

Breve Histórico & Conceitos Básicos



compiladores

interpretadores

montadores

filtros

pré-processadores

carregadores

linkers

compilador cruzado (cross-compiler)

auto-compilável (bootstrapping)

auto-residente

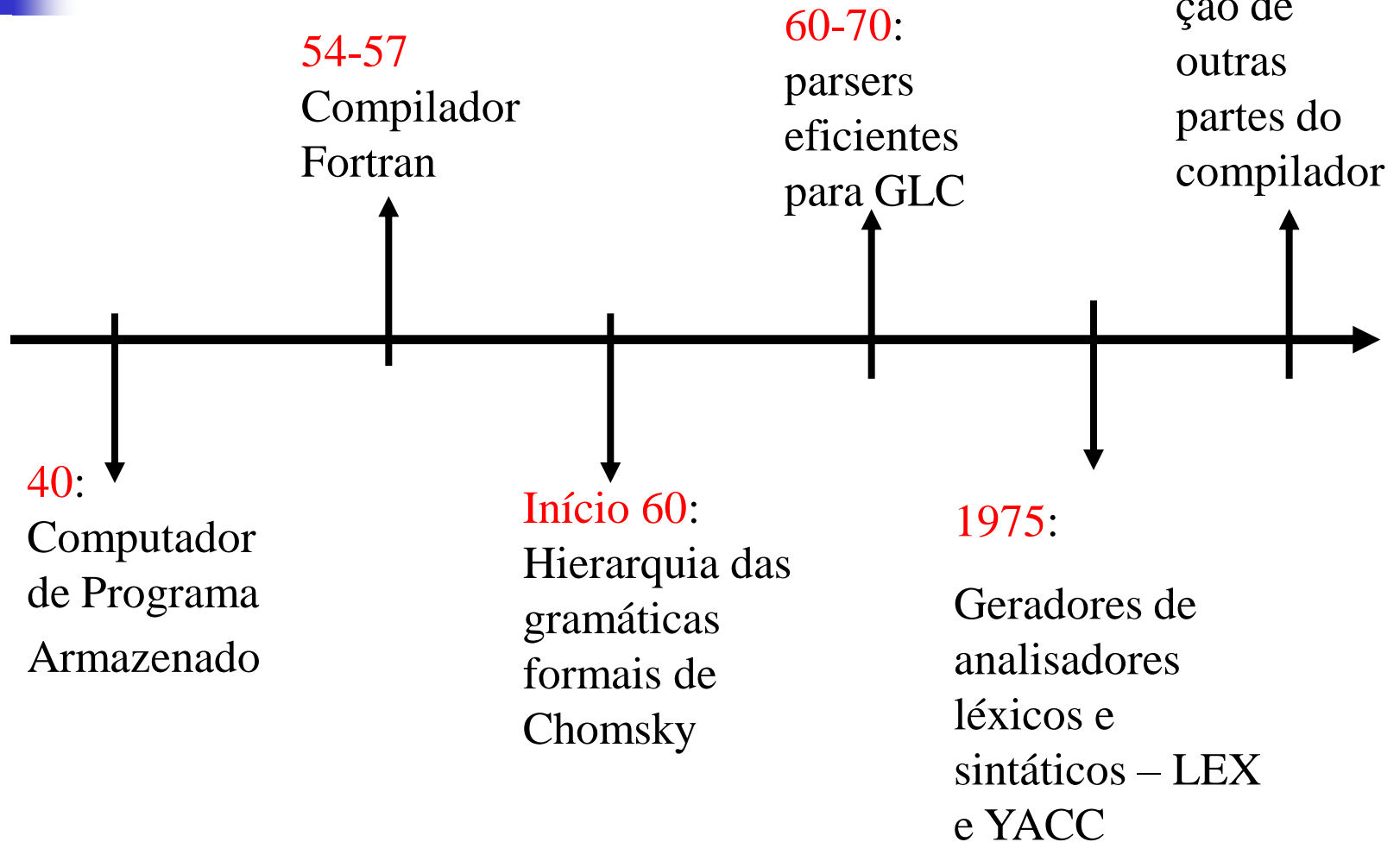
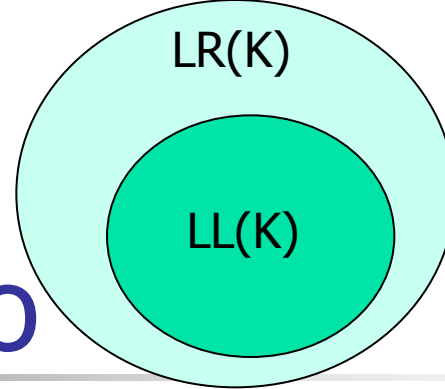
compiler compilers (ou translator writing systems, ou compiler generators)

estrutura de um compilador

formas de organização de um compilador

processo de execução de uma linguagem de alto-nível

Linha do Tempo





Histórico

- Computador de programa armazenado (von Neumann) – final da década de 40
 - Inicialmente os programas eram escritos em código de máquina (computadores tinham pouca memória)
- Linguagem de montagem para facilitar a programação
 - Necessidade de um programa (montador) para traduzir para código de máquina
- Próximo passo: linguagem de alto nível
 - Temor: que fosse impossível ou que o código objeto fosse tão ineficiente a ponto de ser inútil!



Histórico

- Primeira Linguagem de alto nível (Fortran) e seu compilador, graças a Backus, entre 1954 e 1957
 - Muito esforço, pois grande parte dos processos de tradução ainda não eram entendidos naquele tempo,
 - Entretanto com relação a otimização de código estava a frente no seu tempo.

- Chomsky inicia estudos da estrutura da linguagem natural (inglês) no fim dos 50 – início dos 60's
 - Complexidade das gramáticas e os algoritmos para reconhecê-las: tipo 0, 1, 2, 3.

- Determinação de algoritmos eficientes para reconhecer gramáticas livres de contexto: 1960 e 1970.
 - Embora tratem de um subconjunto das GLC, as gramáticas determinísticas LL(K) e LR(K) são capazes de representar todas as linguagens de programação, sintaticamente.



Histórico

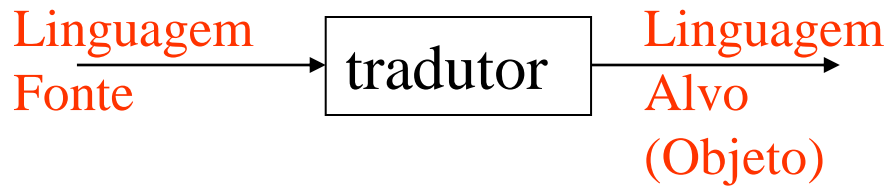
- Vantagens da separação entre reconhecimento de tokens de uma linguagem e suas regras (sintaxe)
 - levou ao uso de métodos para expressar os tokens (uso de autômatos finitos e expressões regulares)
- O estudo dos autômatos finitos e ER
 - Geradores de analisadores léxicos (Lex – unix em 1975)



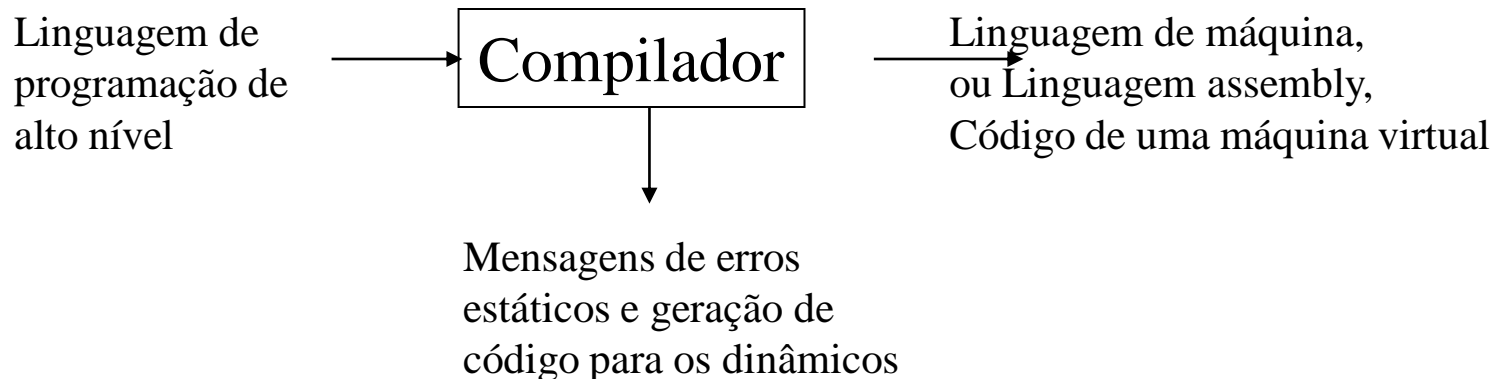
Histórico

- Com uma maior compreensão da análise sintática surgiram métodos para automatizar partes do desenvolvimento de compiladores
 - Geradores de analisadores sintáticos (YACC – unix em 1975), que são usados para gerar compiladores
 - Vejam outros em: <http://catalog.compilertools.net/>
- Década de 70 e 80 houve projetos para automatizar a geração de outras partes do compilador (geração de código)
- Um exemplo é o BEG (back end generator - <http://www.hei.biz/beg/>)
 - *BEG was used to produce an industrial compiler family for C, Modula-2, Fortran-77, and Pascal, for the SPARC processors.*
 - mas a maioria dos cc são na verdade parser generators com a possibilidade de incluir código para Asem e Geração. Veja: http://en.wikipedia.org/wiki/Comparison_of_parser_generators

Compilador



Compilador é um tradutor que transforma automaticamente uma linguagem de programação de **alto nível** para alguma outra forma que viabilize sua execução





Tarefas

- Deve transformar a L.F. em uma L.A. equivalente
- Deve informar os erros do programa fonte, caso existam
- Permite que os programas sejam independentes de máquina (portabilidade)



Propriedades de um bom compilador

- Principal: gere código corretamente
- Seja capaz de tratar de programas de qualquer tamanho que a memória permita
- **Velocidade da compilação** não é a característica principal
- Tamanho do compilador já não é mais um problema atualmente
- User-friendliness se mede pela qualidade das mensagens de erros
- A importância da velocidade e tamanho do código gerado depende do propósito do compilador --- **velocidade do código** vem em primeiro lugar



Tecnologia de Compiladores

- Os primeiros compiladores foram construídos *ad hoc*, sem técnicas, eram vistos como processos complexos e de alto custo
 - Hoje, o processo é bem entendido, sendo a construção de um compilador uma rotina, embora a construção de um compilador eficiente e confiável seja ainda uma tarefa complexa.



Tecnologia de Compiladores

- A tecnologia de compiladores é largamente empregada em muitas áreas:
 - linguagens de formatação de textos (nroff, troff, latex, sgml)
 - “silicon compilers” que transformam uma linguagem de especificação de circuitos VLSI em um projeto de circuitos
 - linguagens de acesso a banco de dados (query languages)
 - editores de texto orientados a sintaxe
 - analisadores estáticos de programa que podem descobrir variáveis não inicializadas, código que nunca será executado, etc.
 - checadores de entrada de dados em qualquer sistema
 - interpretadores de comandos de um S.O.



Formas de Classificar Compiladores


- De acordo com o **tipo** de linguagem objeto gerado

- De acordo com o **formato** do código objeto gerado

Alternativas para a saída de um compilador (1/4)

- Os compiladores podem ser distinguidos de acordo com o **tipo de linguagem objeto** que eles geram:

- **Código de Máquina Puro:** geram código para um conjunto de instrução de máquina sem assumir a existência de rotinas do SO ou rotinas de biblioteca.

Isto quer dizer que o compilador não está contando com nenhum “ambiente de execução” 

Esta abordagem é rara e é utilizada para implementar as linguagens que serão utilizadas para escrever o SO e outros programas de baixo nível.

- **Código de Máquina Aumentado:** geração de código para uma arquitetura aumentada com as rotinas do SO (salvar registradores, acessar BIOS) e de suporte à linguagem (I/O, alocação dinâmica, e funções matemáticas) que devem ser carregadas com o programa.

A combinação de código de máquina do compilador + rotinas do SO + rotinas de suporte é chamada “Máquina Virtual”.

Esta abordagem é mais freqüente.

Alternativas para a saída de um compilador (2/4)

- **Código de Máquina Virtual:** caso extremo de definição de uma máquina virtual em que o código gerado é composto inteiramente de instruções virtuais: geração para uma máquina hipotética.

Existe um interpretador que roda sobre o hardware físico e emula o hardware de um projeto diferente.

Abordagem atrativa pela facilidade de transportar para máquinas diferentes.

Se a máquina virtual é simples, escrever um interpretador é tarefa fácil (geralmente é a escolha para um “compilador didático”)


Exemplo: “Pascal P-Compiler” 

Alternativas para a saída de um compilador (3/4)

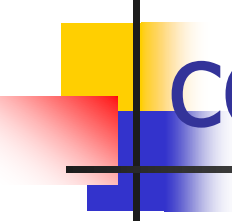
Compiladores também podem ser distinguidos pelo **formato do código objeto** que eles geram:

- **Linguagem Assembly (Montagem):** esta abordagem simplifica a tradução pois várias decisões são deixadas para o montador (endereço de jumps, por exemplo).
 - Pode-se checar a saída e verificar a corretude (assembly é mais legível). É útil para cross-compilers (compilador que roda em uma máquina e gera código para outra) pois gera um arquivo facilmente transportável.

Esta abordagem não é comum, pois precisa de um passo adicional feito pelo montador, que é lento. Algumas vezes o código em assembly é um produto extra de alguns compiladores (pseudo-assembly languages) para checagem de corretude.

- **Código Binário Relocável:** código em que as referências externas e endereços de memória não estão decididos (são endereços  simbólicos).
 - Um passo extra de link-edição (link-editor) é necessário para adicionar as rotinas de biblioteca e outras rotinas pré-compiladas (.obj) que são referenciadas dentro do programa compilado, e assim produzir o código binário absoluto. (Permite compilação independente).

Alternativas para a saída de um compilador (4/4)



O código relocável e o código assembly permitem compilação modular, interfaces com outras linguagens (cross-languages references, isto é, chamadas de rotinas em assembly ou em outras linguagens de alto nível -- - Amzi Prolog) e rotinas de biblioteca.

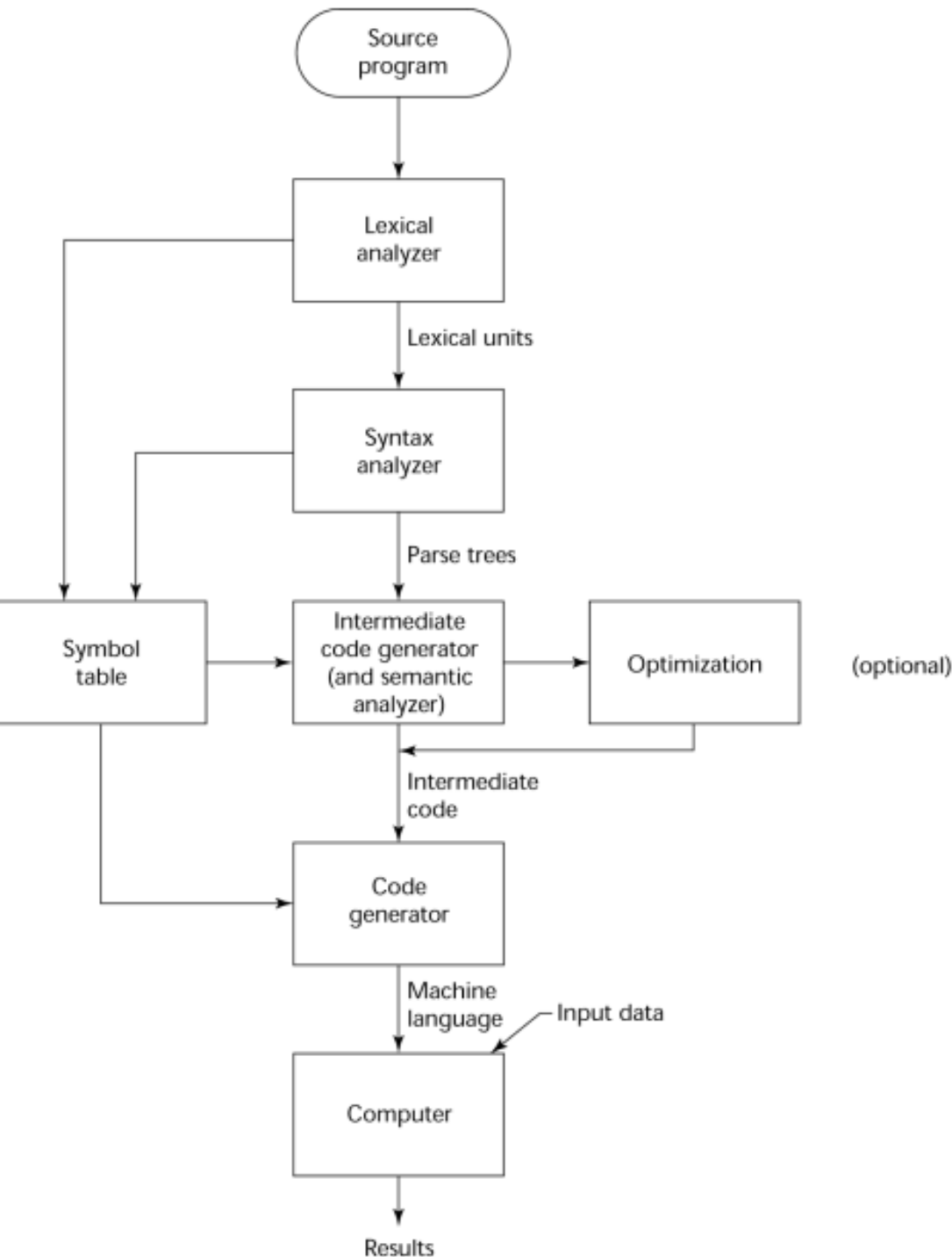
- **Imagem de Memória (Load-and-Go):** a saída do compilador é carregada diretamente na memória e executada ao invés de ser deixada em um arquivo.
 - Esta abordagem é rápida pois não exige link-edição, mas o programa deve ser recompilado a cada execução. Por ser mais rápido é útil para a fase de depuração e uso de estudantes novatos. Exemplo: Turbo Pascal.



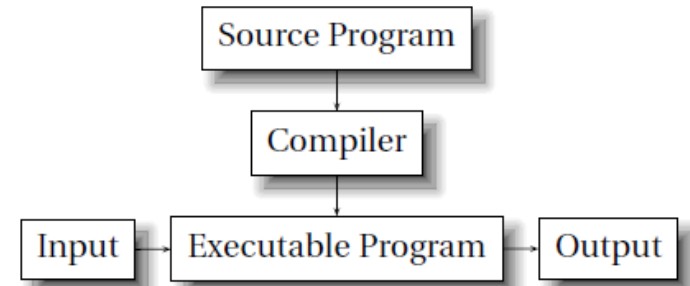
Métodos de Implementação

- Compilação
 - Programas são traduzidos em linguagem de máquina
- Interpretação Pura
 - Programas são interpretados por outro programa (interpretador)
- Sistemas Híbridos
 - Oferecem um compromisso entre compiladores e interpretadores puros

Compilador

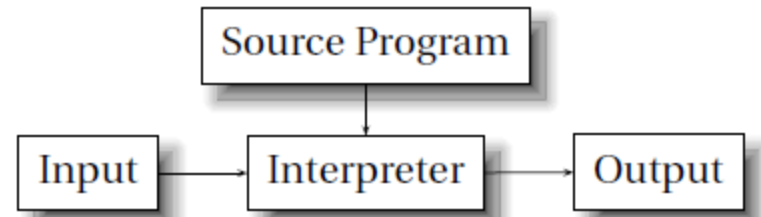


(optional)



Tradução lenta
Execução rápida

Interpretador



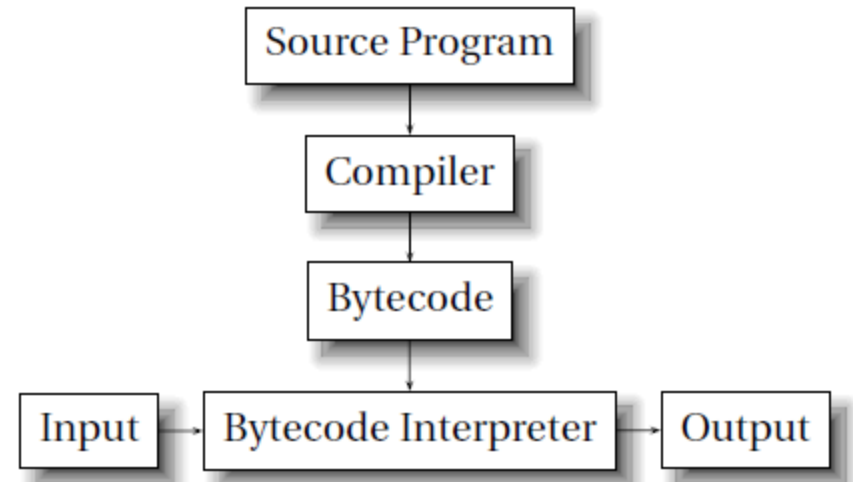
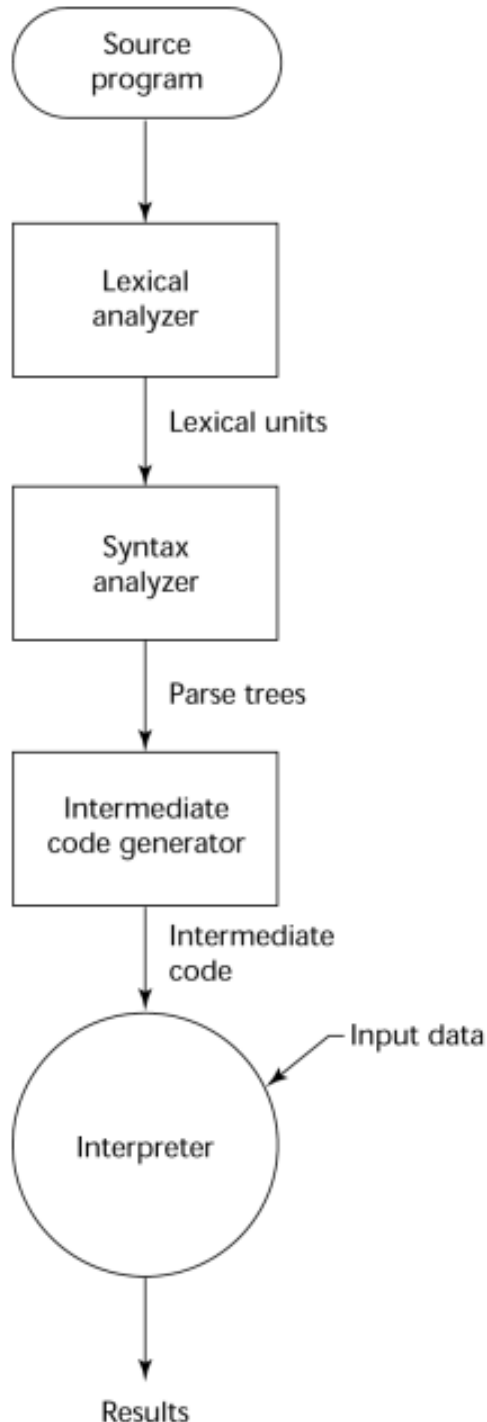
- Não há tradução
- Execução lenta (10 a 100 vezes mais lento que programas compilados)
- Frequentemente requer mais espaço
- Atualmente raro para as linguagens tradicionais, mas
 - está havendo um retorno com as linguagens de script para a Web (por exemplo, JavaScript, PHP)



Sistemas Híbridos

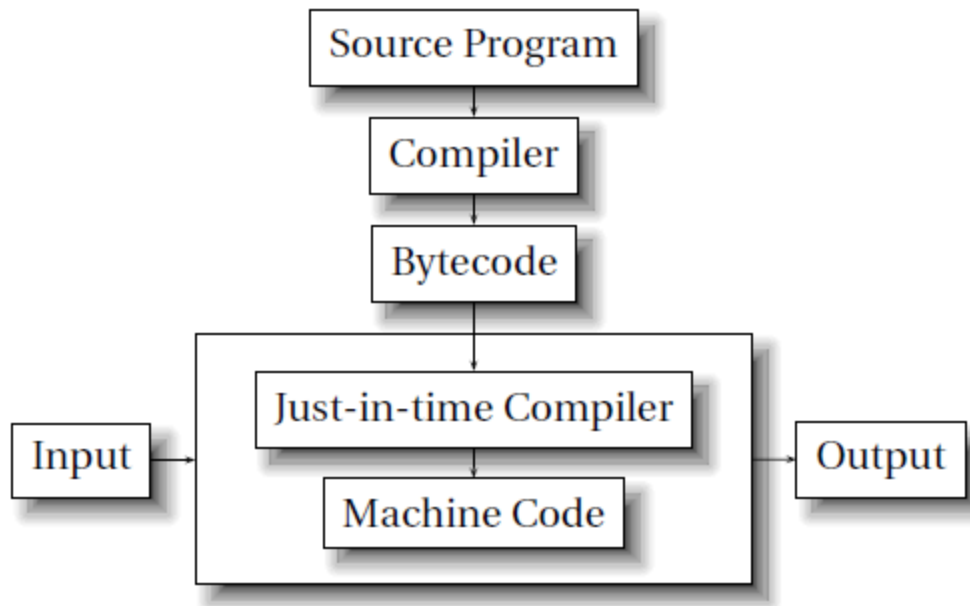
- Há a tradução para uma linguagem intermediária para facilitar a interpretação
 - Mais rápido que interpretação pura
 - Exemplos:
 - Programas Perl são compilados parcialmente para detectar erros antes da interpretação.
 - Implementações iniciais de Java foram híbridas;
 - o código intermediário, *byte code*, fornece portabilidade para qualquer máquina que tenha um **interpretador de byte code e um ambiente de execução** (juntos, são chamados de *Java Virtual Machine*)

Interpretador de Bytecodes



O conceito não é novo...Pascal P-code difundiu a linguagem Pascal

Compilador Just in Time



Une bytecodes com
compilação dinâmica

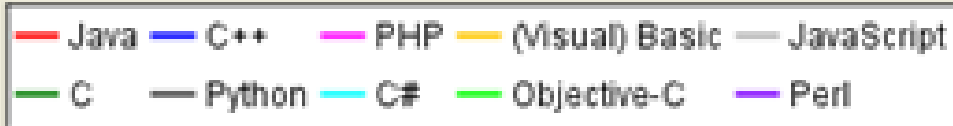
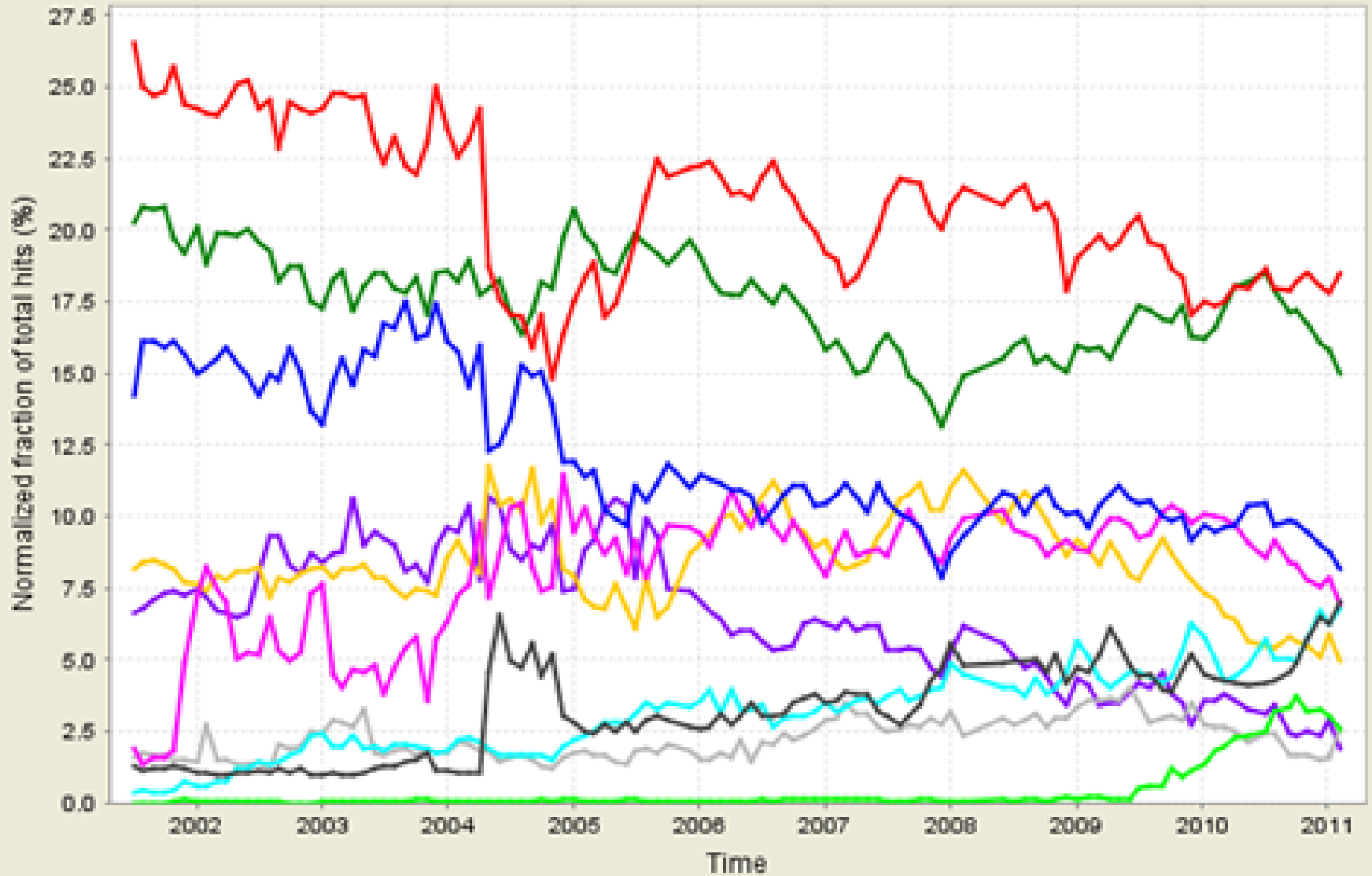
- Traduz inicialmente para uma linguagem intermediária (bytecode)
- Então compila a linguagem intermediária dos subprogramas em código de máquina nativa quando eles são chamados (tempo de execução) para executar na máquina nativa
- Este código é mantido para chamadas subsequentes
- Sistemas JIT são muito usados para Java

TIOBE Programming Community Index for February 2011

Position Feb 2011	Position Feb 2010	Delta in Position	Programming Language	Ratings Feb 2011	Delta Feb 2010	Status
1	1	=	Java	18.482%	+1.13%	A
2	2	=	C	14.988%	-1.62%	A
3	4	↑	C++	8.187%	-1.28%	A
4	7	↑↑↑	Python	7.038%	+2.72%	A
5	3	↓↓	PHP	6.973%	-3.03%	A
6	6	=	C#	6.809%	+1.79%	A
7	5	↓↓	(Visual) Basic	4.924%	-2.13%	A
8	12	↑↑↑↑	Objective-C	2.571%	+0.79%	A
9	10	↑	JavaScript	2.558%	-0.08%	A
10	8	↓↓	Perl	1.907%	-1.69%	A
11	11	=	Ruby	1.615%	-0.82%	A
12	-	=	Assembly*	1.269%	-	A-
13	9	↓↓↓↓	Delphi	1.060%	-1.60%	A
14	19	↑↑↑↑↑	Lisp	0.956%	+0.39%	A
15	37	↑↑↑↑↑↑↑↑	NXT-G	0.849%	+0.58%	A--
16	30	↑↑↑↑↑↑↑↑	Ada	0.805%	+0.44%	A--
17	17	=	Pascal	0.735%	+0.13%	A
18	21	↑↑↑	Lua	0.714%	+0.21%	A--
19	13	↓↓↓↓↓	Go	0.707%	-1.07%	A--
20	32	↑↑↑↑↑↑↑↑	RPG (OS/400)	0.628%	+0.27%	A--

Atualizado mensalmente. Usa máquinas de busca(Google, MSN, Yahoo!, Wikipedia e YouTube) para calcular o número mundial de usuários, cursos e vendedores de linguagens pagas. Não se refere à melhor linguagem nem àquela para qual muitas linhas foram escritas.

Tiobe Programming Community Index



Montadores, Filtros, e Pré-processadores



Linguagem
de montagem

Montador

Linguagem
de máquina

Linguagem de alto nível
(dialeto do Algol)

Filtro

Linguagem de alto nível
muito semelhante à Lf
(Algol 60)

Lf não preparada
(extensões)

Pré-processador

Lf
preparada

Compilador

Processamento de macros,
inclusão de arquivos

Interpretadores

Em vez de traduzir de uma vez o programa-fonte para então executá-lo, decodifica unidades básicas do programa para executá-las imediatamente (não mantém o código-objeto para sempre). Este é um enfoque de interpretação (TRADUZ-EXECUTA), porém consome muito tempo. Ex: BASIC.

- Se a velocidade de compilação for requisito importante prefere-se um compilador
 - Dez ou mais vezes mais rápido que um interpretador
- Uma estratégia mais eficiente envolve a aplicação de técnicas de compilação:



- Neste caso o interpretador refere-se não só ao programa que **simula** a execução do código Intermediário, mas a todo o processo.

Compilador cruzado, auto-residente, auto-compilável

C^{LF} LO
LD

- Compilador cruzado roda em uma máquina e produz código para outra.
- Compilador auto-residente (ou nativo) é executado na mesma máquina para qual gerou código objeto.
- *Em: [http://en.wikipedia.org/wiki/Bootstrapping_\(compilers\)](http://en.wikipedia.org/wiki/Bootstrapping_(compilers)):*
 - *A large proportion of programming languages are bootstrapped, including [BASIC](#), [C](#), [Pascal](#), [Factor](#), [Haskell](#), [Modula-2](#), [Oberon](#), [OCaml](#), [Common Lisp](#), [Scheme](#), [Python](#) and more.*
- Compilador auto-compilável. Porque alguém faria um compilador para uma linguagem sendo implementado nela própria????
 - Lisp foi a primeira linguagem a ter um em 1962

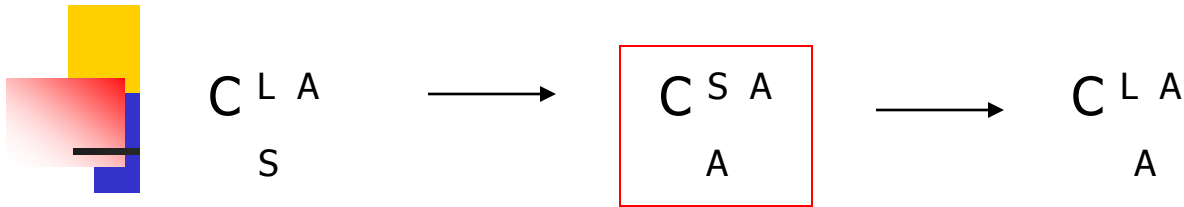


Auto-compilável

- Há vantagens? Facilita o transporte de uma máquina para outra. Suponha que queremos uma linguagem L para as máquinas A e B .

- 1o. Passo: $C \begin{matrix} S & A \\ & A \end{matrix}$ Pode ser escrito numa linguagem aceita por A , assembly por exemplo

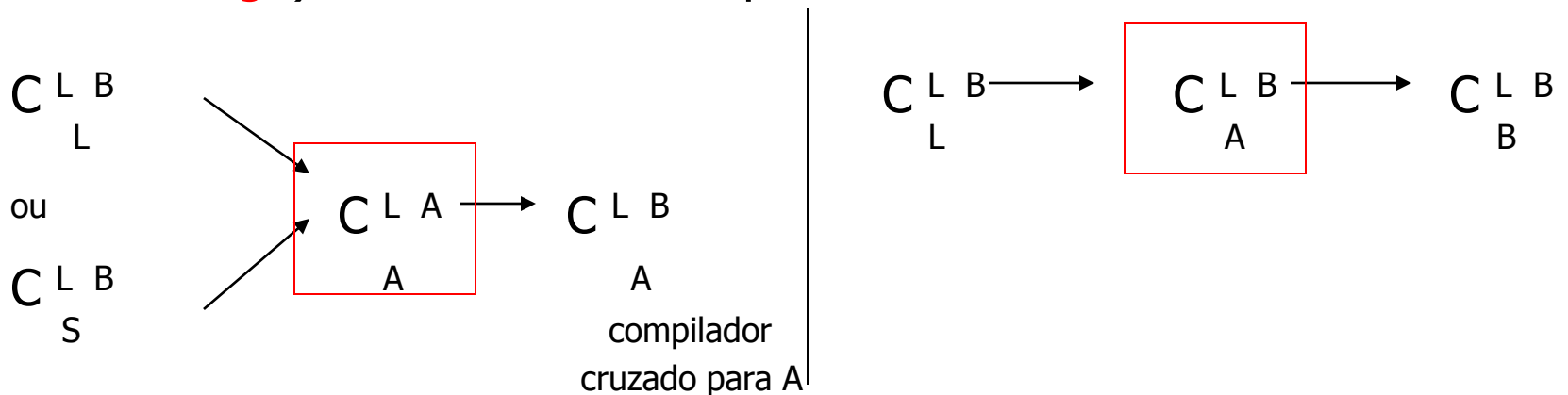
- 2o. Passo: $C^L A$
S



Agora suponha que desejamos ter $C^L B$
B

Como temos $C^L A$ fica mais fácil convertê-lo em $C^L B$
S L

do que escrever um outro compilador (só temos que mudar a geração de código). Processo em dois passos:





Compiler Compilers

- Ferramentas para auxiliar o desenvolvimento de compiladores:
 - geradores de analisadores léxicos (LEX, por exemplo)
 - geradores de analisadores sintáticos (YACC, por exemplo, possui também chamadas de rotinas para as ações semânticas e geração de código)
 - Geradores de código (recebem uma representação intermediária - abstract syntax trees to graph-based intermediate representations - e uma descrição da máquina alvo)
 - Um exemplo é o BEG (back end generator - <http://www.hei.biz/beg/>)
 - sistemas complexos que produzem o compilador inteiro (toolkit)
 - [Vejam outros em: http://catalog.compilertools.net/](http://catalog.compilertools.net/)
 - <http://www.thefreecountry.com/programming/compilerconstruction.shtml>





Ambiente de Execução


Infra-estrutura com que o programa objeto conta para sua correta execução:

- uma interface com o SO
- um conjunto de rotinas que deixam disponíveis os recursos oferecidos pela linguagem (rotinas aritméticas, trigonométricas, pacotes gráficos, etc.)
- um conjunto de procedimentos de manutenção do ambiente: gerenciamento de memória, da passagem de parâmetros, do controle da recursividade, etc.





Pascal P-Compiler

- Escrito por um grupo encabeçado por N. Wirth.
- Este compilador gera código chamado P-Code para uma máquina virtual à pilha 
- Problema: a velocidade era 4 vezes menor que o código compilado, mas podemos traduzir o P-Code para um código de máquina real.
- Este método difundiu o Pascal (Pascal UCSD) e o tornou disponível para uma grande variedade de máquinas.
 - <http://en.wikipedia.org/wiki/P-Code>





Stack Machine

X

Register Machine

- No passado máquinas a pilha reservavam uma parte da memória principal para avaliar expressões. Esta implementação era mais lenta que o uso de registradores pois requer muitas referências à memória principal.
- Máquinas como Lilith, Burroughs e HP usam um pequeno array de registradores rápidos para implementar a pilha de expressões.
- Uma máquina a pilha pode ser mais rápida do que máquinas baseadas em registradores, pois necessitam de poucas instruções.
- Atende bem linguagens com procedimentos recursivos, pois seu uso está intimamente relacionado ao uso de pilhas.
- Máquinas a pilha suportam efetivamente linguagens de alto nível estruturadas por blocos (proc. e funções).



Exemplo de Código Binário Relocável

Duas classes de instruções precisam de mais informações:
 referências a endereços de variáveis e funções externas.
 referências a endereços de memória, por exemplo:

b := a + 2

linguagem de alto nível

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

compilador gera código assembly

```
0001 01 00 00000000 *
0011 01 10 00000010
0010 01 00 00000100 *
```

assembler/compilador gera código de máquina relocável

Pode ser carregado a partir de uma posição L de memória

cod. R1 acesso direto
 acesso imediato

* bit de relocação, associado a cada operando no cod. relocável

link editor gera código de máquina absoluto e escolheu L = 15

```
0001 01 00 00001111
0011 01 10 00000010
0010 01 00 00010011
```

end. de a (15)

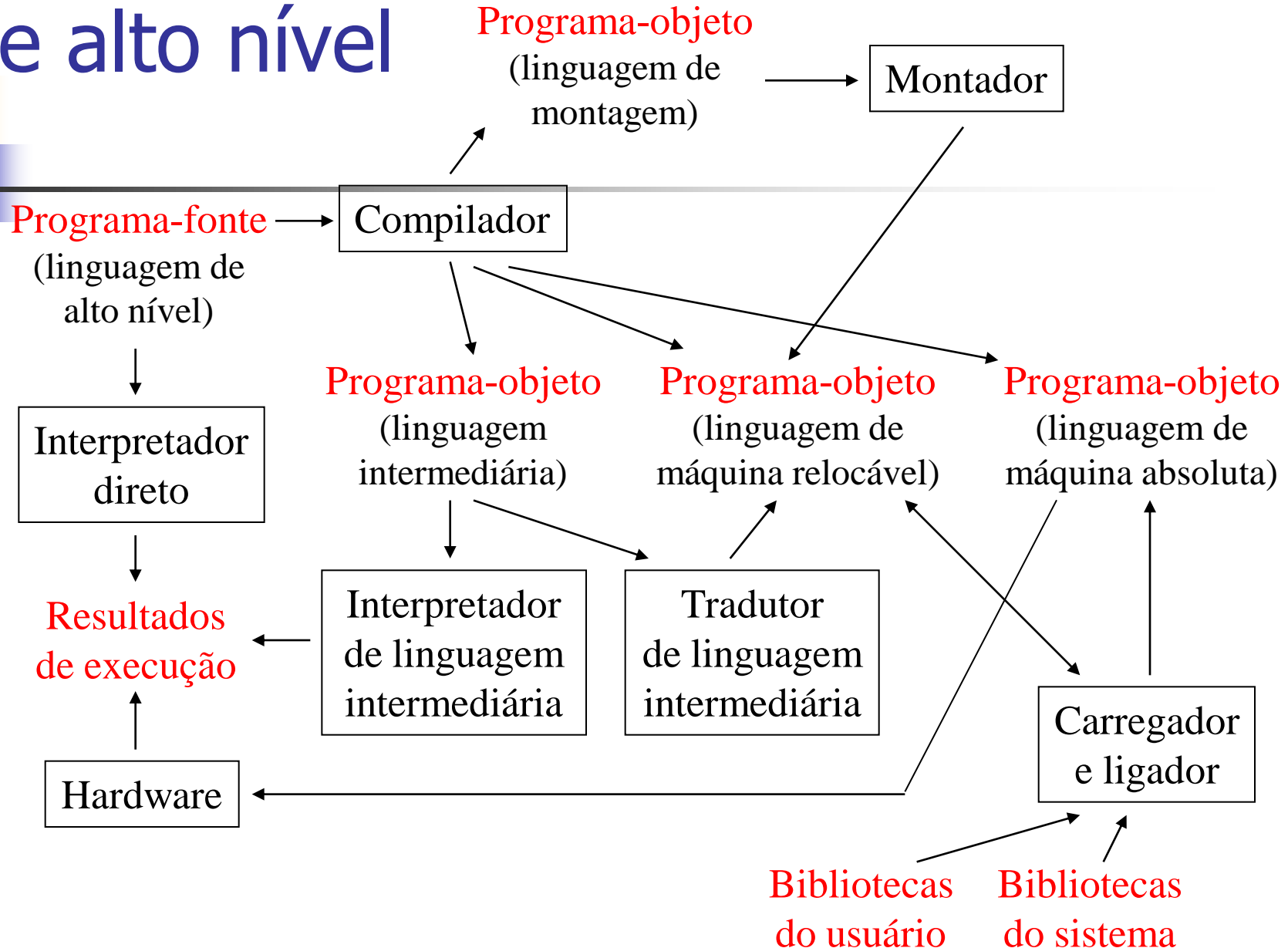
Tabela de símbolos do montador

a	0
b	4

end. de b (19)

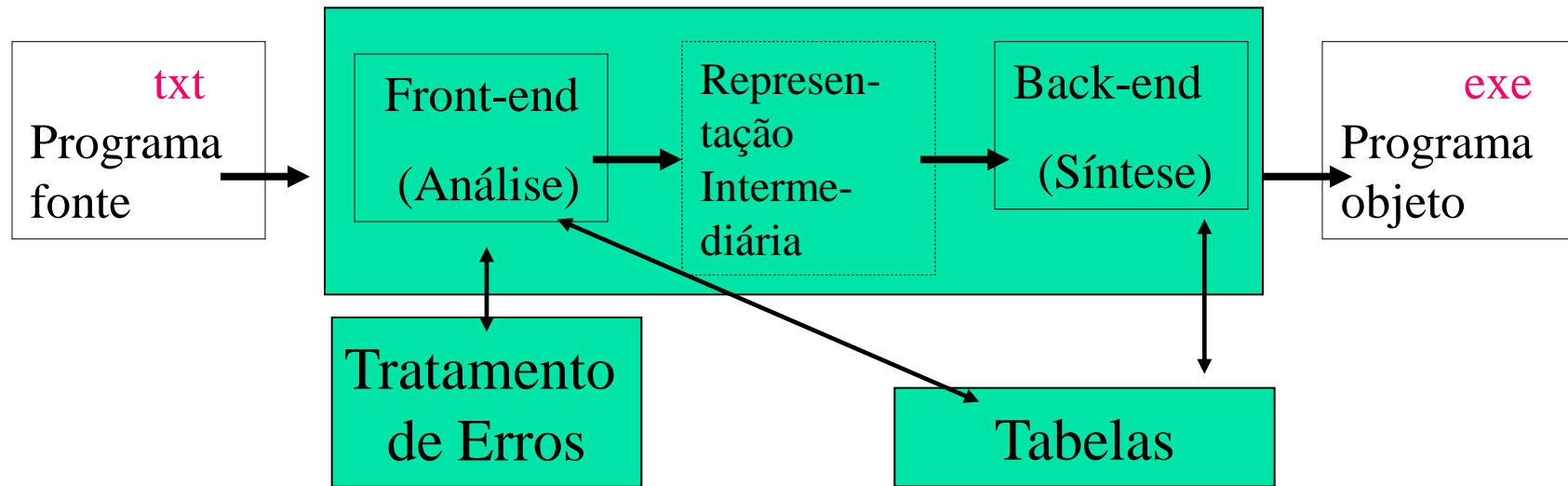


Execução de linguagens de alto nível



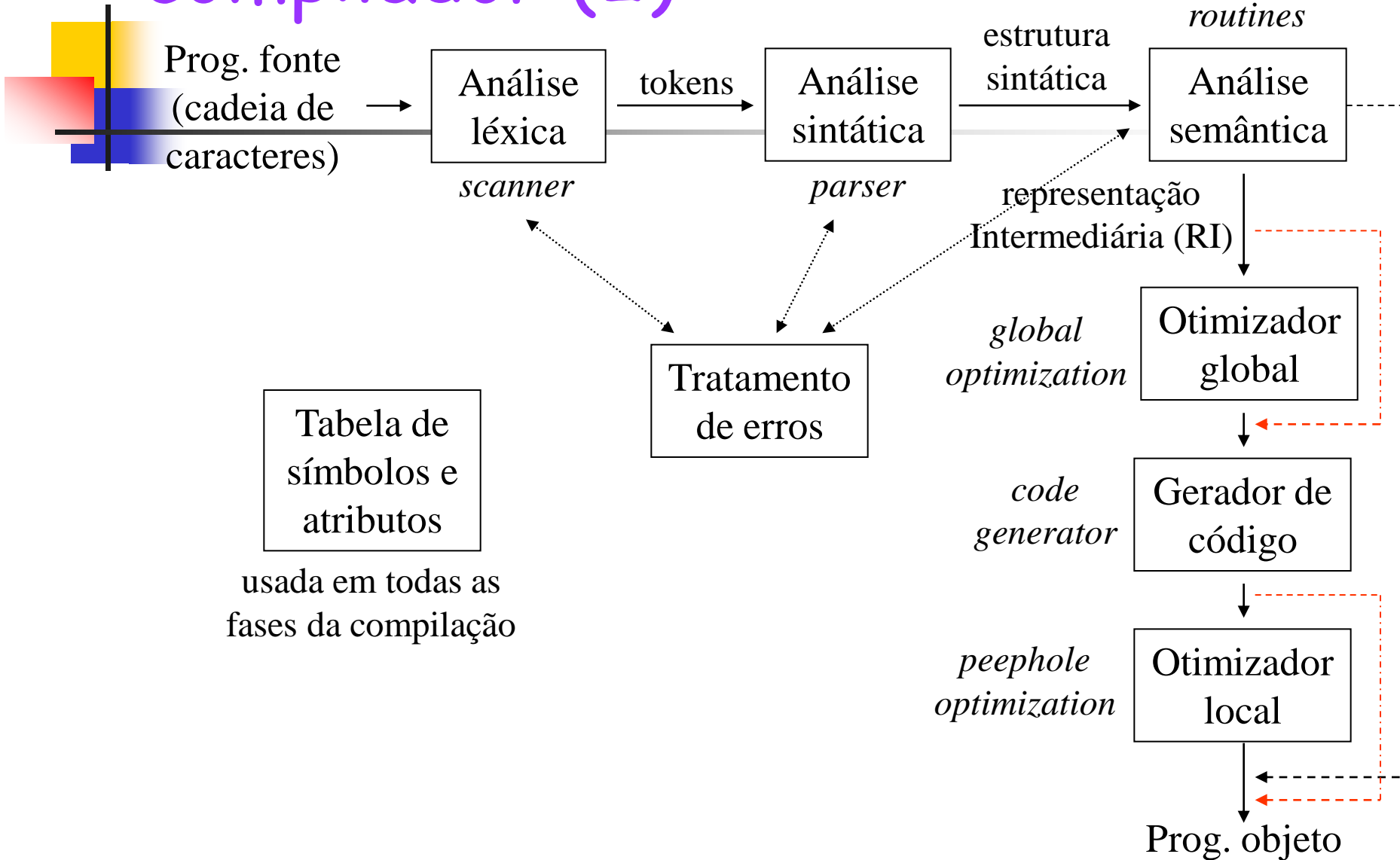
estrutura conceitual de um compilador (1)

Fases da Análise: A Léxica, A Sintática, A Semântica

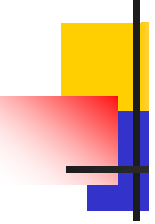


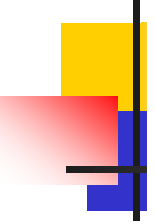
Fases da Síntese: Otimização Global, Geração de Código, Otimização Local

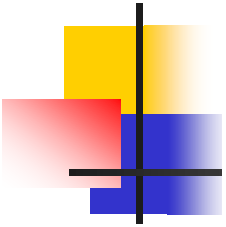
estrutura conceitual de um compilador (2)



- **Análise Léxica:** responsável por ler o arquivo em que está armazenado o programa fonte (da esquerda para a direita) e por reconhecer os *tokens* (ou itens léxicos) e lhes dar um rótulo:
 - Palavras reservadas: *begin, if, var, ... s_begin, s_if, s_var, ...*
 - Identificadores: *X, Y, Z, integer, boolean... id id id id id*
 - Símbolos simples e compostos: *; , := + .. s_; s_, s:= s_+*
 - Constantes:
 - inteiras e reais: *23 23.4 n_int n_real*
 - caracteres e strings: *'a' 'compiladores' caractere cadeia*
 - lógicas: *true, false id id*
- Esses rótulos são usados na formação das regras sintáticas e alguns *tokens* podem ser inseridos na **Tabela de Símbolos**
- Deve também reportar ao usuário a ocorrência de erros léxicos:
 - fim inesperado de arquivo, mal formação de constantes (inteiras, reais, lógicas, literais), caracteres não permitidos no vocabulário terminal da linguagem.

- 
- **Análise Sintática:** responsável por processar os tokens até reconhecer uma regra sintática que, posteriormente será analisada semanticamente, dado que a gramática é livre de contexto.
 - Pode ou não gerar uma estrutura sintática. Em muitos casos só aceita ou o programa.
 - Deve também reportar ao usuário a ocorrência de erros sintáticos: then esperado,) esperado,

- 
- **Análise Semântica** =
 - Análise de contexto +
 - Checagem de tipos +
 - Bookkeeping (gerenciamento da Tabela de Símbolos) +
 - Geração de Código Intermediário (RI)
 - As rotinas semânticas podem gerar alguma RI do programa ou gerar diretamente código objeto.
 - Devem reportar os erros de contexto e de tipos:
 - **variável não declarada, número de parâmetros reais não bate com o número de parâmetros formais, tipo inteiro esperado, incompatibilidade de tipos, etc.**
 - Se uma RI é gerada, então ela serve como entrada ao gerador de código para produzir um programa em linguagem de máquina ou montagem.
 - A RI pode, opcionalmente, ser transformada por um otimizador global para que um código de máquina mais eficiente seja gerado.

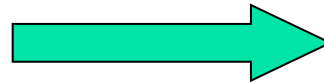


- **O. Global:** melhorias que, em geral, independem da linguagem de máquina. Por exemplo:

- eliminação de cálculos repetidos dentro de malhas trazendo-se para fora do loop.
- Eliminação de sub-expressões iguais:

$A := B + C + D$

$E := B + C + F$



$T1 := B + C$

$A := T1 + D$

$E := T1 + F$

- **Geração de Código:** gera código relocável, absoluto ou de montagem.
- **O. Local:** melhorias no código objeto.



formas de organização de um compilador

- As várias fases de um compilador podem ser executadas em seqüência ou ter sua execução combinada:
- **Compilação em Vários Passos:** A execução de algumas fases terminam antes de iniciar a execução das fases seguintes.
 - Vantagem: possibilidade de otimizações no código
 - Desvantagem: aumento do Tempo de Compilação
- **Compilação de um Passo:** O programa-objeto é produzido à medida que o programa-fonte é processado.
 - Vantagem: eficiência
 - Desvantagem: dificuldade de introdução de otimizações



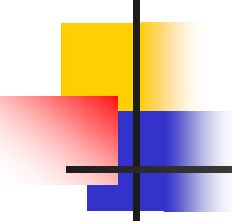
Possibilidades de Organização

- **Um único passo para Análise e Síntese**
 - Todas as fases são intercaladas, nenhuma RI é gerada. Quando as linguagens permitem o comando: goto rótulo

.... rótulo

Alguns compiladores deixam a informação de endereço em branco e só a preenchem quando encontram o endereço. Esta técnica se chama “backpatching” (remendo)

- **Único passo + “Peephole Optimization” (local)**
 - O otimizador toma o código de máquina gerado e olhando somente umas poucas instruções por vez melhora tal código.

- 
-
- Um único passo para Análise e Síntese da RI mais um passo para a Geração de Código
 - O front-end é independente de máquina. Facilidade para portar
 - Vários passos para a Análise
 - Vários passos para a Síntese
 - Nestas duas opções algumas fases terminam antes de outras.
 - Existe o aumento do tempo de compilação, pois os dados podem ser guardados em arquivos.
 - Como vantagem temos a possibilidade de gerar verdadeiras otimizações no código gerado ou na RI₄₄



Nosso Projeto

- Notação **EBNF** para as regras da GLC vai definir os programas gramaticalmente corretos.
- Uso de uma ferramenta para gerar parser (P2) + analisador léxico (P1): **JavaCC**.
- Semântica Estática (de tempo de Compilação) será fornecida oportunamente (será implementada no P3), com inclusão de código java na especificação do parser.
- Compilador de **um Passo**: todas as fases entrelaçadas; compilador dirigido por sintaxe (programa principal).