

As fases de um compilador



Compilando um programa simples

estrutura de um compilador

formas de organização de um compilador

processo de execução de uma linguagem de alto-nível



Compilando um programa simples

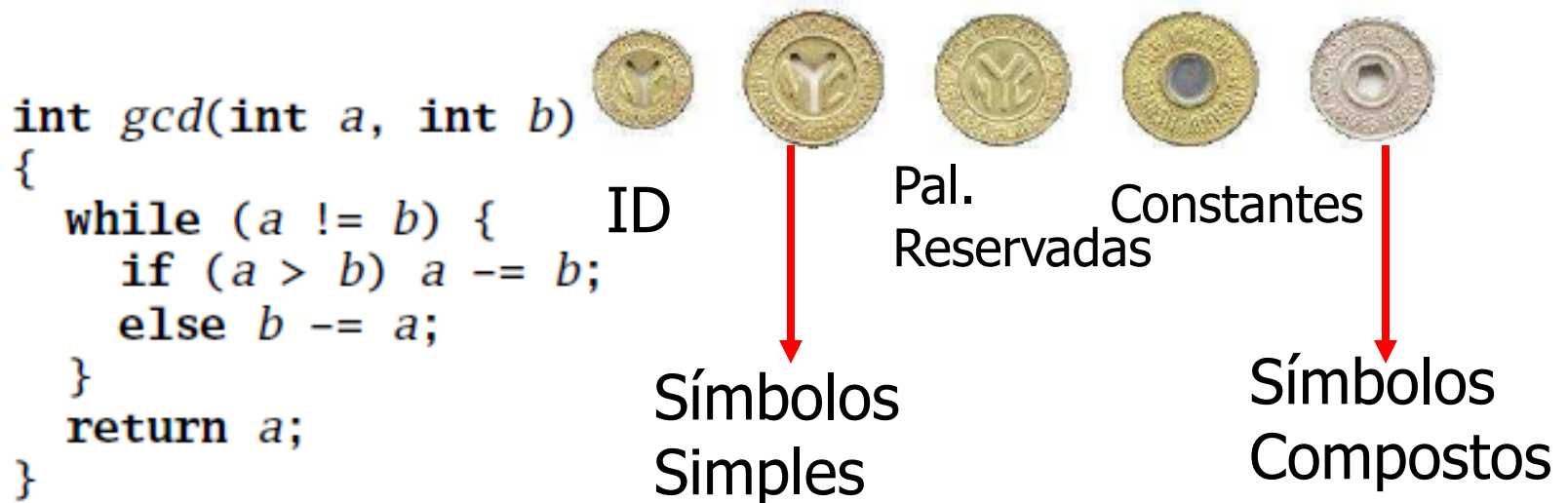
```
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

O que o compilador vê: o texto é uma sequência de caracteres

```
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

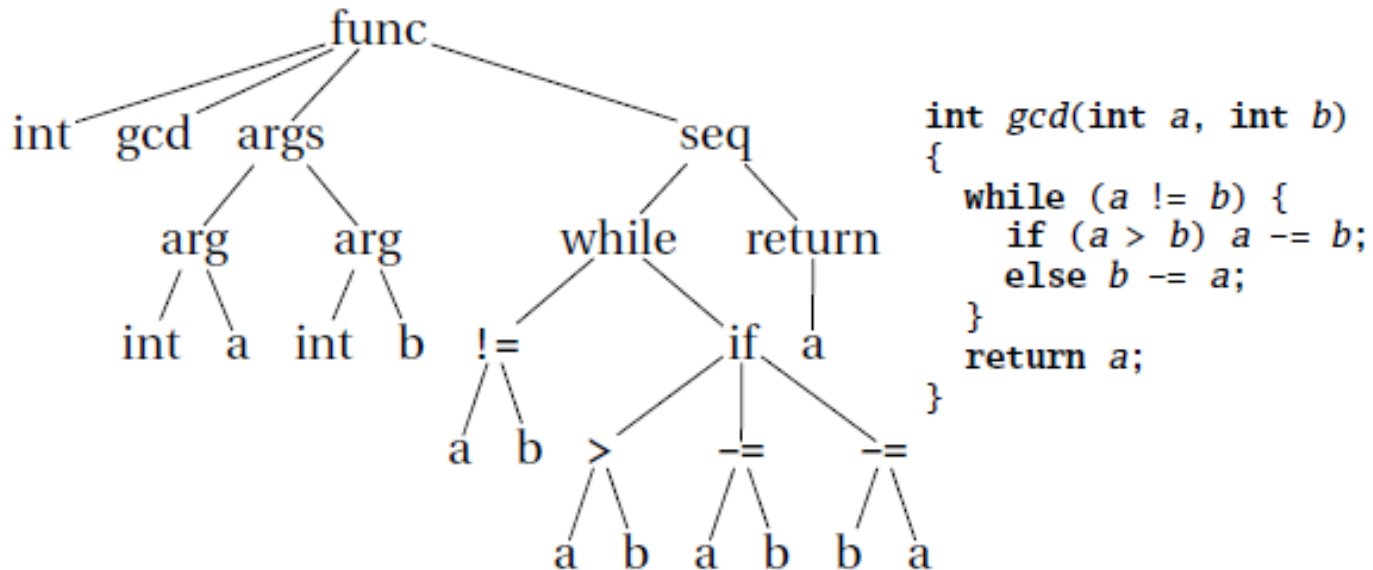
```
i n t s p g c d ( i n t s p a , s p i
n t s p b ) n l { n l s p s p w h i l e s p
( a s p ! = s p b ) s p { n l s p s p s p s p i
f s p ( a s p > s p b ) s p a s p - = s p b
; n l s p s p s p s p e l s e s p b s p - = s p
a ; n l s p s p } n l s p s p r e t u r n s p
a ; n l } n l
```

Análise Léxica fornece tokens: uma cadeia de tokens sem espaços e sem comentários

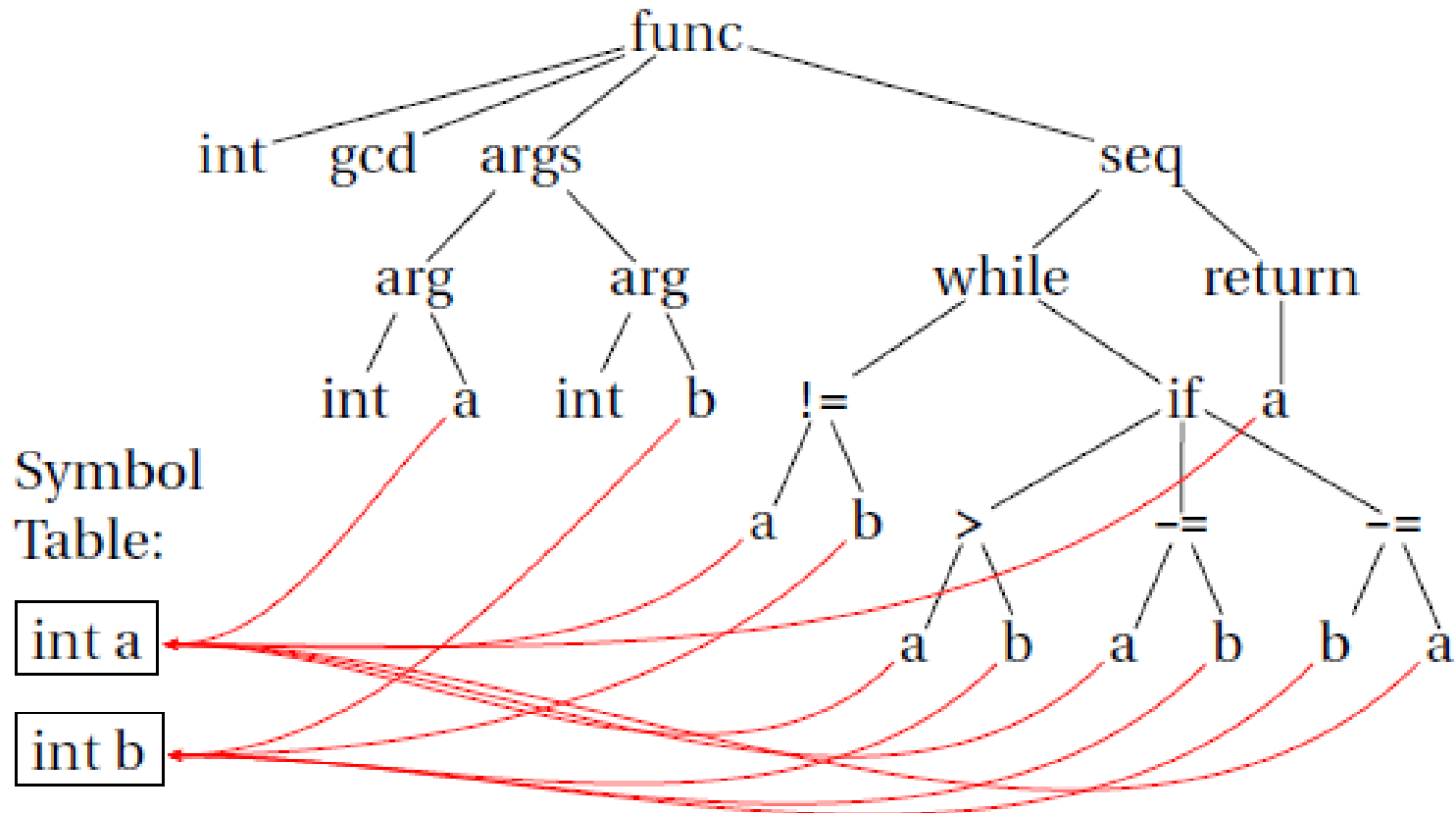


int	gcd	(int	a	,	int	b)	{	while	(a		
!=	b)	{	if	(a	>	b)	a	-=	b	;	else
b	-=	a	;	}	return	a	;	}						

Análise Sintática fornece uma árvore abstrata construída das regras da gramática



Análise Semântica resolve símbolos, com tipos checados



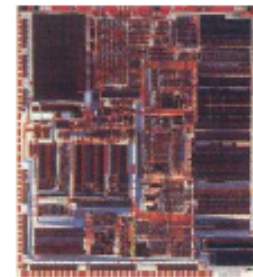
Tradução para uma linguagem intermediária (three-address code: linguagem assembler idealizada com infinitos registradores)

```
L0: sne    $1,  a,  b
      seq   $0, $1, 0
      btrue $0, L1   % while (a != b)
      sl   $3,  b,  a
      seq   $2, $3, 0
      btrue $2, L4   % if (a < b)
      sub  a,   a,  b % a -= b
      jmp  L5
L4: sub  b,   b,  a % b -= a
L5: jmp  L0
L1: ret   a
```

```
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

Geração de código para assembler 80386

```
gcd:  pushl %ebp                % Save FP
      movl %esp,%ebp
      movl 8(%ebp),%eax      % Load a from stack
      movl 12(%ebp),%edx     % Load b from stack
.L8:  cmpl %edx,%eax
      je   .L3               % while (a != b)
      jle .L5               % if (a < b)
      subl %edx,%eax        % a -= b
      jmp .L8
.L5:  subl %eax,%edx        % b -= a
      jmp .L8
.L3:  leave                 % Restore SP, BP
      ret
```





Compilador

- “Um compilador é um programa que transforma um outro programa escrito em uma **linguagem de programação de alto nível** (LP de alto nível) qualquer em instruções que o computador é capaz de entender e executar.”
- Além de definir **LP de alto nível** com relação à geração de linguagens, podemos definir com base nos 4 paradigmas.

O que é uma linguagens de programação



- Uma linguagem de programação é uma linguagem destinada a ser usada por uma **pessoa** para expressar um **processo** através do qual um **computador** pode resolver um **problema**
- Dependendo da perspectiva, têm-se
 - Pessoa = paradigma lógico/declarativo
 - Processo = paradigma funcional
 - Computador = paradigma imperativo
 - Problema = paradigma orientado a objetos



Paradigma lógico/declarativo

- Perspectiva da pessoa
- Um programa lógico é equivalente à descrição do problema expressa de maneira formal, similar à maneira que o ser humano raciocinaria sobre ele
- Exemplo de linguagem: PROLOG



Paradigma funcional

- Perspectiva do processo
- A visão funcional resulta num programa que descreve as operações que devem ser efetuadas (processos) para resolver o problema
- Exemplo de linguagem: LISP

Paradigma

imperativo/procedimental

- Perspectiva do computador
- Baseado na execução seqüencial de comandos e na manipulação de estruturas de dados
- Exemplos de linguagens: FORTRAN, COBOL, ALGOL 60, APL, BASIC, PL/I, ALGOL 68, PASCAL, C, MODULA 2, ADA



Paradigma orientado a objetos

- Perspectiva do problema
- Modelagem das entidades envolvidas como objetos que se comunicam e sofrem operações
- Exemplos de linguagens: SIMULA 67, SMALLTALK
 - C++, C# e Java: linguagens híbridas (paradigmas imperativo e orientado a objetos),

Não são Turing-completas, pois não podem simular uma MT, mas são usadas para preparação de documentos

Linguagens de Markup: HTML, SGML, XML

Linguagens de Scripts ou extensão:
AWK, Perl, PHP, Python, Ruby, LUA, JavaScript

Linguagens de propósito especial:

YACC para criar parsers

- LEX para criar analisadores léxicos
- MATLAB para computação numérica
- SQL para aplicações com BD

O que fazer com o resto das linguagens?

Um pouco de história



- Linguagens que introduziram conceitos importantes e que ainda estão em uso
 - 1955-1965: FORTRAN, COBOL, ALGOL 60, LISP, APL, BASIC (aplicações simples; preocupação com a eficiência)
 - 1965-1971 (com base em ALGOL): PL/I, SIMULA 67, ALGOL 68, PASCAL (pessoas se tornam importantes; preocupação com a inteligibilidade do código, melhores estruturas de controle)
 - Anos 70 e 80: MODULA 2, ADA, C++, Java (mudança de processos para dados; Abstração, herança e polimorfismo)

Assembly Language

Before: numbers

55
89E5
8B4508
8B550C
39D0
740D
39D0
7E08
29D0
39D0
75F6
C9
C3
29C2
EBF6

After: Symbols

```
gcd: pushl %ebp
      movl  %esp, %ebp
      movl  8(%ebp), %eax
      movl  12(%ebp), %edx
      cmpl  %edx, %eax
      je    .L9
.L7:  cmpl  %edx, %eax
      jle  .L5
      subl %edx, %eax
.L2:  cmpl  %edx, %eax
      jne  .L7
.L9:  leave
      ret
.L5:  subl  %eax, %edx
      jmp  .L2
```

FORTRAN

Before

```
gcd: pushl %ebp
      movl  %esp, %ebp
      movl  8(%ebp), %eax
      movl  12(%ebp), %edx
      cmpl  %edx, %eax
      je    .L9
.L7:  cmpl  %edx, %eax
      jle  .L5
      subl %edx, %eax
.L2:  cmpl  %edx, %eax
      jne  .L7
.L9:  leave
      ret
.L5:  subl  %eax, %edx
      jmp  .L2
```

After: Expressions, control-flow

```
10   if (a .EQ. b) goto 20
      if (a .LT. b) then
          a = a - b
      else
          b = b - a
      endif
      goto 10
20   end
```

COBOL

Added type declarations, record types, file manipulation

```
data division.
```

```
file section.
```

```
* describe the input file
```

```
fd employee-file-in
```

```
label records standard
```

```
block contains 5 records
```

```
record contains 31 characters
```

```
data record is employee-record-in.
```

```
01 employee-record-in.
```

```
02 employee-name-in pic x(20).
```

```
02 employee-rate-in pic 9(3)v99.
```

```
02 employee-hours-in pic 9(3)v99.
```

```
02 line-feed-in pic x(1).
```



From cafepress.com

LISP, Scheme, Common LISP

Functional, high-level languages

```
(defun gnome-doc-insert ()
  "Add a documentation header to the current function.
  Only C/C++ function types are properly supported.
  (interactive)
  (let (c-insert-here (point))
    (save-excursion
      (beginning-of-defun)
      (let (c-arglist
            c-funcname
            (c-point (point))
            c-comment-point
            c-isvoid
            c-doinstert)
        (search-backward "(")
        (forward-line -2)
        (while (or (looking-at "^$")
                     (looking-at "^ *}")
                     (looking-at "^ \\*")
                     (looking-at "^#"))
          (forward-line 1))
```

APL

Powerful operators, interactive language, custom character set

```
[0] Z←GAUSSRAND N;B;F;M;P;Q;R
[1] ⍠Returns  $\omega$  random numbers having a Gaussian normal distribution
[2] ⍠ (with mean 0 and variance 1) Uses the Box-Muller method.
[3] ⍠ See Numerical Recipes in C, pg. 289.
[4] ⍠
[5] Z←10
[6] M←1+2*31 ⍠ largest integer
[7] L1:Q←N-ρZ ⍠ how many more we need
[8] →(Q≤0)/L2 ⍠ quit if none
[9] Q←⌈1.3×Q÷2 ⍠ approx num points needed
[10] P←1+(2÷M-1)×1+?(Q,2)ρM ⍠ random points in -1 to 1 square
[11] R←+/P×P ⍠ distance from origin squared
[12] B←(R≠0)∧R<1
[13] R←B/R ⍠ P←B×P ⍠ points within unit circle
[14] F←(1-2×(ρR)÷R)★.5
[15] Z←Z,.,P×F,.[1.5]F
[16] →L1
[17] L2:Z←N+Z
[18] ⍠ ArchDate: 12/16/1997 16:20:23.170
```

Source: Jim Weigang, <http://www.chilton.com/~jimw/gstrand.html>

At right: Datamedia APL Keyboard





Algol, Pascal, Clu, Modula, Ada

Imperative, block-structured language, formal syntax definition, structured programming

```
PROC insert = (INT e, REF TREE t)VOID:
  # NB inserts in t as a side effect #
  IF TREE(t) IS NIL THEN t := HEAP NODE := (e, TREE(NIL), TREE(NIL))
  ELIF e < e OF t THEN insert(e, l OF t)
  ELIF e > e OF t THEN insert(e, r OF t)
  FI;

PROC trav = (INT switch, TREE t, SCANNER continue, alternative)VOID:
  # traverse the root node and right sub-tree of t only. #
  IF t IS NIL THEN continue(switch, alternative)
  ELIF e OF t <= switch THEN
    print(e OF t);
    traverse( switch, r OF t, continue, alternative)
  ELSE # e OF t > switch #
    PROC defer = (INT sw, SCANNER alt)VOID:
      trav(sw, t, continue, alt);
      alternative(e OF t, defer)
    FI;
  FI;
```

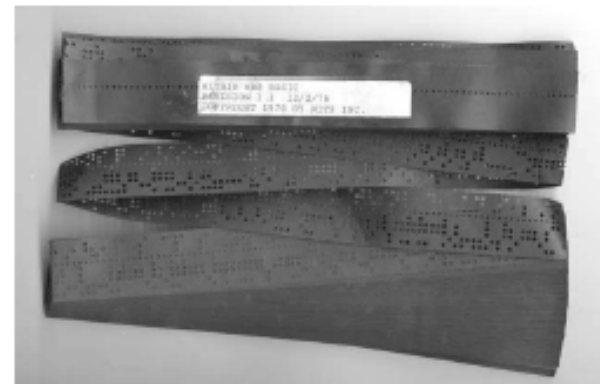
Algol-68, source <http://www.csse.monash.edu.au/~lloyd/tildeProgLang/Algol68/treemerge.a68>

BASIC

Programming for the masses

```
10 PRINT "GUESS A NUMBER BETWEEN ONE AND TEN"  
20 INPUT A$  
30 IF A$ <> "5" THEN GOTO 60  
40 PRINT "GOOD JOB, YOU GUESSED IT"  
50 GOTO 100  
60 PRINT "YOU ARE WRONG. TRY AGAIN"  
70 GOTO 10  
100 END
```

Started the whole Bill Gates/
Microsoft thing. BASIC was
invented by Dartmouth
researchers John George Kemeny
and Thomas Eugene Kurtz.



Simula, Smalltalk, C++, Java, C#

The object-oriented philosophy

```
class Shape(x, y); integer x; integer y;
virtual: procedure draw;
begin
    comment -- get the x & y coordinates --;
    integer procedure getX;
        getX := x;
    integer procedure getY;
        getY := y;

    comment -- set the x & y coordinates --;
    integer procedure setX(newx); integer newx;
        x := newx;
    integer procedure setY(newy); integer newy;
        y := newy;
end Shape;
```




C

Efficiency for systems programming

```
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

ML, Miranda, Haskell

Functional languages with a syntax

```
structure RevStack = struct
  type 'a stack = 'a list
  exception Empty
  val empty = []
  fun isEmpty (s:'a stack):bool =
    (case s
     of [] => true
      | _ => false)
  fun top (s:'a stack): =
    (case s
     of [] => raise Empty
      | x::xs => x)
  fun pop (s:'a stack):'a stack =
    (case s
     of [] => raise Empty
      | x::xs => xs)
  fun push (s:'a stack,x: 'a):'a stack = x::s
  fun rev (s:'a stack):'a stack = rev (s)
end
```



sh, awk, perl, tcl, python, php

Scripting languages: glue for binding the universe together

```
class() {  
  classname='echo "$1" | sed -n '1 s/ *:.*/p' '  
  parent='echo "$1" | sed -n '1 s/^.*/: */p' '  
  hppbody='echo "$1" | sed -n '2,$p' '  
  
  forwarddefs="$forwarddefs  
  class $classname;"  
  
  if (echo $hppbody | grep -q "$classname()"); then  
    defaultconstructor=  
  else  
    defaultconstructor="$classname() {}"  
  fi  
}
```



SQL

Database queries

```
CREATE TABLE shirt (  
    id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,  
    style ENUM('t-shirt', 'polo', 'dress') NOT NULL,  
    color ENUM('red', 'blue', 'white', 'black') NOT NULL,  
    owner SMALLINT UNSIGNED NOT NULL  
        REFERENCES person(id),  
    PRIMARY KEY (id)  
);
```

```
INSERT INTO shirt VALUES  
(NULL, 'polo', 'blue', LAST_INSERT_ID()),  
(NULL, 'dress', 'white', LAST_INSERT_ID()),  
(NULL, 't-shirt', 'blue', LAST_INSERT_ID());
```



Prolog

Logic Language

edge(a, b).

edge(b, c).

edge(c, d).

edge(d, e).

edge(b, e).

edge(d, f).

path(X, X).

path(X, Y) :- edge(X, Z), path(Z, Y).



Sintaxe e semântica

- A descrição de uma linguagem de programação envolve dois aspectos principais
 - Sintaxe: conjunto de regras que determinam quais construções são corretas
 - Semântica: descrição de como as construções da linguagem devem ser interpretadas e executadas
- Em Pascal: $a := b$
 - Sintaxe: comando de atribuição correto
 - Semântica: substituir o valor de a pelo valor de b



Sintaxe

- As gramáticas de linguagens de programação são utilizadas para produzir ou reconhecer cadeias?



Sintaxe

- Descrição de linguagens de programação por meio de gramáticas livres de contexto
- A maioria das linguagens não são livres de contexto, mas sensíveis ao contexto
 - Por exemplo, variável deve ser declarada antes de ser usada
- Métodos para reconhecer gramáticas sensíveis ao contexto são complexos.
- Na prática, especifica-se uma gramática livre de contexto para a linguagem de programação e trata-se a sensibilidade ao contexto de maneira informal
 - Tabela de símbolos



Gramáticas e reconhecedores

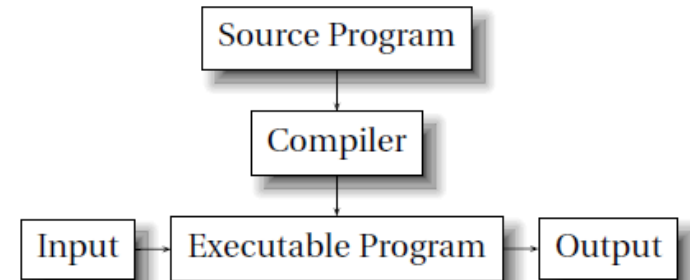
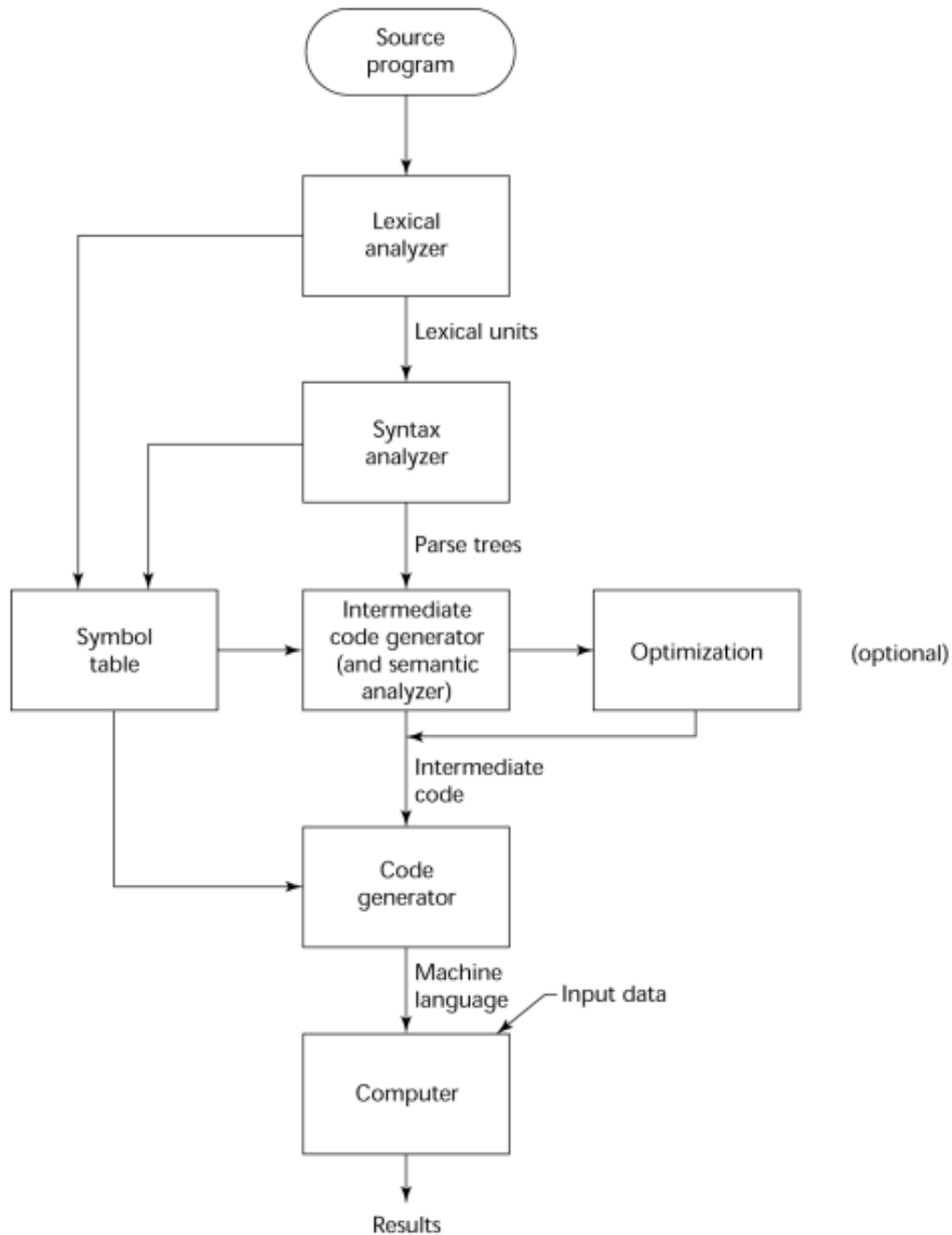
Gramáticas	Reconhecedores
Irrestrita	Máquina de Turing
Sensível ao contexto	Máquina de Turing com memória limitada
Livre de contexto	Autômato a pilha
Regular	Autômato finito



Métodos de Implementação

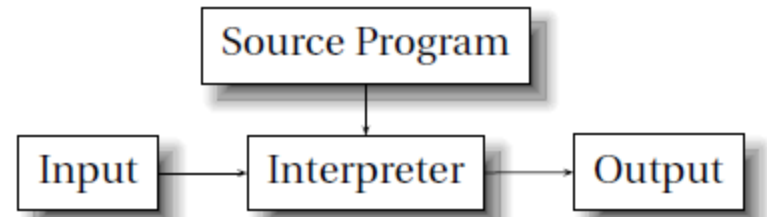
- Compilação
 - Programas são traduzidos em linguagem de máquina
- Interpretação Pura
 - Programas são interpretados por outro programa (interpretador)
- Sistemas Híbridos
 - Oferecem um compromisso entre compiladores e interpretadores puros

Compilador



Tradução lenta
Execução rápida

Interpretador



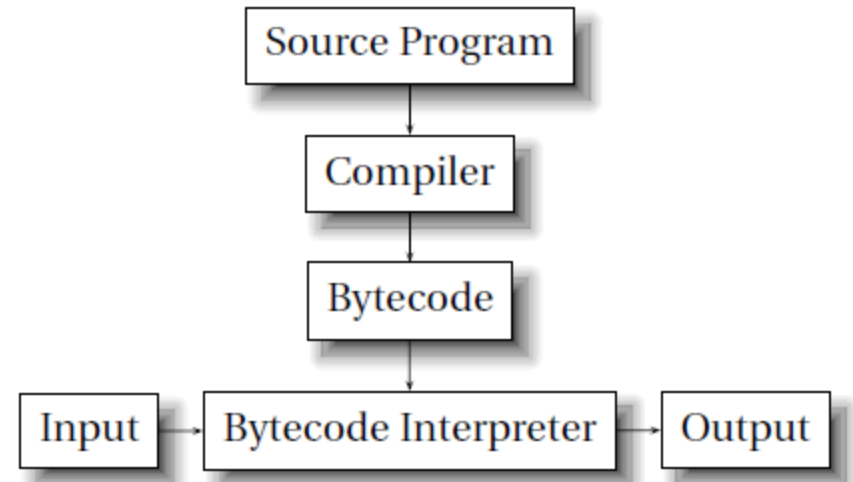
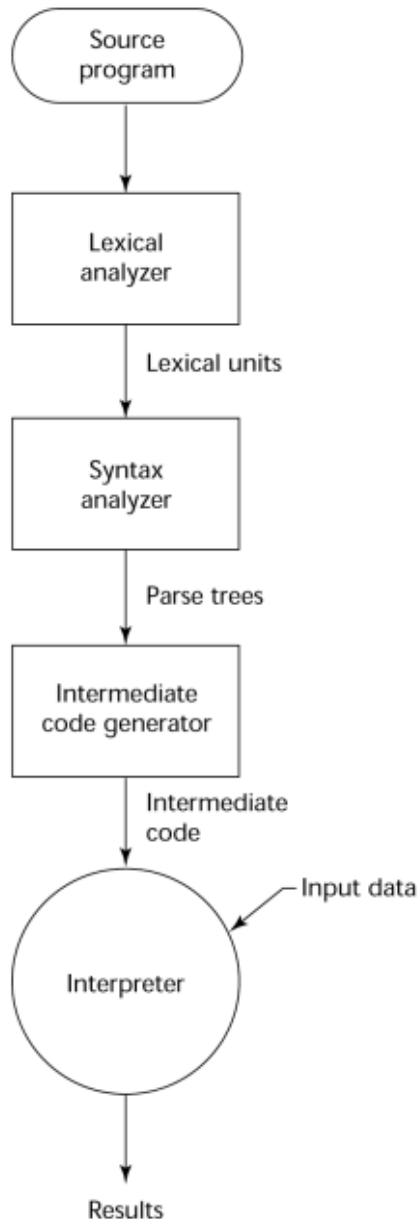
- Não há tradução
- Execução lenta (10 a 100 vezes mais lento que programas compilados)
- Frequentemente requer mais espaço
- Atualmente raro para as linguagens tradicionais, mas está havendo um retorno com as linguagens de script para a Web (por exemplo, JavaScript, PHP)



Sistemas Híbridos

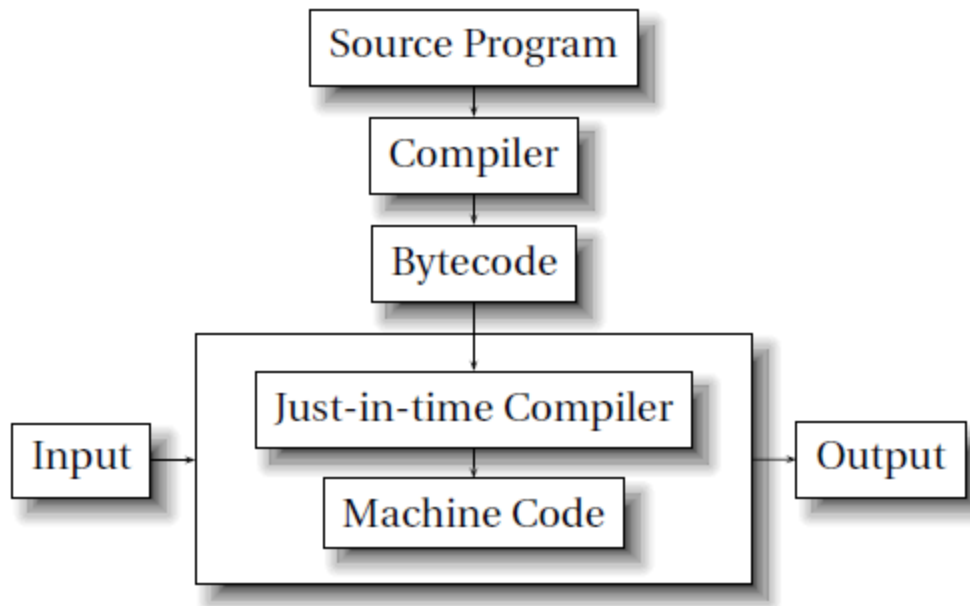
- Há a tradução para uma linguagem intermediária para facilitar a interpretação
- Mais rápido que interpretação pura
- Exemplos:
 - Programas Perl são compilados parcialmente para detectar erros antes da interpretação.
 - Implementações iniciais de Java foram híbridas; o código intermediário, *byte code*, 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



- Traduz inicialmente para uma linguagem intermediária
- Então compila a linguagem intermediária dos subprogramas em código de máquina quando eles são chamados
- Este código é mantido para chamadas subsequentes
- Sistemas JIT são muito usados para Java
- Linguagens .NET são implementadas com um sistema JIT

TIOBE Programming Community Index for February 2010

Position Feb 2010	Position Feb 2009	Delta in Position	Programming Language
1	1	=	Java
2	2	=	C
3	5	↑↑	PHP
4	3	↓	C++
5	4	↓	(Visual) Basic
6	6	=	C#
7	7	=	Python
8	8	=	Perl
9	9	=	Delphi
10	10	=	JavaScript
11	11	=	Ruby
12	32	↑↑↑↑↑↑↑↑	Objective-C
13	-	↑↑↑↑↑↑↑↑	Go
14	14	=	SAS
15	13	↓↓	PL/SQL
16	17	↑	ABAP
17	16	↓	Pascal
18	18	=	ActionScript
19	23	↑↑↑↑	Lisp/Scheme
20	24	↑↑↑↑	MATLAB

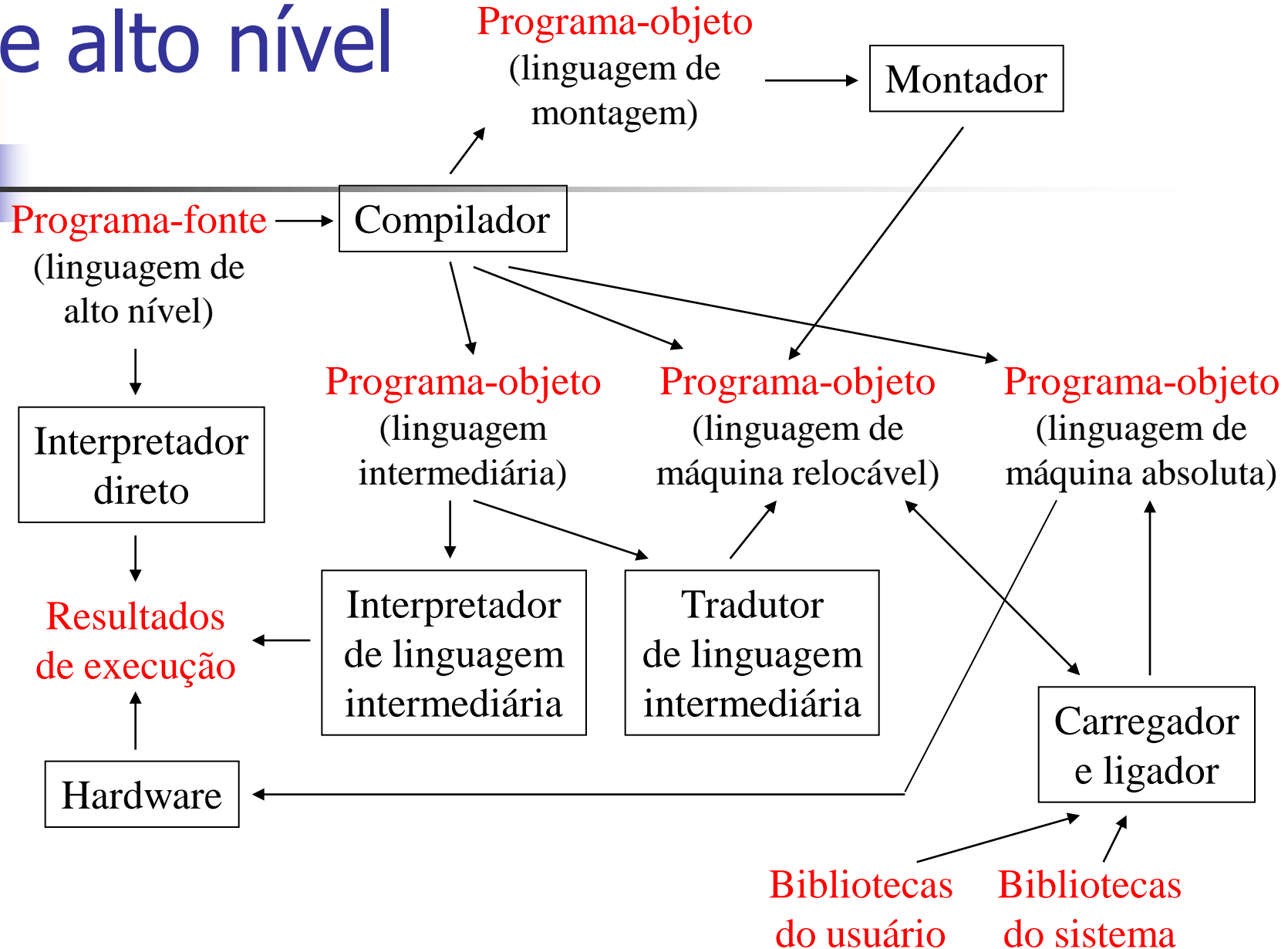
Atualizado mensalmente. Usa máquinas de busca (Google, MSN, Yahoo!, Wikipedia e YouTube) para calcular o número mundial de usuários, cursos e vendedores de linguagens pagas.

Não se refere à melhor linguagem nem àquela para qual muitas linhas foram escritas.



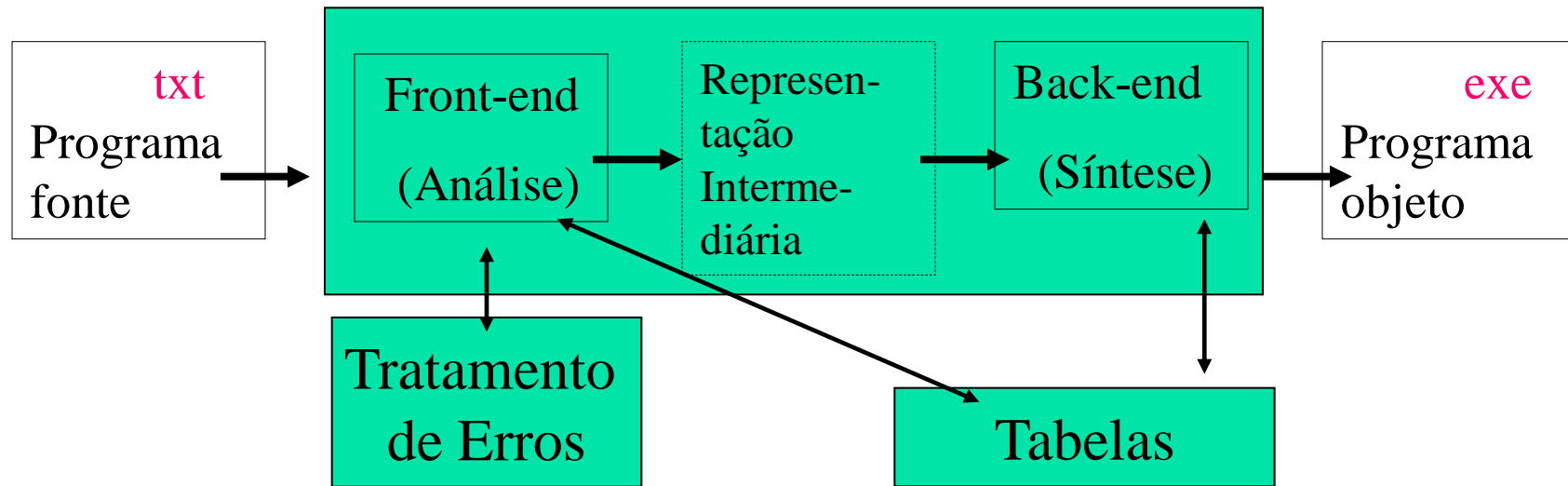
<http://golang.org/>

Execução de linguagens de alto nível



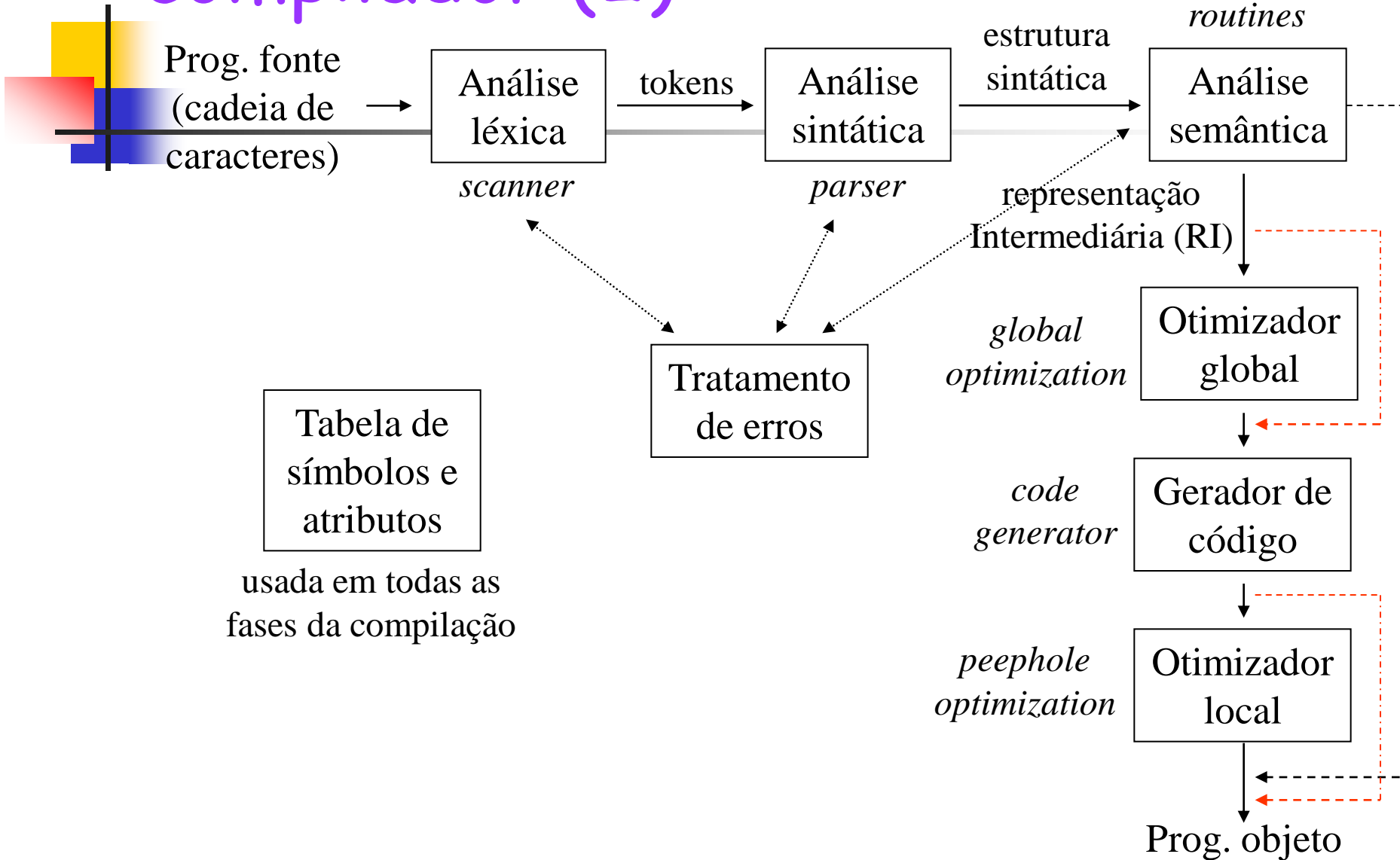
estrutura conceitual de um compilador (1)

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



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

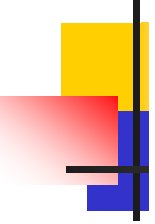
estrutura conceitual de um compilador (2)

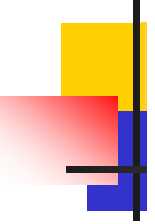


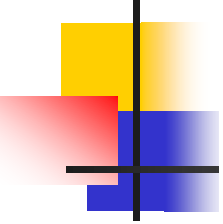


As fases e os erros que elas reportam

- **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, ...**

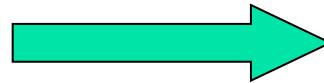
- 
- **Análise Semântica** =
 - Análise de contexto +
 - Checagem de tipos +
 - Bookkeeping (gerenciamento da Tabela de Símbolos) +
 - Geração de Código Intermediário (RI)
 - As rotinas semânticas podem gerar alguma RI do programa ou gerar diretamente código objeto.
 - Devem reportar os erros de contexto 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



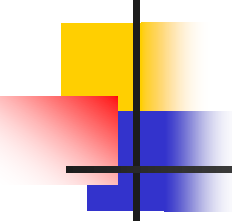
Possibilidades de Organização

- 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 RI₅₁



Nosso Projeto

- Notação **EBNF** para as regras da GLC vai definir os programas gramaticalmente corretos.
- Uso de uma ferramenta para gerar parser + analisador léxico: **JavaCC**.
- Semântica Estática (de tempo de Compilação) será fornecida oportunamente: realizada com inserção de código Java no arquivo de especificação do parser+lexer
- Não realizaremos a geração de código; só trataremos do Front-End do compilador
- Compilador de **um Passo**: todas as fases entrelaçadas; compilador dirigido por sintaxe (programa principal).