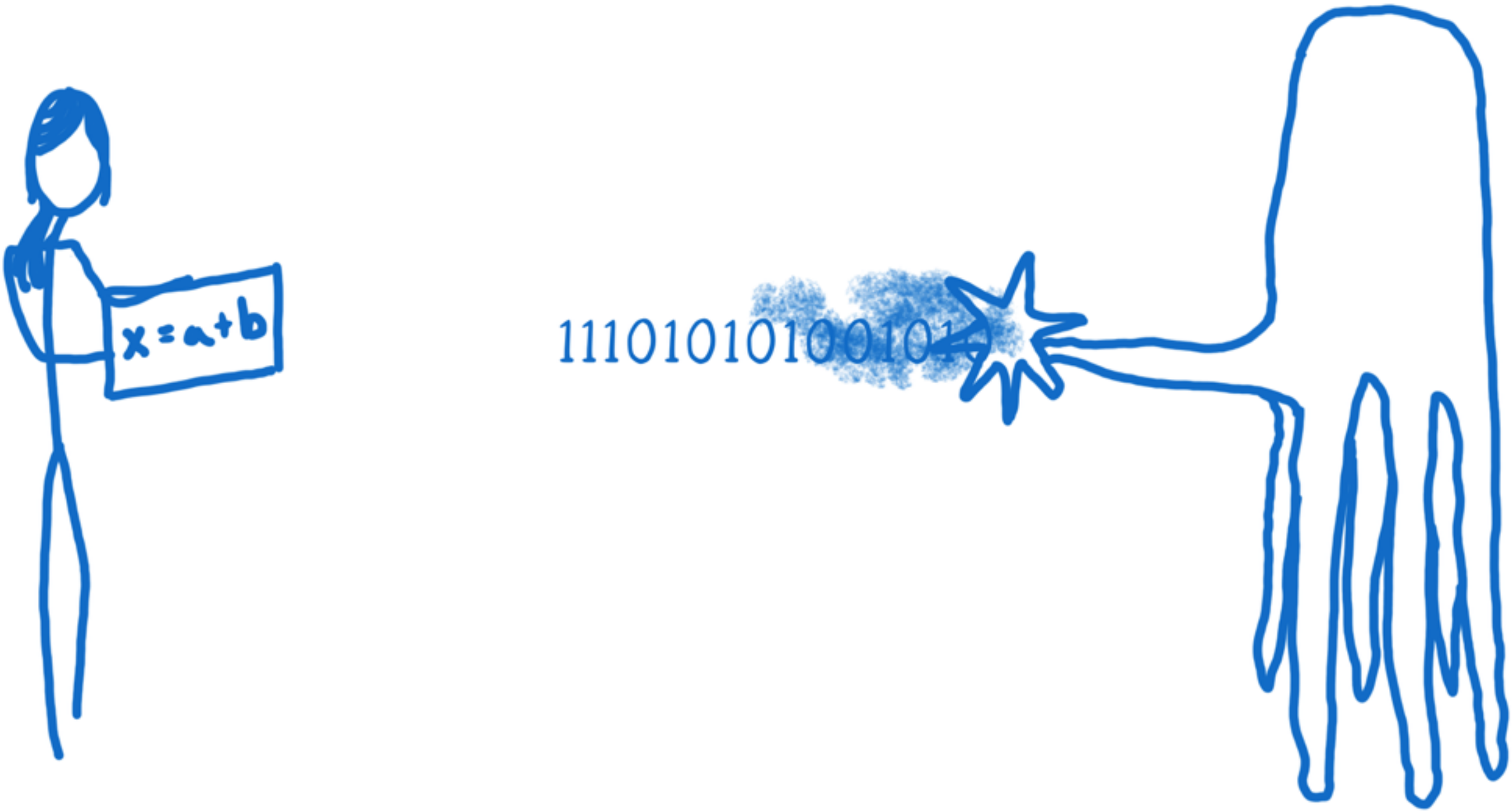
Teoria e Implementação de Linguagens Computacionais

aka Compiladores (IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

A definição de uma linguagem é essencial para a comunicação



https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

Como expressar uma linguagem que o computador entenda?

Linguagens de Programação

No entanto, antes de rodar um programa…

...precisamos traduzi-lo em algo que um computador possa executar

COMPILADORES

traduzir de uma linguagem fonte (mais abstrata) para uma linguagem alvo (mais concreta)

linguagem de programação

vs.

linguagem natural

níveis de abstração

Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman.
Compilers: Principles, Techniques, and Tools (*2nd Edition*)

código-fonte

otimizado para humanos...

código assembly

otimizado para hardware...

quando surgiu a primeira linguagem de programação?

Plankalkül (1943)

apenas projetada…

Short Code (1949)

John Mauchly

Mais História…

- No início dos anos 50, Grace Hopper desenvolveu o sistema A-0 para UNIVAC I
- 1957: Compilador de FORTRAN feito pela IBM
- 1962: Primeiro bootstrapping compiler para Lisp
- … hoje em dia: milhares de linguagens…

compiladores...

Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman.
Compilers: Principles, Techniques, and Tools (*2nd Edition*)

Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman.
Compilers: Principles, Techniques, and Tools (*2nd Edition*)

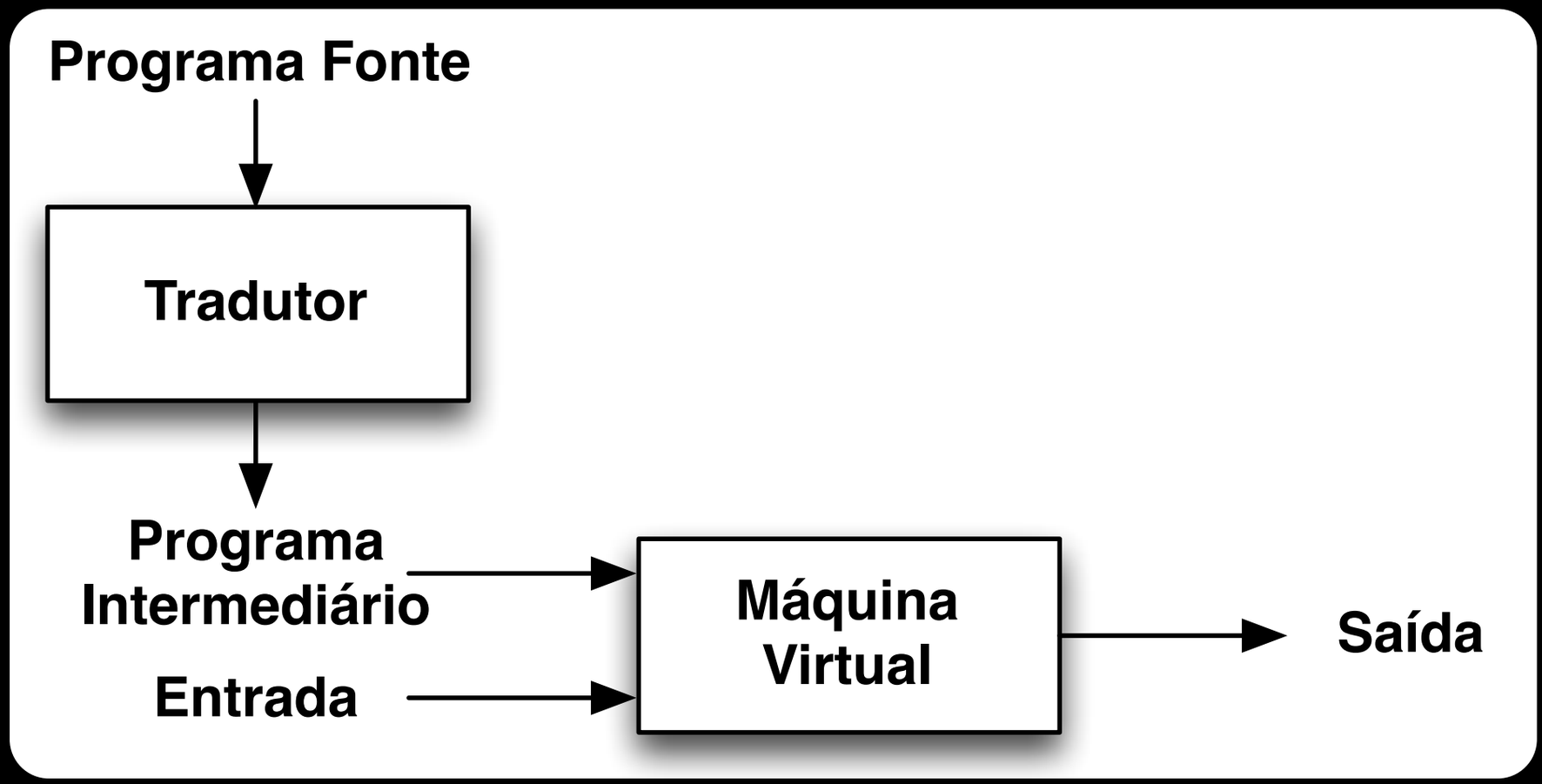
Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman.
Compilers: Principles, Techniques, and Tools (*2nd Edition*)

Compilador

vs.

Interpretador

E Java, se encaixa onde?



Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman.
Compilers: Principles, Techniques, and Tools (*2nd Edition*)

Princípios Fundamentais

O compilador deve preservar o sentido do programa sendo compilado.

muita pesquisa sendo feita em provar que um compilador é correto...

procure por CompCert, Verified Software Toolchain, Vellvm...

O compilador deve melhorar o programa fonte de alguma forma discernível.

What's in the box?

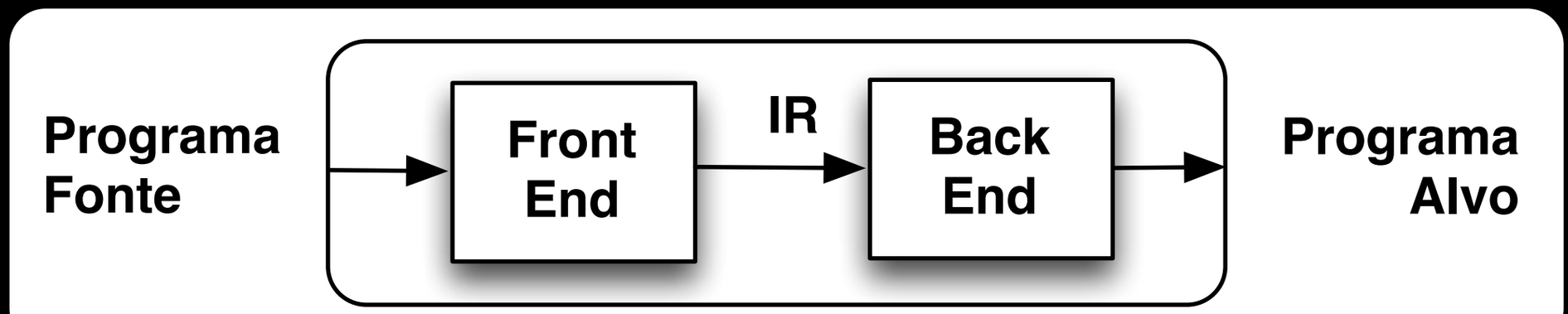
dividir para conquistar

tradução em passos

Duas Grandes Fases

Análise e Síntese

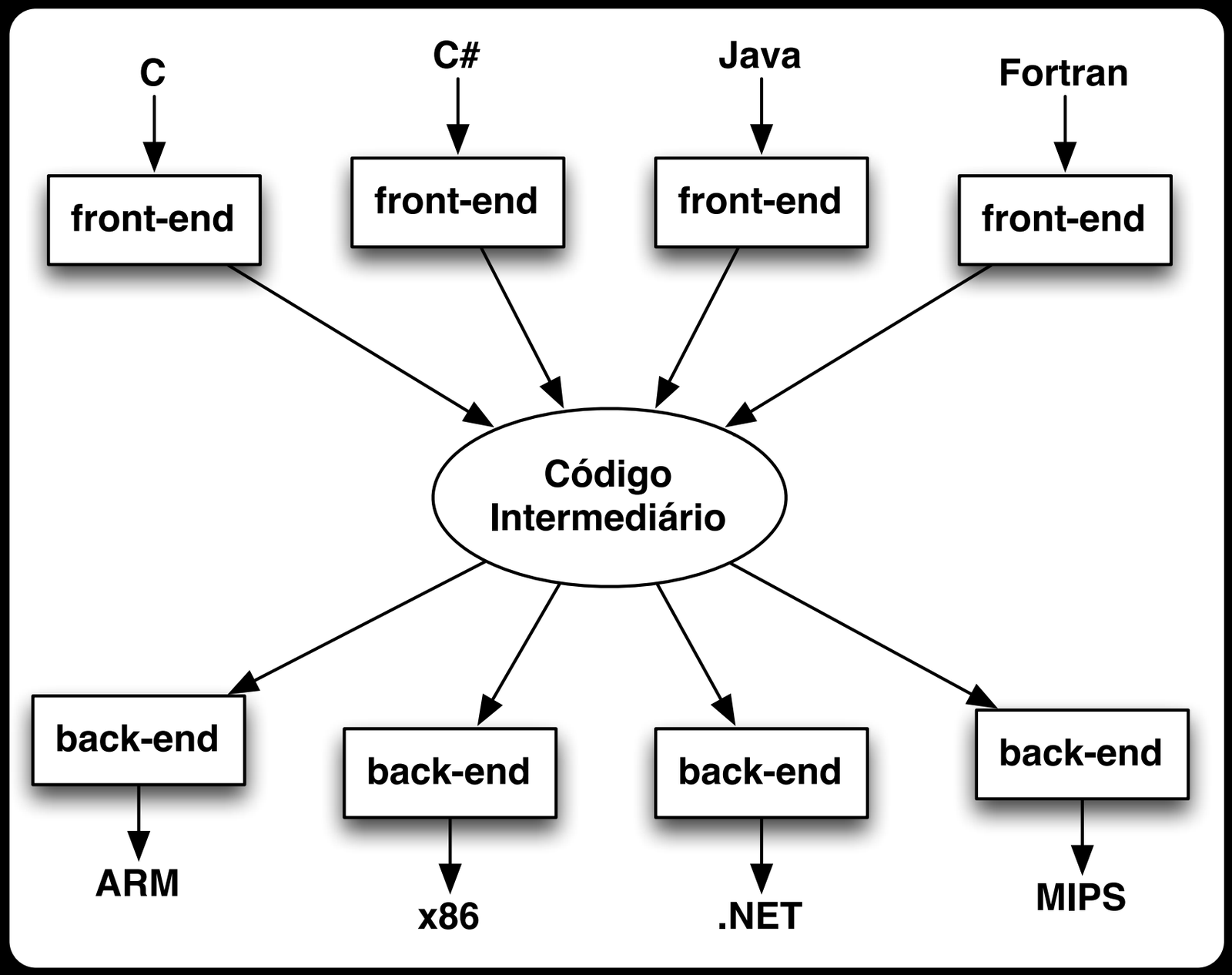
Adaptado de Keith Cooper, Linda Torczon.
Engineering a Compiler (*2nd Edition*)

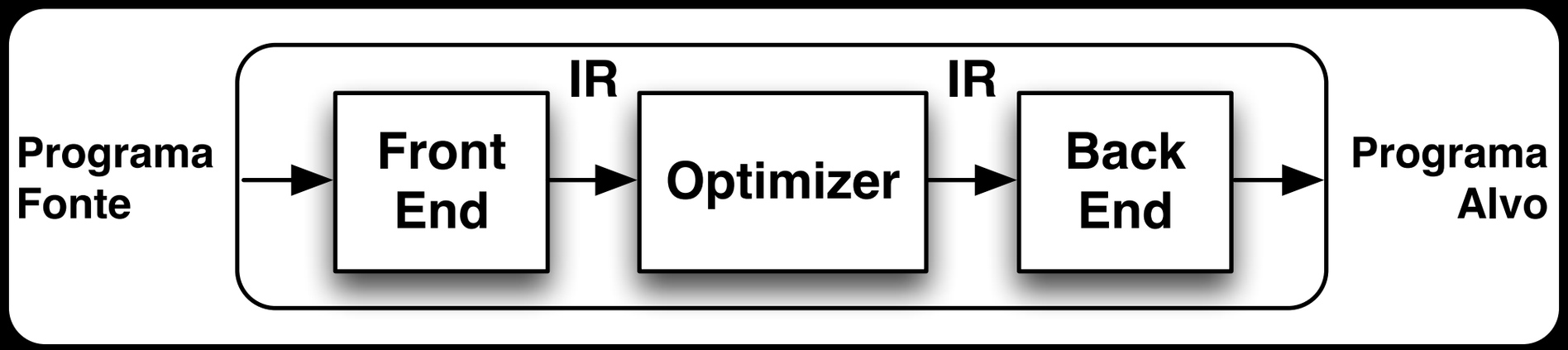


Adaptado de Keith Cooper, Linda Torczon.
Engineering a Compiler (*2nd Edition*)

quais as vantagens de separar as fases e usar representação intermediária?

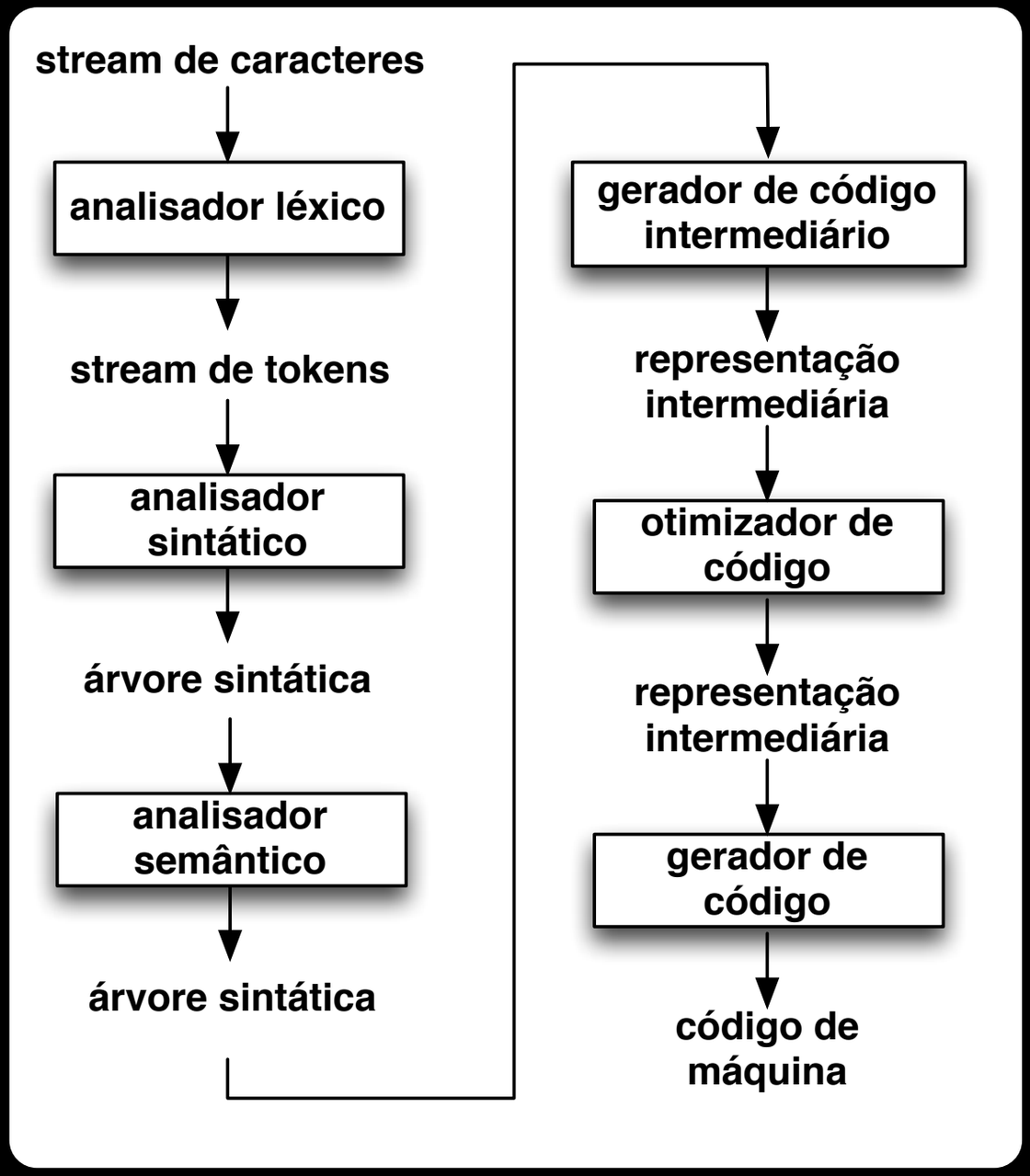
Separação permite criar múltiplos compiladores





Adaptado de Keith Cooper, Linda Torczon.
Engineering a Compiler (*2nd Edition*)

What's in the box?



Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman.
Compilers: Principles, Techniques, and Tools (*2nd Edition*)

Análise

(*front-end*)

Análise do programa fonte

- Análise léxica
  - Organiza caracteres de entrada em grupos, chamados tokens

- Análise sintática
  - Organiza tokens em uma estrutura hierárquica

- Análise semântica
  - Checa se o programa respeita regras básicas de consistência

Análise léxica (*scanning*)

- Lê os caracteres de entrada e os agrupa em sequências chamadas tokens
- Os tokens podem conter algum valor associado
- Os tokens são consumidos na fase seguinte (parsing)

Exemplo

```
position = initial + rate * 60
```

Exemplo

```
position = initial + rate * 60
```

Analisador Léxico

<**identificador**, 1>, <=>,
<**identificador**, 2>, <+>,
<**identificador**, 3>, <*>,
<**number**, 60>

Tabela de Símbolos

	nome	**tipo**	
1	position	-	…
2	initial	-	
3	rate	-	
…			

Análise Léxica

Analisador Léxico

O Projetista do compilador caracteriza o analisador léxico por meio de expressões regulares (ERs)

Análise Léxica

Analisador Léxico

O Projetista do compilador caracteriza o analisador léxico por meio de expressões regulares (ERs)

A geração do analisador léxico é automática a partir da definição das ERs.
Ver: JFlex, lex, FLEX, etc…

Tabela de símbolos

- Estrutura de dados usada para guardar identificadores e informações sobre eles.
- Por exemplo:
  - tipo do identificador
  - escopo: onde o identificador é válido no programa
  - se for um procedimento ou função: número e tipo dos argumentos, forma de passagem dos parâmetros e tipo do resultado.

Tabela de símbolos

	nome	**tipo**	
1	position	-	…
2	initial	-	
3	rate	-	
…			

Tabela de símbolos

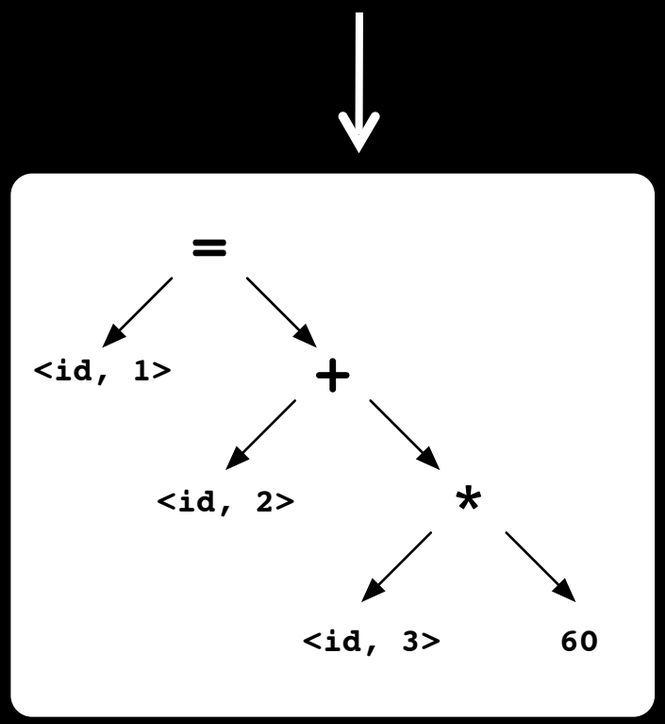
	nome	tipo	
1	position	-	…
2	initial	-	
3	rate		
	…		

Usada e atualizada em várias etapas do processo de compilação.

Análise sintática (*parsing*)

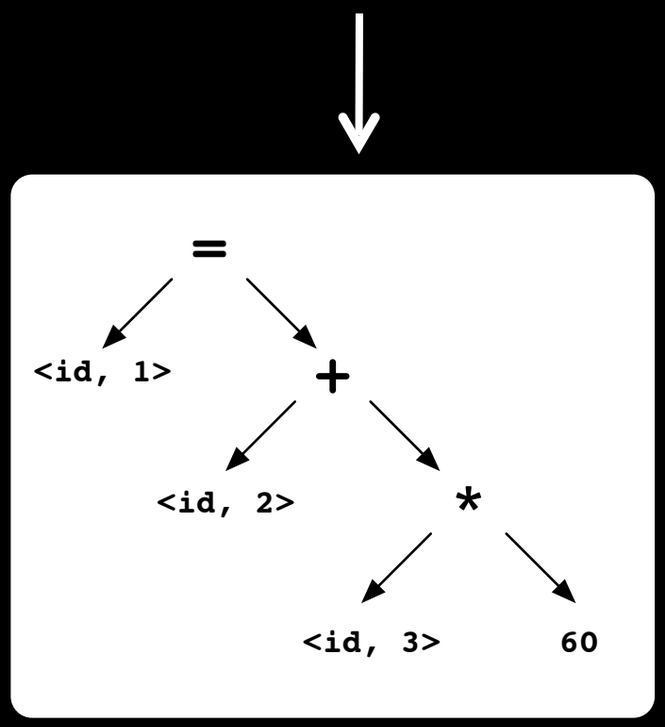
- Entrada: tokens gerados pela fase anterior
- Com base nos tokens, cria uma representação hierárquica para representar a estrutura gramatical do programa fonte
- Tipicamente utilizam-se árvores sintáticas, onde cada nó interno representa uma operação e os nós filhos representam argumentos

<**identificador**, 1>, <=>,
<**identificador**, 2>, <+>,
<**identificador**, 3>, <*>,
<**number**, 60>



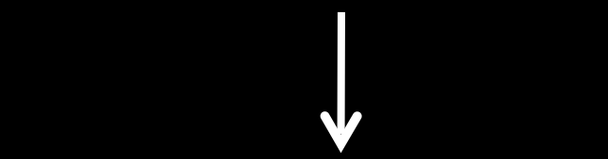
<**identificador**, 1>, <=>,
<**identificador**, 2>, <+>,
<**identificador**, 3>, <*>,
<**number**, 60>

Analisador Sintático

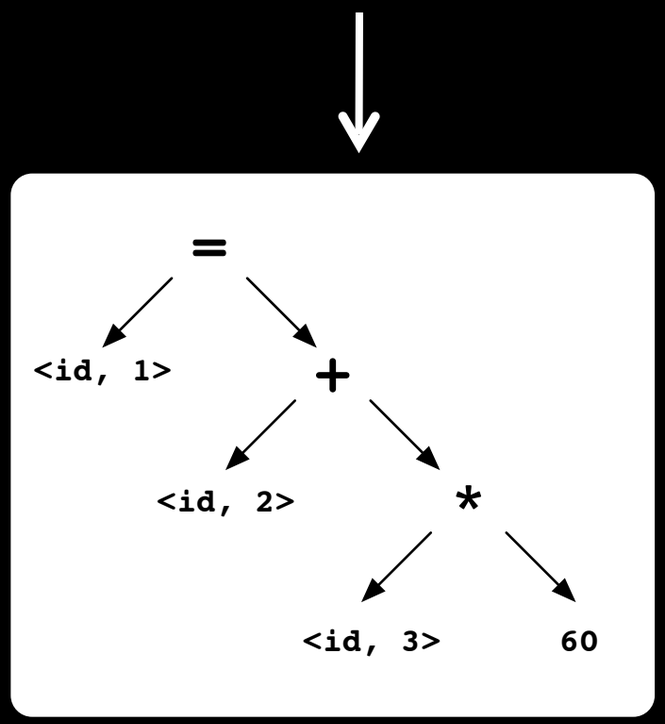


Gramática livre de contexto (BNF) caracteriza a linguagem

<**identificador**, 1>, <=>,
<**identificador**, 2>, <+>,
<**identificador**, 3>, <*>,
<**number**, 60>



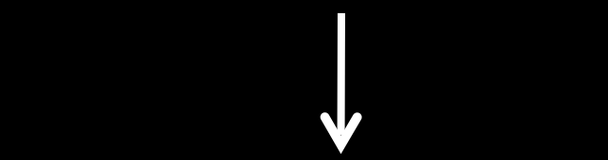
Analisador Sintático

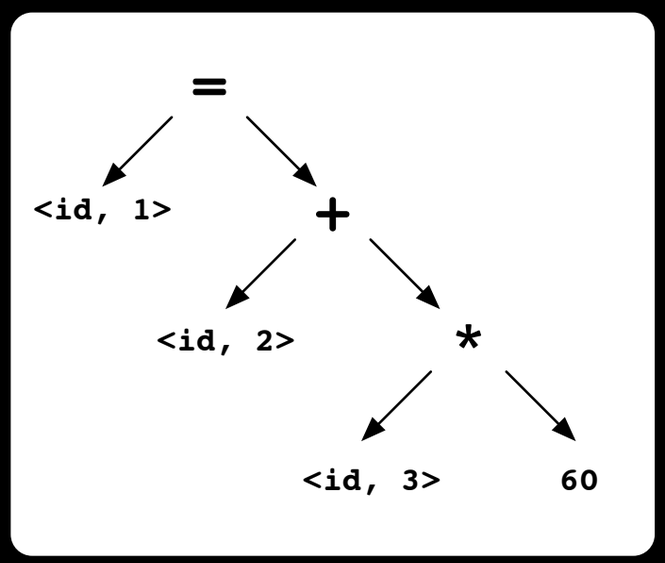


Gramática livre de contexto (BNF)

A geração do parser a partir de uma BNF pode ser automatizada. Vide: CUP, bison, yacc, etc…

<**identificador**, 1>, <=>,
<**identificador**, 2>, <+>,
<**identificador**, 3>, <*>,
<**number**, 60>





Gramática livre de contexto (BNF)

A geração do parser a par
ser
CU

Para cada classe gramatical da BNF haverá uma estrutura de dados correspondente

Análise Semântica

- Usa a árvore sintática e a tabela de símbolos para checar consistência semântica do programa com a definição da linguagem
- Por exemplo: *type checking*
- A linguagem pode permitir *coercions*
  - conversão automática de tipos compatíveis

Exemplo

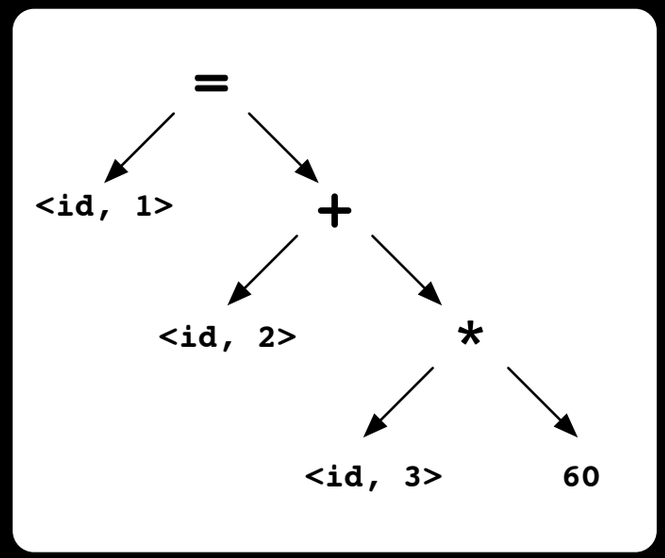
```
x = x + 3.0
```

é uma expressão correta?

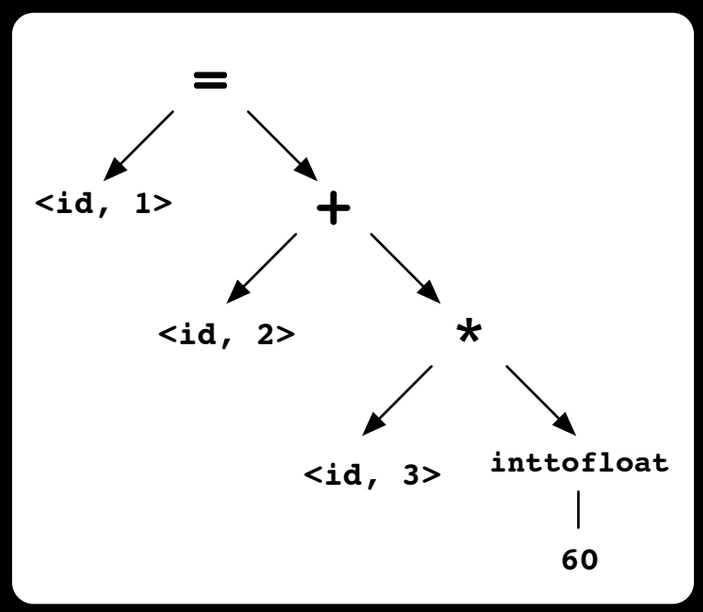
Verificação de tipos

```
x = x + 3.0
```

está sintaticamente correta, mas pode estar
semanticamente certa ou errada,
dependendo do tipo de `x`



Analisador Semântico



Intermediate Representations

Código Intermediário

- Durante o processo de tradução, um compilador pode construir uma ou mais representações intermediárias do programa
- Árvores sintáticas são representações intermediárias, usadas durante a fase de análise

Código Intermediário

- Muitos compiladores geram uma representação intermediária de baixo nível, similar à linguagem de máquina
- Esta linguagem deve, idealmente, ser fácil de ser produzida e fácil de ser analisada e transformada para a linguagem destino
- Exemplo: *three-address code*
  - no máximo três operandos por instrução

```
position = initial + rate * 60
```

```
id1 = id2 + id3 * inttofloat(60)
```

↓

Gerador de Código Intermediário

↓

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Otimização

Otimização de Código

- Realiza transformações no código com objetivo de melhorar algum aspecto relevante
  - desempenho, memória, tamanho do executável, consumo de energia, etc…
  - Análise + Transformação
  - Pode ser específica a uma arquitetura ou geral
    - Específica: register allocation
    - Geral: *Constant* (*folding and*) *propagation*

Análise

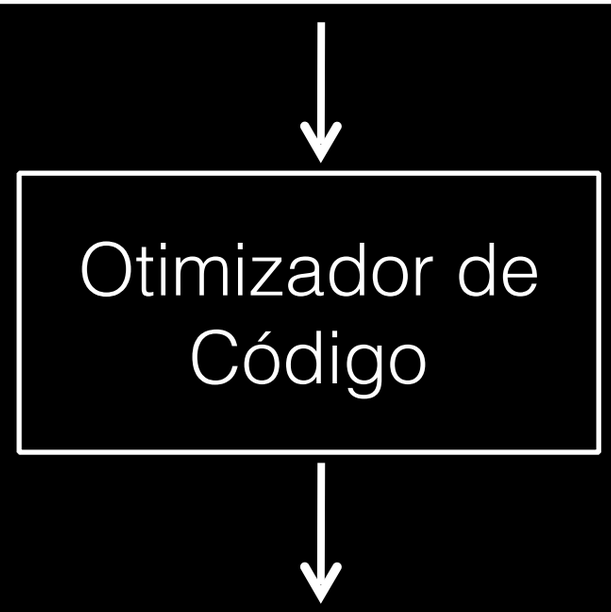
- Determina onde o compilador pode aplicar otimizações de forma segura e benéfica
- Análise de fluxo de dados
    - sistemas de equações envolvendo conjuntos, derivados da estrutura do código
- Análise de dependências
    - raciocinar sobre valores que podem ser assumidos por subexpressões

Transformação

- O compilador usa os resultados da análise para reescrever o código de forma mais eficiente
- Mover código, reduzir código…
- Variam em efeito, escopo, e análise necessária para habilitá-las

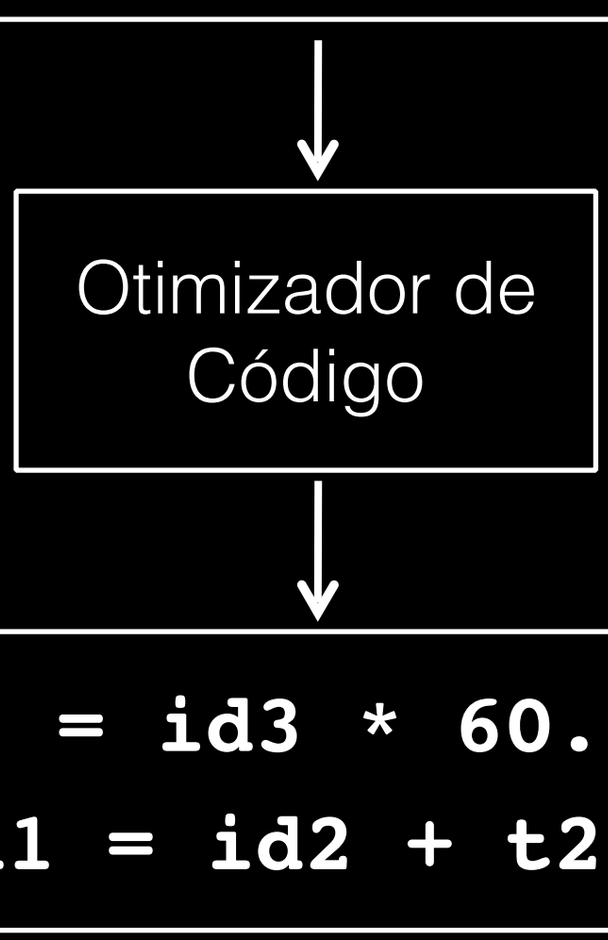
O que dá pra otimizar?

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```



Constant Propagation

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```




```
t2 = id3 * 60.0
id1 = id2 + t2
```

Síntese

(*back-end*)

Síntese (*back-end*)

- Seleção de instruções
- Agendamento de operações
- Alocação de registradores

Seleção de Instruções

- Reescrever operações de IR em operações de linguagem de máquina
- A princípio, comumente ainda abstrai quantidade de registradores simbólicos
- Pode se beneficiar de operações especiais na máquina alvo

Alocação de Registradores

- É mais eficiente realizar operações manipulando dados próximos a CPU, em registradores
- Como associar as muitas variáveis do programa a poucos registradores?
- Objetivo: minimizar *spilling*
    - processo de descarga e recarga de registradores a partir da memória

Instruction Scheduling

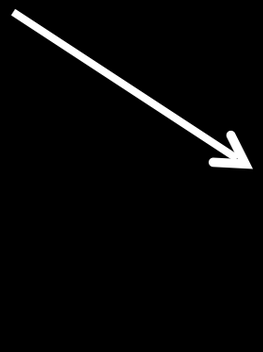
- Para produzir código eficiente, o gerador de código pode precisar reordenar operações
- Muitos processadores podem iniciar novas operações enquanto uma operação de longa latência executa
- Objetivo é minimizar número de ciclos ‘perdidos’ esperando por operandos

Geração de Código

- Vários problemas complexos surgem durante geração de código, e pra piorar, interagem
- Reordenar instruções pode acabar aumentando o número de registradores necessários
- Alocação de registradores pode criar ‘falsa’ sensação de dependência entre valores, o que prejudica *instruction scheduling*

Geração de Código

```
t2 = id3 * 60.0
id1 = id2 + t2
```



Gerador de Código

```
LDF   R2,   id3
MULF  R2,   R2   #60.0
LDF   R1,   id2
ADDF  R1,   R1, R2
STF   id1,  R1
```

```
position = initial + rate * 60
```

Por que estudar isto?

Por que estudar isto?

- Microcosmo da ciência da computação
- Aplicação bem sucedida de conceitos teóricos a problemas práticos
- Tecnologia presente em ferramentas de inspeção de código, descoberta de erros
- Melhorar conhecimento e habilidades de programação

O que você prefere?

```
C7 06 0000 0002
```

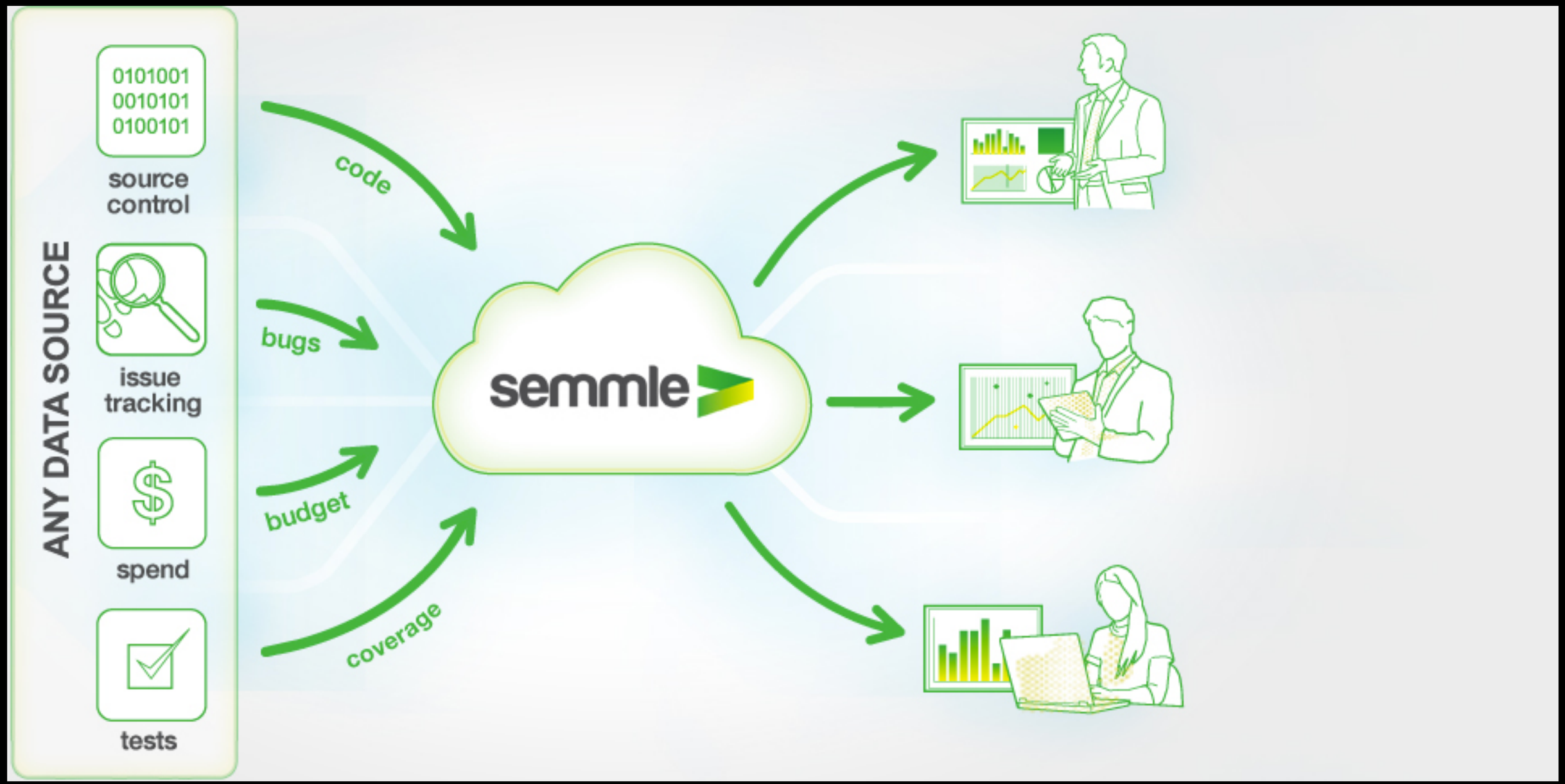
```
MOV X, 2
```

```
x = 2;
```

Talvez você precise fazer…

- Interpretador de linha de comando ou *scripting language*
- Leitura de linha de comando
- Jogos de computador, níveis e objetos definidos por usuário
- Leitura e processamento de XML, JSON, YAML
- *Syntax highlighting* em IDEs
- Estender LaTeX
- Portar código de uma linguagem a outra

Aplicações



https://semmle.com

lint

The Android `lint` tool is a static code analysis tool that checks your Android project source files for potential bugs and optimization improvements for correctness, security, performance, usability, accessibility, and internationalization.

In Android Studio, the configured `lint` and other IDE inspections run automatically whenever you compile your program. You can also manually run inspections in Android Studio by selecting **Analyze > Inspect Code** from the application or right-click menu. The *Specify Inspections Scope* dialog appears so you can specify the desired inspection profile and scope.

http://tools.android.com/tips/lint

http://findbugs.sourceforge.net

klocwork
a Rogue Wave Company

1.866.556.2967
info@klocwork.com

REQUEST A FREE TRIAL

PRODUCTS EVENTS RESOURCES CUSTOMERS HELP COMPANY

Products

CODE **CONFIDENCE.**

Get to value faster with secure, tested code.

APPLICATION SECURITY

CODE REFACTORING

STATIC CODE ANALYSIS

Application Security

Protect your code and your organization from harmful attack.

Strengthen your code.

CODE REVIEW

DEBUGGING COMPLEX CODE

http://www.klocwork.com

ACE

ACE: Your long term strategic partner for advanced system software solutions

HOME | COMPILER | CONSULTING | CORP

LLVM-TURBO

CoSy

SUPERTEST

SERVICES

SUPPORT

CoSy EXPRESS

PARTNERS

MyCoSy

SuperTest QS

Search

ACE Associated Compiler Experts bv

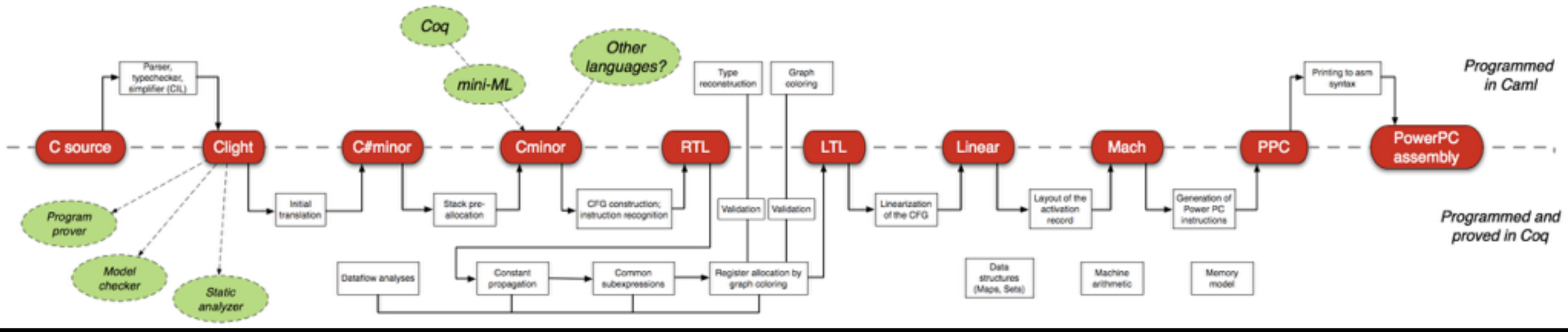
Home of CoSy, the compiler development system

ACE Associated Compiler Experts bv is a member of the ACE Group and was founded in 1996 to focus on the development and commercialization of products and services for professional compiler development.

Products

CoSy is the professional compiler development system from ACE. It is a mature, well-established compiler generation framework that provides compiler engineers with a complete and solid foundation. CoSy enables the compiler expert to excel at what he likes best: achieving best-in-class performance compilers for specific target architectures in a highly cost-effective manner. This powerful, versatile compiler development system is the result of intensive research and development by parent company ACE Associated Computer Experts, together with renowned European Research Institutes. CoSy Express is a novel OEM compiler generator technology from ACE that opens up new possibilities for architecture designers to quickly create optimizing compilers for new processor architectures. CoSy Express streamlines rapid compiler generation by delivering the architecture-specific power of CoSy to architecture designers and users of configurable processors. Integrated into hardware/software co-design EDA tools, CoSy Express enables semi-automated generation of optimizing compilers based upon a single architecture description. Incorporated in software tools for configurable architectures, CoSy Express enables end users of flexible architectures to instantly generate their own specific optimizing compiler and explore the optimal architecture-compiler configuration for the specific application domain. SuperTest is ACE's compiler test and validation suite that provides most thorough testing of C/C++ compilers. Based upon more than 3 decades of experience in compiler construction and validation, the over 3 million independent tests in SuperTest check every corner of the compiler, identifying issues in language conformance, as well internal functioning. Considering the damage a faulty compiler can do, can you afford not to test your compiler with SuperTest?

http://www.ace.nl/compiler



http://compcert.inria.fr

COVERITY®

Products Services Roles Customers Resources Blog **FREE TRIAL**

Synopsys Completes Acquisition of Seeker from Quotium

Code Advisor On Demand

Static Analysis in the Cloud

LEARN MORE

- Award-winning source code analysis in the cloud
- Easy to use: the root cause of each defect is explained
- Including OWASP Top 10 and CWE 25
- Integrated with GitHub

https://www.synopsys.com/

BIRL

O BIRL *(Bambam's "It's show time" Recursive Language)* é a linguagem de programação mais treze já inventada.
Deve ser utilizada apenas por quem realmente constrói fibra e não é água com código. É uma linguagem extremamente simples porém com poder para derrubar todas as árvores do parque Ibirapuera.
Programando em BIRL, é verão o ano todo!

https://birl-language.github.io

```
OH O HOME AI PO (MONSTRO SOMAR(MONSTRO A, MONSTRO B))
    BORA CUMPADE A + B;
BIRL

HORA DO SHOW
    MONSTRO A, B, RES;
    CE QUER VER ESSA PORRA? ("Entra com a e b ai cumpade!!\n");
    QUE QUE CE QUER MONSTRAO? ("%d %d", &A, &B);
    RES = AJUDA O MALUCO TA DOENTE SOMAR(A, B);
    CE QUER VER ESSA PORRA? ("Oh o resultado ai po: %d\n", RES);
    ELE QUE A GENTE QUER? (RES == 37)
        CE QUER VER ESSA PORRA? ("É 37 anos caralho!\n");
    NAO VAI DAR NAO
        CE QUER VER ESSA PORRA? ("Manda o double biceps!\n");
    BIRL
    BORA CUMPADE 0;
BIRL
```

https://birl-language.github.io

ArnoldC

A programming language based on the one liners of Arnold Schwarzenegger

More on Github | Download

Try it yourself

hello.arnoldc

```
IT'S SHOWTIME
TALK TO THE HAND "hello world"
YOU HAVE BEEN TERMINATED
```

http://lhartikk.github.io/ArnoldC/

O que é if688?

- *Teoria* necessária para o projeto de linguagens
- *Implementação* de compiladores
- Princípios e técnicas de construção de compiladores se aplicam a diversos domínios
    - linguagens de programação, arquitetura de computadores, linguagens formais e autômatos, algoritmos, engenharia de software

Objetivos Gerais da Disciplina

- Este curso explora os princípios, algoritmos, e estruturas de dados envolvidos no projeto e implementação de compiladores.
- O objetivo é fornecer fundamentos para desenvolvimento da compreensão da teoria e prática de compiladores, e de questões envolvidas na implementação de linguagens.

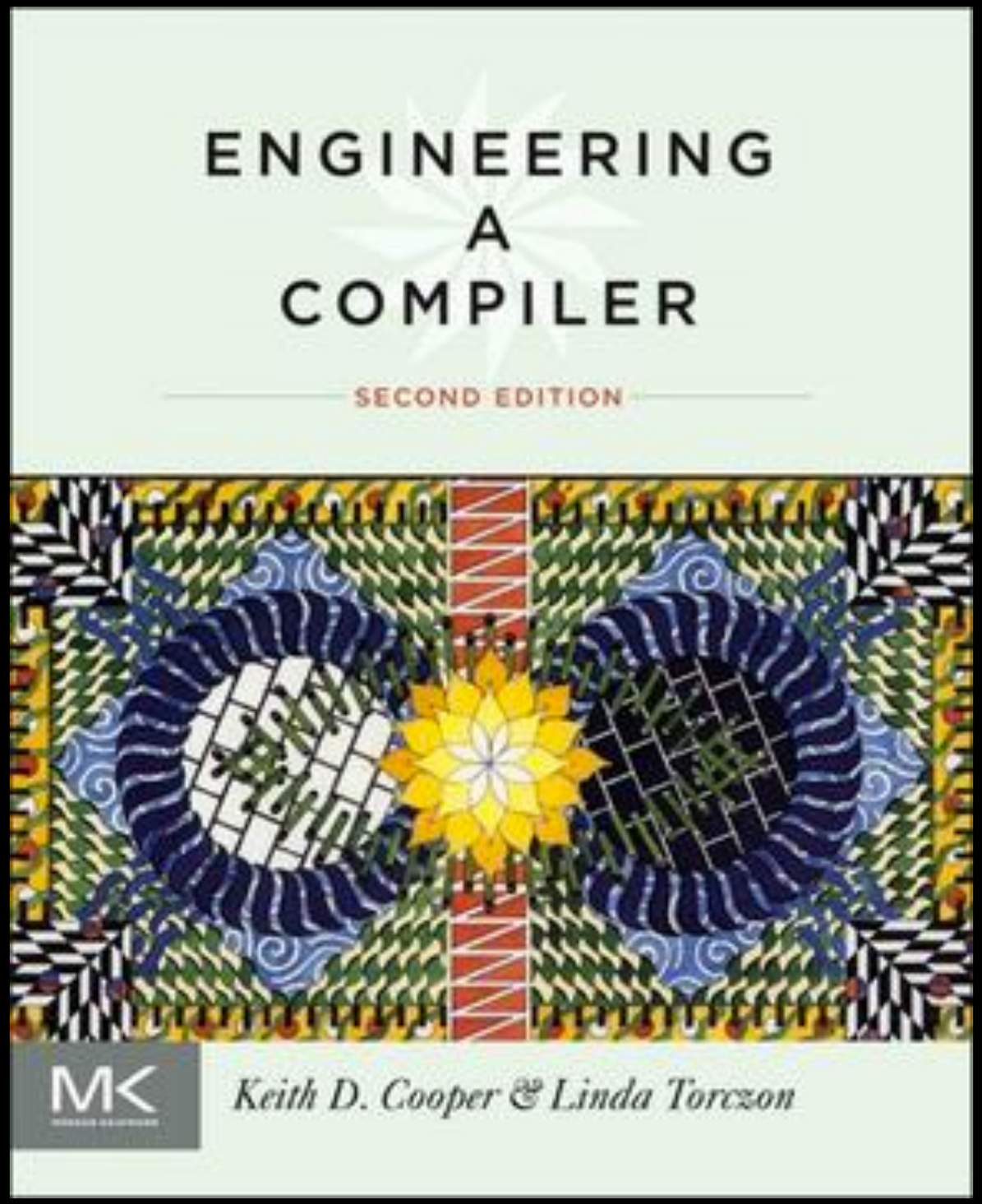
Durante a aula...

- Evite distrair os demais alunos, com conversas e/ou uso de dispositivos eletrônicos
  - (normalmente devem estar desligados!)
- Se você tem uma dúvida, fique à vontade para perguntar
- Se você não entendeu alguma explicação, fique à vontade para perguntar
- Teremos chamada ao final de cada aula

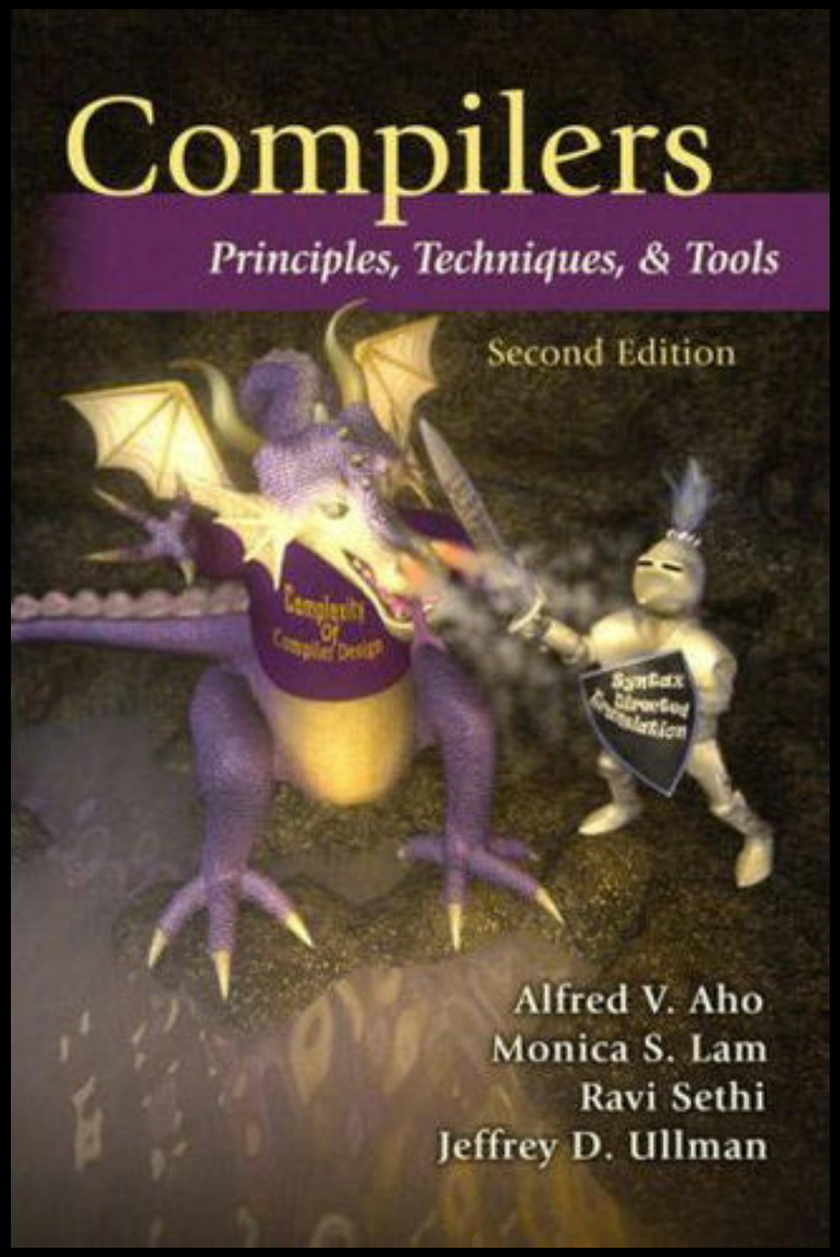
Conteúdo Planejado

- Introdução a Compiladores
- Conceitos Básicos de Linguagens
- Análise Léxica
- Análise Sintática
- Análise Semântica
- Representação de Código Intermediário
- Ambiente de Execução
- Análise e Otimização
- Geração de Código

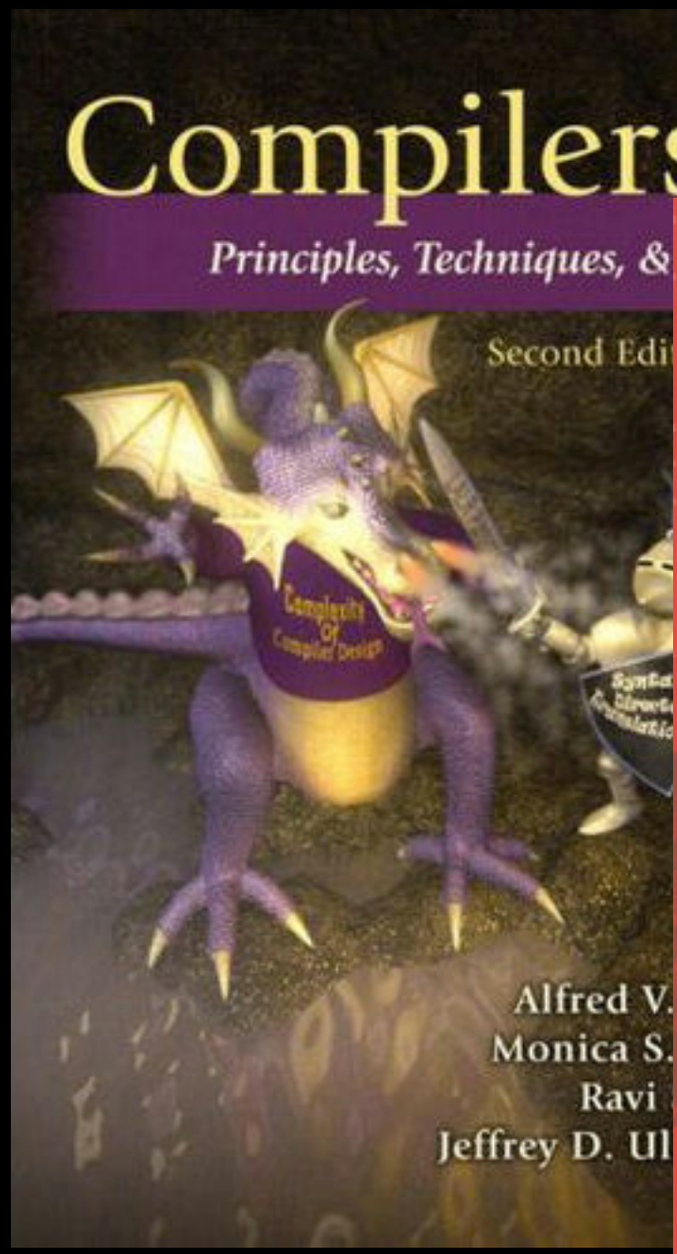
Bibliografia Básica

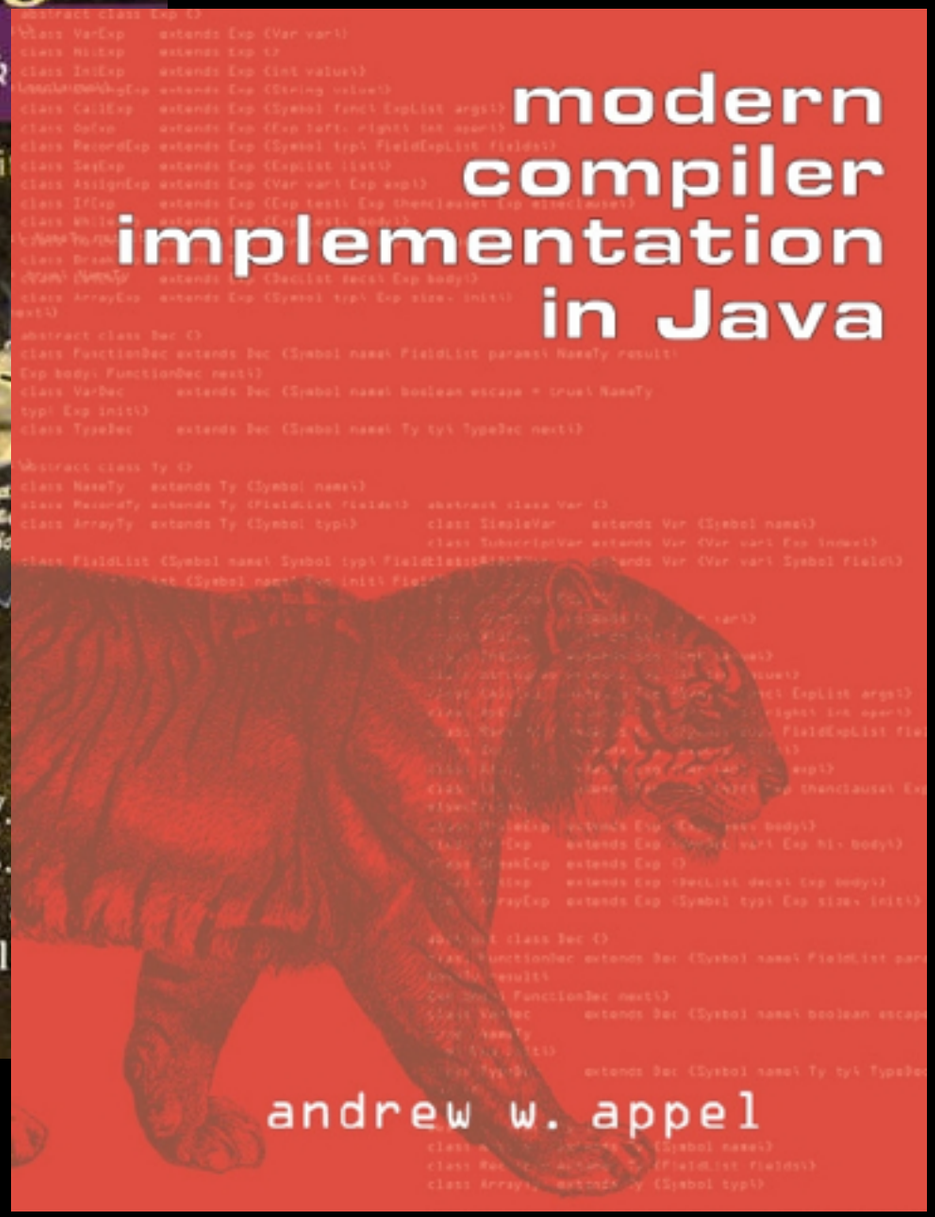


Bibliografia Complementar

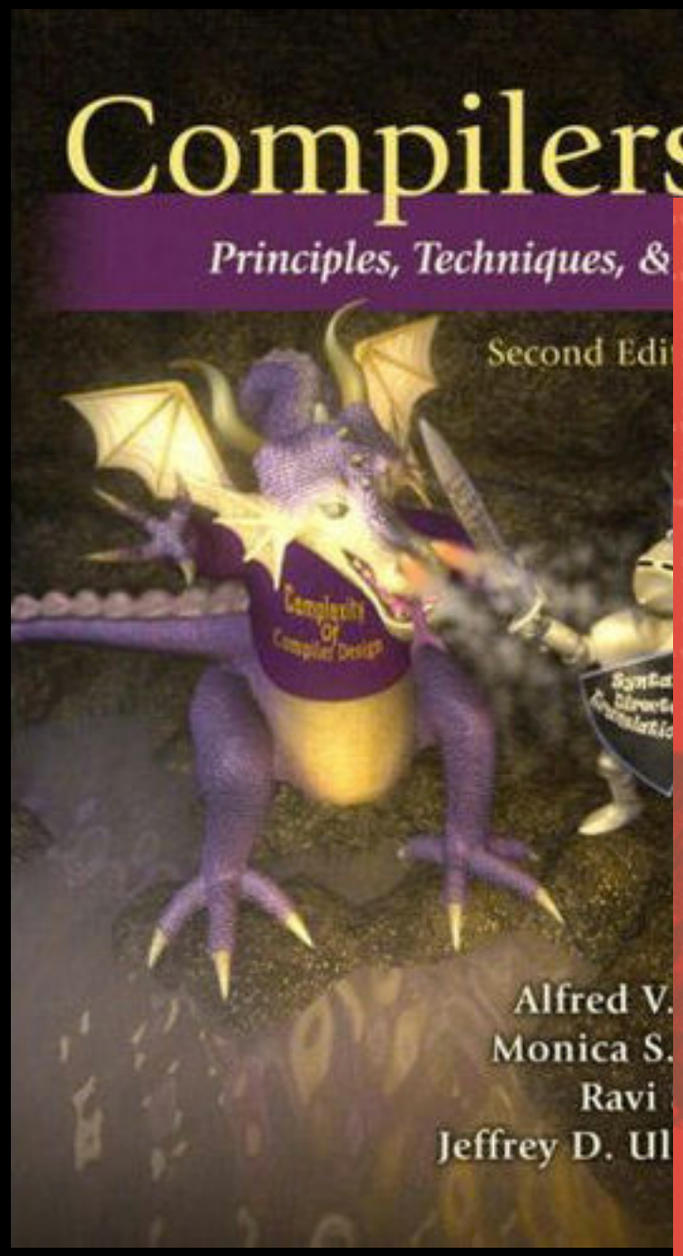


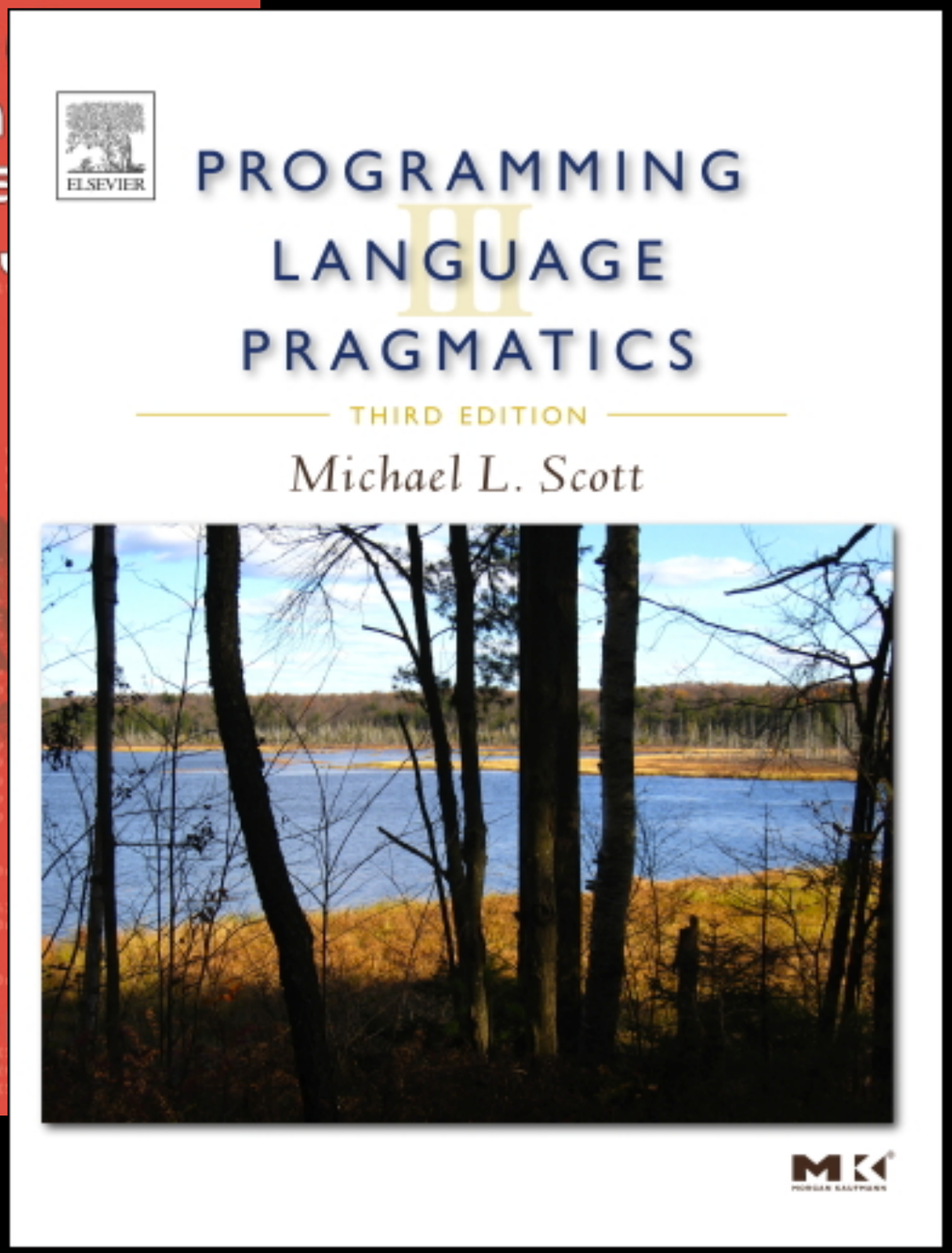
Bibliografia Complementar





Bibliografia Complementar





Bibliografia

- Keith Cooper & Linda Torczon. ***Engineering a Compiler*** (*2nd Edition*). Morgan Kaufmann, 2012

- Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. ***Compilers: Principles, Techniques, and Tools*** (*2nd Edition*). Addison-Wesley, 2006

- Andrew Appel & Jens Palsberg. ***Modern Compiler Implementation in Java*** (*2nd edition*). Cambridge University Press, 2003.

- Michael Scott. ***Programming Language Pragmatics*** (*3rd edition*). Morgan Kaufmann, 2009.

achou que não ia
falar de prova?

Avaliação

- (**N1**+**N2**)/2, onde:
- **N1** = Prova1 (60%) + Exercícios (40%)
  - Prova1 = Teste com assunto dado até o momento
  - Exercícios = Tarefas passadas durante primeira unidade
- **N2** = Prova2 (60%) + Exercícios (40%)
  - Prova2 = Teste com assunto dado a partir de Prova1
  - Exercícios = Tarefas passadas durante segunda unidade
- **Final:** Teste com todo o assunto da matéria

Contato

- Slack: **http://if688.github.io**
  Atualizando ainda…

- Slack: **https://if688.slack.com/signup**
  Faça o login com seu email do CIn.

- E-mail: **lmt@cin.ufpe.br**
  Sala C012

Teoria e Implementação de Linguagens Computacionais

aka Compiladores (IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt