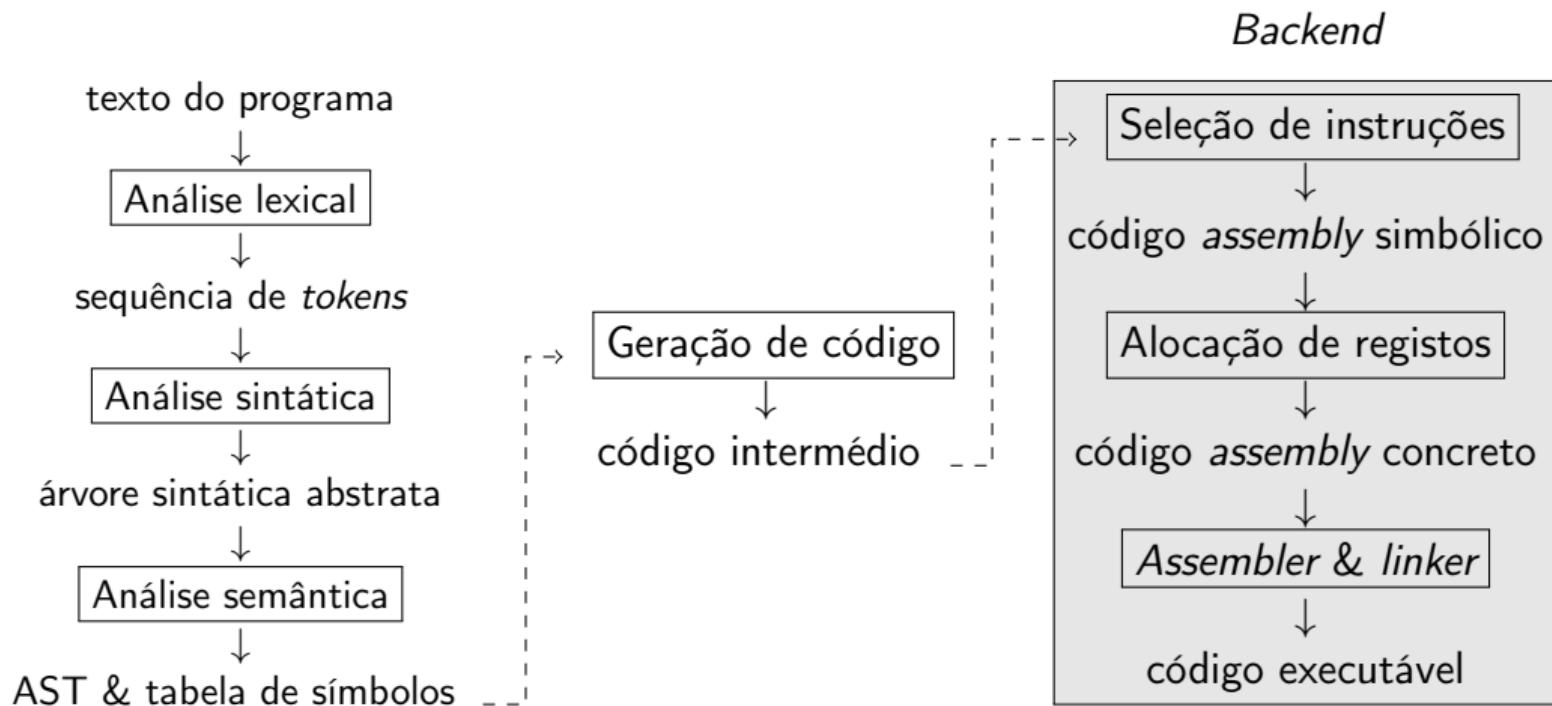
¹E.g. X86, ARM, ...

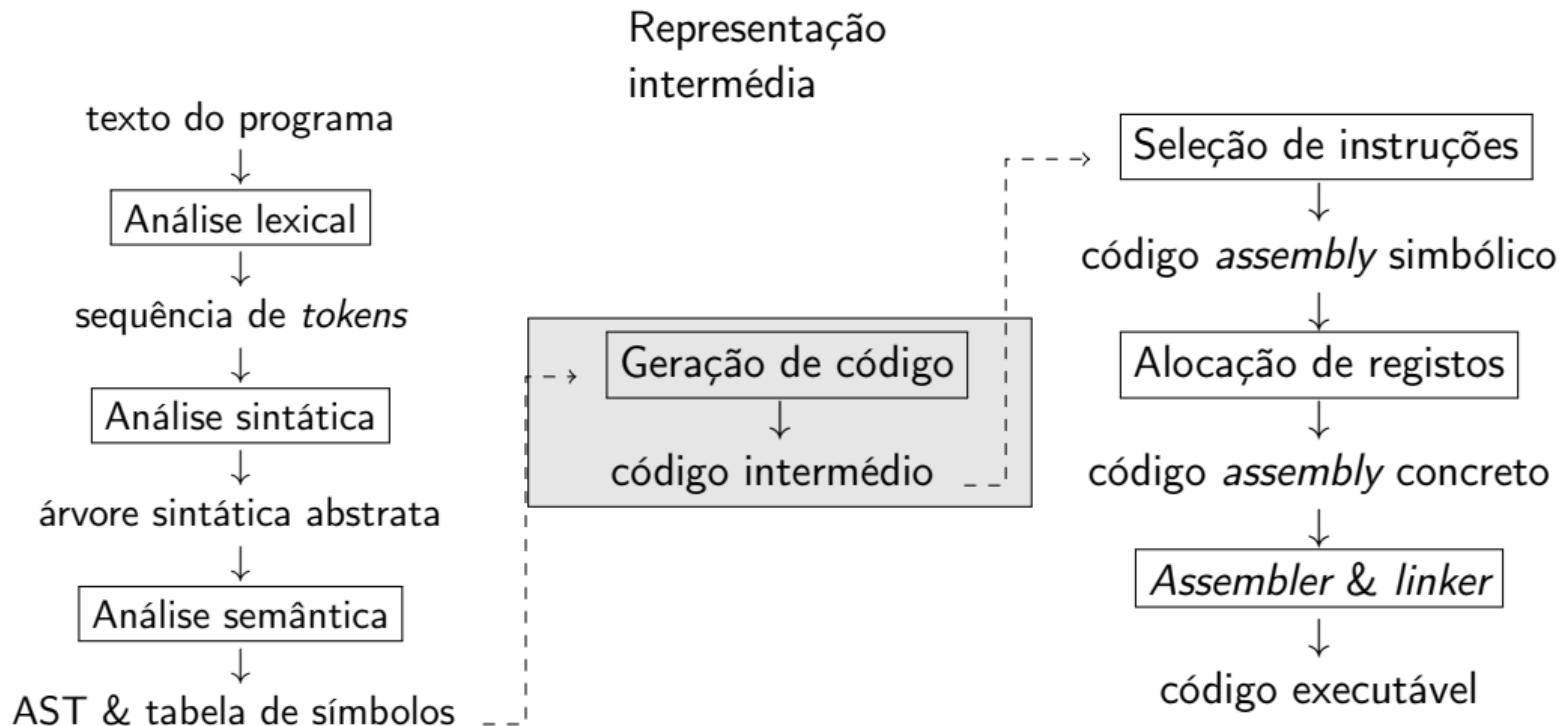
²Java Virtual Machine



Frontend







- ▶ As fases de *frontend* lidam com linguagem fonte
- ▶ As fases de *backend* lidam com a geração de código máquina
- ▶ A fase intermédia é independente da linguagem fonte e do código máquina
- ▶ Esta decomposição torna o compilador mais simples
- ▶ Permite re-utilização de componentes
 - ▶ o *backend* do GCC é usado para compilar C, C++ ou Objective-C
 - ▶ o *backend* LLVM é usado pelo Clang, Swift, Rustc e (opcionalmente) GHC
- ▶ Extras: optimizações em código intermédio e/ou no *backend*

Em vez de um compilador, podemos implementar um *interpretador*:

- ▶ efetua a análise lexical e sintática da linguagem fonte (tal como um compilador);
- ▶ executa diretamente o programa usando a árvore sintática
- ▶ em alternativa: pode gerar e executar um código intermédio

Vantagens da interpretação:

- ▶ Mais simples do que a compilação
- ▶ Mais fácil suportar diferentes arquiteturas de computadores
- ▶ Suporta desenvolvimento interativo (*read-eval-print-loop*)

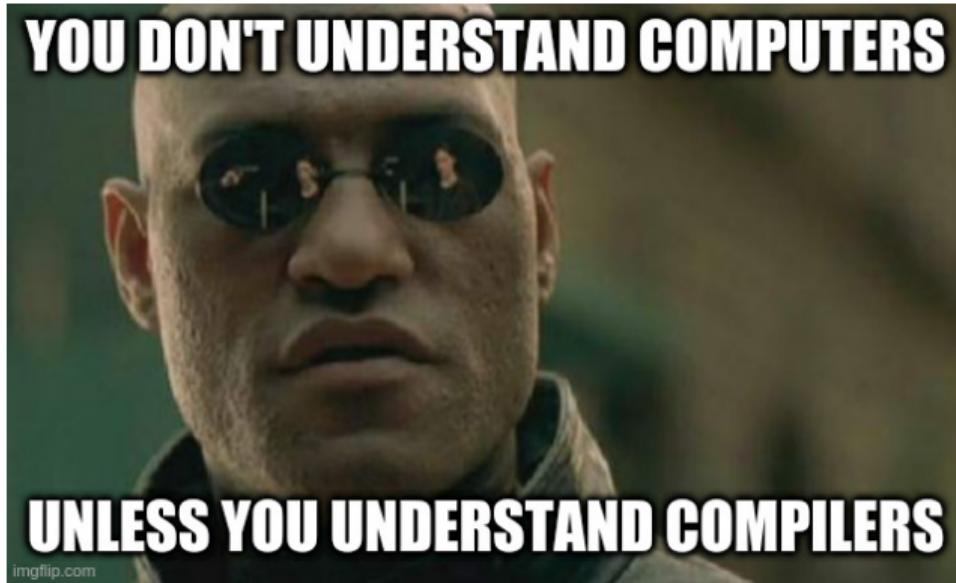
Vantagens da compilação:

- ▶ Código compilado pode ser bastante mais eficiente
- ▶ Executáveis podem ser distribuídos separadamente do compilador

Algumas implementações combinam compiladores e interpretadores (e.g. GHC, OCaml).

A maioria dos programadores não vai escrever o seu próprio compilador.

Porquê então estudar compiladores?



- ▶ Compreender a ligação entre linguagens de alto e baixo nível
- ▶ Técnicas de *parsing* e interpretação são úteis em muitas aplicações
- ▶ É possível que tenha de implementar um compilador ou interpretador para uma *linguagem específica de domínio*
- ▶ Ficará mais capaz de escrever código correto
- ▶ Ficará mais capaz de escrever código eficiente

- ▶ São programas complexos, logo é preferível usar uma linguagem de alto-nível
- ▶ Efetuam transformações entre formatos
(código fonte → AST → código intermédio → código máquina)
- ▶ Linguagens funcionais fortemente tipadas são boas escolhas

A screenshot of a Google search results page. The search query "functional programming compiler" is entered in the search bar. Below the search bar, there are filters for "All", "Images", "Videos", "News", "Maps", and "More". There are also "Settings" and "Tools" buttons. The search results indicate "About 9,900,000 results (0.77 seconds)". The first result is a snippet from stackoverflow.com, which discusses how compilers often work with trees and how functional languages like Haskell have features that make it easier to work with them. The snippet ends with the date "May 25, 2010". Below the snippet, the URL "stackoverflow.com › questions › why-is-writing-a-compil..." and the title "Why is writing a compiler in a functional language easier" are shown. At the bottom of the snippet, there is an ellipsis (...). The page includes standard navigation icons for back, forward, and search.

functional programming compiler

All Images Videos News Maps More Settings Tools

About 9,900,000 results (0.77 seconds)

Often times a **compiler** works a lot with trees. The source code is parsed into a syntax tree. ... **Functional** language have features like pattern-matching and good support for efficient recursion, which make it easy to work with trees, so that's why they're generally considered good languages for writing **compilers**. May 25, 2010

stackoverflow.com › questions › why-is-writing-a-compil...

Why is writing a compiler in a functional language easier

...

- ▶ No entanto: compiladores de linguagens de propósito genérico são tipicamente escritos na própria linguagem (“self-hosting”)
- ▶ Exemplos:

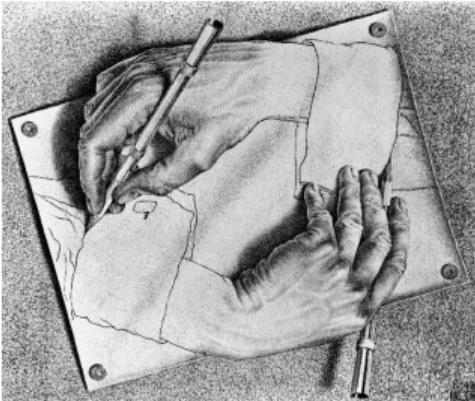
[GCC](#) é escrito em C

[Javac](#) é escrito em Java

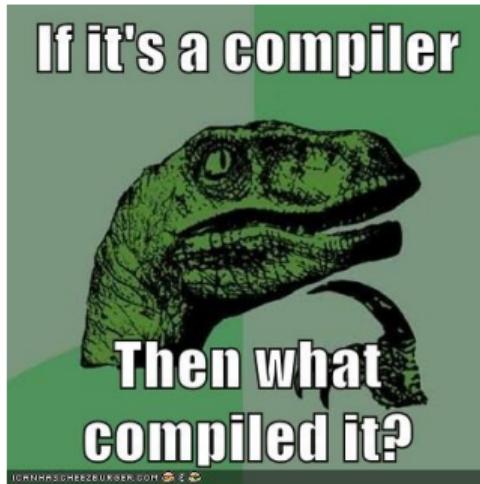
[GHC](#) é escrito em Haskell

[Ocamlc](#) é escrito em OCaml

[Rustc](#) é escrito em Rust



- ▶ Prova da maturidade da linguagem e ferramentas (“eat your own dog food”)
- ▶ Facilita comunicação entre *developers* (uma só linguagem)



Problema: necessitamos de um compilador para compilar o compilador...



Exemplo histórico: a linguagem C.

- ▶ O primeiro compilador de C (1973) foi baseado no compilador de B (uma linguagem anterior)
- ▶ O primeiro compilador de B foi implementado numa outra linguagem de alto nível (TMG) do sistema *Multics*
- ▶ O compilador foi re-escrito em B por estágios até ser capaz de compilar a si próprio...

Fonte: *The Development of the C language*, Dennis Ritchie;
<https://www.bell-labs.com/usr/dmr/www/chist.html>.

Segundo exemplo: o compilador de Rust.

2006-2010: primeiros compilador escrito em OCaml

2010: inicio de re-escrita do compilador em Rust (ainda compilado pelo compilador em OCaml)

Desde 2011: o compilador é desenvolvido em Rust e *self-hosted*

Fonte: *Wikipedia*

[https://en.wikipedia.org/wiki/Rust_\(programming_language\)](https://en.wikipedia.org/wiki/Rust_(programming_language)).

Objetivos e funcionamento

Compiladores e interpretadores

Extras

O que faz um compilador real?

The image shows two side-by-side windows from the Compiler Explorer tool. The left window is titled 'C++ source #1' and contains the following C++ code:

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
```

The right window is titled 'MIPS64 gcc 5.4 (el) (Editor #1, Compiler #1) C++' and displays the generated assembly code:

```
1 square(int):
2         daddiu $sp,$sp,-32
3         sd      $fp,24($sp)
4         move    $fp,$sp
5         move    $2,$4
6         sll     $2,$2,0
7         sw      $2,0($fp)
8         lw      $3,0($fp)
9         lw      $2,0($fp)
```

Both windows have standard interface elements like tabs, dropdown menus, and status bars at the bottom.

- ▶ Compiler explorer: <https://godbolt.org/>
- ▶ Várias linguagens, compiladores e arquiteturas
- ▶ Permite também comparar efeitos de otimizações
- ▶ Vídeo tutorial: https://youtu.be/4_HL3PH4wDg