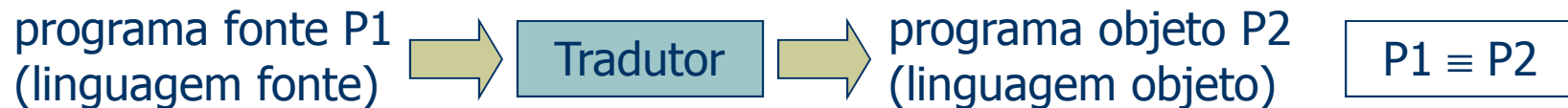


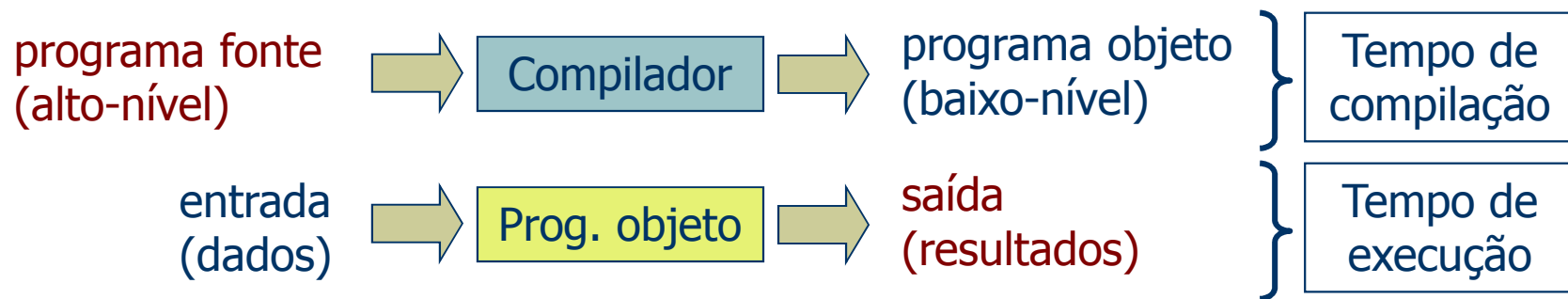
Parte 2 - Construção de Compiladores

2.1 - Conceitos Fundamentais

- Um tradutor é um programa que toma como entrada um programa escrito em uma linguagem de programação (a linguagem fonte) e produz como saída um programa (equivalente) em outra linguagem (a linguagem objeto).



- Se a linguagem fonte é uma linguagem de alto-nível (C, Pascal, Fortran, ...) e a linguagem objeto é uma linguagem de baixo-nível ("assembly" ou linguagem de máquina) então o tradutor é conhecido como compilador. O programa objeto será (eventualmente) executado. Temos então:



Construção de Compiladores

- ♦ Existem outros tradutores que transformam uma linguagem de programação numa linguagem simplificada denominada código intermediário, que pode ser diretamente executado por um programa conhecido como interpretador. Os interpretadores são freqüentemente menores que os compiladores e facilitam a implementação de construções complexas da linguagem. A principal desvantagem dos interpretadores é que o tempo de execução de um programa interpretado é usualmente maior do que o de um programa equivalente compilado. Outros tradutores importantes:
 - Montador: linguagem “assembly” → linguagem de máquina
 - Pré-processador: linguagem de alto nível → linguagem de alto nível

Estrutura de um compilador

- ♦ É comum dividir o processo de compilação em uma série de sub-processos denominados fases. Uma fase é uma operação que toma como entrada uma representação do programa-fonte e produz como saída, outra representação. Numa compilação as seguintes fases podem ser identificadas:
 - análise léxica
 - análise sintática
 - geração de código intermediário
 - otimização de código
 - geração de código objeto

Construção de Compiladores

- ♦ Além dessas fases, existem ainda dois sub-processos importantes:
 - operação com a tabela de símbolos
 - análise e recuperação de erros

Análise léxica

- ♦ Em um programa, certas seqüências de caracteres devem, por conveniência, ser tratadas como um único símbolo. Por exemplo:
 - identificadores
 - constantes
 - palavras-chave (begin, end, repeat, if, ...)
 - um ou mais espaços em branco
 - caracteres duplos (:=, ->, **, ...)
- ♦ O papel do analisador léxico (ou “scanner”) é agrupar certos caracteres terminais em entidades únicas conhecidas como “tokens”. A saída do scanner pode ser uma seqüência de pares da forma:

(tipo do token, informação associada)

onde o primeiro componente é uma categoria sintática (como <identificador> ou <constante>, por exemplo) e o segundo contém informações relativas a esse token (como a cadeia de caracteres que o forma, por exemplo).

Construção de Compiladores

- ◆ Exemplo: $\text{custo} = (\text{preco} + \text{taxa}) * 0.98$

Vamos supor que os identificadores são mapeados em tokens do tipo `<id>` e que constantes são mapeadas em tokens do tipo `<cte>`. Nesse caso teremos:

token	tipo	informação
1	<code><id></code>	"custo"
2	=	
3	(
4	<code><id></code>	"preco"
5	+	
6	<code><id></code>	"taxa"
7)	
8	*	
9	<code><cte></code>	"0.98"

e a saída do scanner poderia ser:

`<id1> = (<id2> + <id3>)*<cte>`

Operações com a tabela de símbolos

Informações sobre alguns tokens são armazenadas em uma ou mais tabelas denominadas tabelas de símbolos. No exemplo acima, será importante (quando da análise semântica ou da geração de código) saber que `<id1>` representa uma variável real "custo" ou que `<cte>` representa uma constante de ponto flutuante "0.98".

Análise sintática

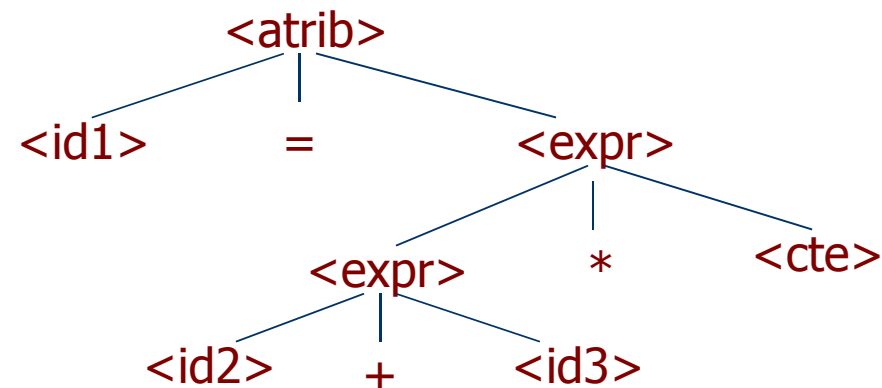
- ◆ A cadeia de tokens produzida pelo analisador léxico forma a entrada para o analisador sintático (ou "parser"), que examina o tipo de cada token para determinar se certas convenções estruturais explícitas na definição sintática da linguagem são obedecidas.

Construção de Compiladores

- ♦ É essencial também (para a geração de código) que o parser produza uma representação adequada da estrutura sintática da cadeia de tokens recebida.
- ♦ Exemplo: a análise sintática de:

$\langle id1 \rangle = (\langle id2 \rangle + \langle id3 \rangle) * \langle cte \rangle$

pode produzir como saída uma estrutura como:



- ♦ O analisador sintático pode também, na saída, fazer uma chamada ao analisador semântico, que irá verificar, por exemplo, se os identificadores envolvidos possuem tipos compatíveis.

Construção de Compiladores

Geração de código

- ♦ A estrutura construída pelo analisador sintático é usada para gerar código. Esse código pode ser linguagem de máquina mas, freqüentemente, é uma linguagem intermediária (como o “assembly”, por exemplo).
- ♦ Existem vários métodos para especificar como o código intermediário deve ser construído. Um método particularmente elegante é a tradução orientada pela sintaxe: associamos a cada nó n da árvore sintática, uma cadeia $C(n)$ de código; o código para um nó qualquer é obtido concatenando-se (numa ordem pré-estabelecida) os códigos associados aos descendentes desse nó (o processo, portanto, é “bottom-up”).
- ♦ Exemplo: Seja um computador com o seguinte conjunto de instruções:

instrução	ação
LOAD m	$ACC \leftarrow c(m)$
ADD m	$ACC \leftarrow c(ACC) + c(m)$
MPY m	$ACC \leftarrow c(ACC) * c(m)$
STORE m	$m \leftarrow c(ACC)$
LOAD=m	$ACC \leftarrow m$
ADD=m	$ACC \leftarrow c(ACC) + m$
MPY=m	$ACC \leftarrow c(ACC) * m$

Construção de Compiladores

- Seja $cmax(n)$ o comprimento máximo de um caminho de n até uma folha.

Algoritmo:

- para todas as folhas (isto é, nós n tais que $cmax(n) = 0$):

$C(n) =$ nome do identificador, se $n = \langle id \rangle$

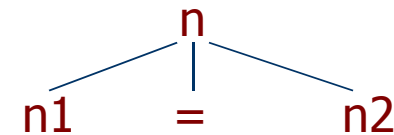
$C(n) =$ = constante, se $n = \langle cte \rangle$

$C(n) =$ Λ , se $n = + \mid *$

- para todos os nós n tais que $cmax(n) = i$ ($i = 1, 2, \dots$)

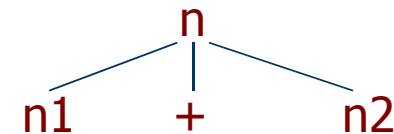
$C(n) =$ LOAD $C(n2)$
STORE $C(n1)$

se



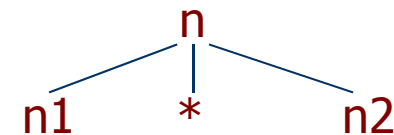
$C(n) =$ $C(n2)$
STORE $Tcmax(n)$
LOAD $C(n1)$
ADD $Tcmax(n)$

se



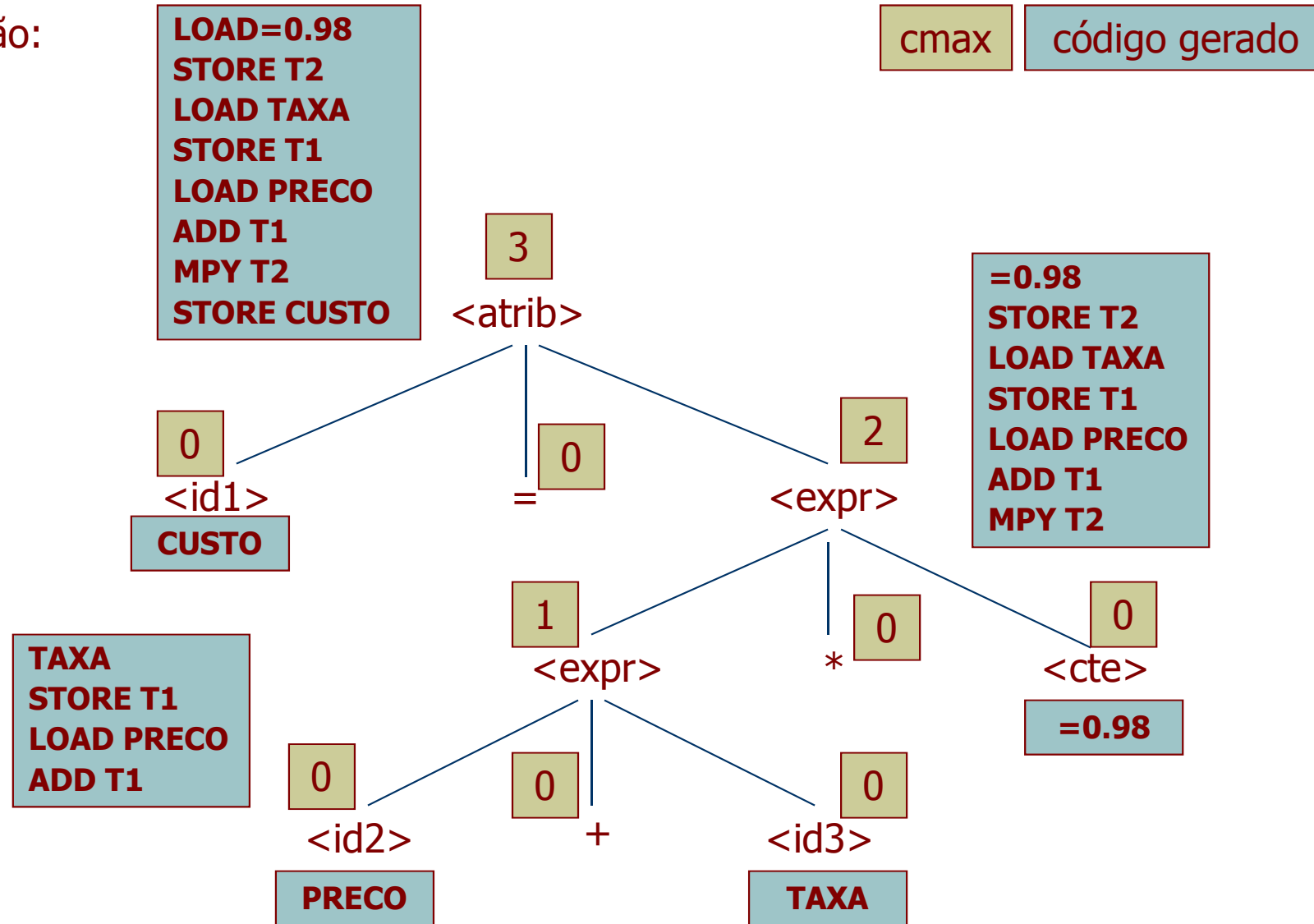
$C(n) =$ $C(n2)$
STORE $Tcmax(n)$
LOAD $C(n1)$
MPY $Tcmax(n)$

se



Construção de Compiladores

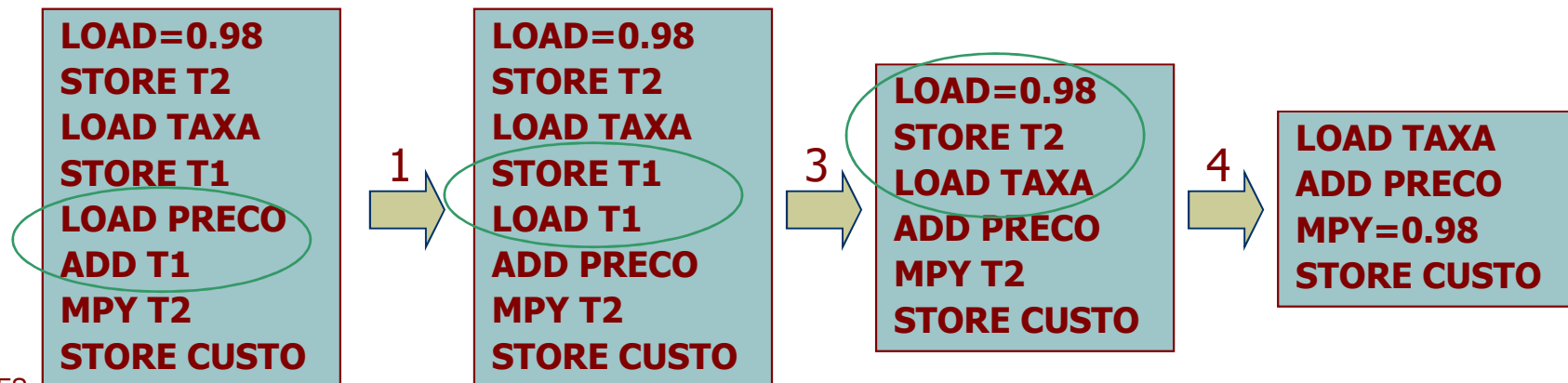
- Então:



Construção de Compiladores

Otimização de código

- ♦ Tentativa de tornar programas objetos mais eficientes, isto é, mais rápidos e/ou mais compactos. Entretanto, não existe um algoritmo para encontrar o menor ou o mais rápido programa equivalente a um outro. Logo, a “otimização” de código é, na maioria das vezes, um “melhoramento” de código.
- ♦ Algumas técnicas:
 1. podemos trocar $\text{LOAD } \alpha; \text{ADD } \beta$ por $\text{LOAD } \beta; \text{ADD } \alpha$ para todo α e β , desde que não haja um desvio para $\text{ADD } \beta$.
 2. analogamente, podemos trocar $\text{LOAD } \alpha; \text{MPY } \beta$ por $\text{LOAD } \beta; \text{MPY } \alpha$
 3. $\text{STORE } \alpha; \text{LOAD } \alpha$ pode ser retirado (desde que α não seja mais usado)
 4. $\text{LOAD } \alpha; \text{STORE } \beta$ pode ser retirado se for seguido de outro LOAD , desde que as ocorrências subsequentes de β sejam trocadas por α até o próximo $\text{STORE } \beta$.
- ♦ Exemplo:



Construção de Compiladores

Análise e recuperação de erros

- ♦ Um compilador não deve se perturbar com a entrada que recebe: ele deve ter uma resposta para qualquer entrada. Para cadeias de entrada que não representam programas válidos, o compilador deve fornecer mensagens apropriadas de diagnóstico.
- ♦ Um compilador tem a oportunidade de detectar erros em três fases: análise léxica, análise sintática e geração de código (uma variável usada sem declaração irá provocar um erro quando da análise semântica ou da geração de código intermediário). Alguns erros são facilmente detectados e (presumivelmente) corrigidos.

Exemplo: $A = B + 2C$ provavelmente deveria ser $A = B + 2 * C$

Entretanto, em geral, é difícil para o compilador olhar para um programa errado e dizer o que ele deveria ser.

- ♦ Na implementação de um compilador, uma ou mais fases são combinadas num único módulo denominado passo. Um passo lê uma representação do programa-fonte, faz as transformações especificadas por suas fases e escreve sua saída num arquivo temporário, que poderá ser lido por passos subseqüentes. O número de passos (e o agrupamento de fases em passos) depende de vários fatores, tais como:
 - a estrutura da linguagem fonte tem um forte efeito sobre o número de passos

Construção de Compiladores

Exemplo: numa linguagem onde é permitida a declaração de uma entidade ocorrer após o seu uso, o código para tais entidades não pode ser gerado até que a declaração tenha sido analisada (ou seja, pelo menos dois passos serão requeridos para a geração de código).

- o meio no qual o compilador deve operar

Exemplo: um compilador multi-passo pode ser feito de forma a usar menos espaço que um compilador de passo-único (que deve caber inteiro na memória). Um compilador multi-passo, no entanto, é mais lento do que um compilador de passo-único (devido à leitura e escrita dos arquivos intermediários).

Resumindo, a estrutura de um compilador é a seguinte:

