Introdução à Compilação

Linguagens Formais A

Prof. Giovani Rubert Librelotto

Índice

- Linguagens
- Evolução das Linguagens de Programação
- Tradutores de Linguagens de Programação
- Estrutura de um Tradutor
- Geradores de Compiladores

Linguagem (1)

- O meio mais eficaz de comunicação entre pessoas é a linguagem (língua ou idioma);
- Na programação de computadores, a linguagem serve como meio de comunicação entre o programador e o computador.
  - Liga o pensamento humano e o processamento de máquina.

Linguagem (2)

- O desenvolvimento de um programa se torna mais fácil se a linguagem de programação em uso estiver próxima ao problema a ser resolvido (linguagem de alto nível).

- Os computadores digitais, por sua vez, aceitam e entendem somente sua própria linguagem de máquina (dita de baixo nível), consistindo tipicamente de seqüências de zeros e uns.

Linguagem (3)

- As linguagens de programação mais utilizadas hoje são de alto nível:
  - linguagens naturais ou no domínio da aplicação em questão.

- Para que se tornem operacionais, os programas escritos em linguagens de alto nível devem ser traduzidos para linguagem de máquina, através de compiladores ou interpretadores:
  - Aceitam (como entrada) uma código expresso em uma linguagem fonte e produzem o mesmo algoritmo expresso em outra linguagem, dita linguagem objeto.

Evolução das Linguagens de Programação

Evolução das Linguagens de Programação

- Cronologicamente, as linguagens de programação podem ser classificadas em cinco gerações:

(1ª) linguagens de máquina

(2ª) linguagens simbólicas (*Assembly*)

} Baixo Nível

(3ª) linguagens orientadas ao usuário

(4ª) linguagens orientadas à aplicação

(5ª) linguagens de conhecimento

} Alto Nível

(1ª) Linguagens de Máquina

- Os primeiros computadores eram programados em linguagem de máquina, em notação binária.
  - um código de operação e um ou dois endereços de registradores ou de memória.

- A programação de um algoritmo complexo usando este tipo de linguagem é complexa, cansativa e fortemente sujeita a erros:
  - zeros e uns, praticamente.

(2ª) Linguagens Simbólicas (*Assembly*)

- Linguagens projetadas para minimizar as dificuldades da programação em notação binária:
  - Códigos de operação e endereços binários substituídos por mnemônicos.

- O processamento de um programa em linguagem simbólica requer tradução para linguagem de máquina, antes de ser executado.

(3ª) Linguagens Orientadas ao Usuário

- As linguagens de 3ª geração ser classificadas em:
  - *linguagens procedimentais:* um programa especifica um procedimento para solucionar um problema. O programa descreve, de forma direta, "como" será resolvido o problema em questão. Exemplos: BASIC, ALGOL, PASCAL, C, etc.
  - *linguagens declarativas:* dividem-se em *funcionais* (LISP) e *lógicas* (PROLOG).

(3ª) Linguagens Orientadas ao Usuário

- As linguagens de 3ª geração foram projetadas para profissionais de processamento de dados e não para usuários finais.

(4ª) Linguagens Orientadas à Aplicação

- Linguagens projetadas para atender a classes específicas de aplicações
- Os programas em linguagens de 4ª geração necessitam menor número de linhas de código do que os programas correspondentes codificados em linguagens de programação convencionais.
- Empregam diversos outros mecanismos, como por exemplo, preenchimento de formulários, interação via vídeo (menus), e auxílio para a construção de gráficos.
- Algumas são, meramente, geradores de relatórios ou pacotes gráficos; outras são capazes de gerar aplicações completas.

(4ª) Linguagens Orientadas à Aplicação

- Os principais objetivos são:
  1. facilitar a programação de computadores;
  2. apressar o processo de desenvolvimento de aplicações;
  3. facilitar e reduzir o custo de manutenção de aplicações;
  4. minimizar problemas de depuração;
  5. gerar código sem erros a partir de requisitos de expressões de alto nível.

(4ª) Linguagens Orientadas à Aplicação

- Exemplos: LOTUS 1-2-3, EXCEL, SQL, SUPERCALC, VISICALC, DATATRIEVE, FRAMEWORK.

(5ª) linguagens de conhecimento

- As linguagens de 5ª geração são usadas principalmente na área de Inteligência Artificial.

- Tais linguagens facilitam a representação do conhecimento que é essencial para a simulação de comportamentos inteligentes.

Tradutores de Linguagens de Programação

Tradutores de Linguagem de Programação

- *Tradutor*, no contexto de linguagens de programação, é um sistema que aceita como entrada um programa escrito em uma linguagem de programação (*linguagem fonte*) e produz como resultado um programa equivalente em outra linguagem (dita *linguagem objeto*).

Tradutores de Linguagem de Programação

- Os tradutores de linguagens de programação podem ser classificados em:
  1. Montadores (*assemblers*)
  2. Macro-assemblers
  3. Compiladores
  4. Pré-compiladores, pré-processadores ou filtros
  5. Interpretadores

Montadores (*assemblers*)

- São os tradutores que mapeiam instruções em *linguagem simbólica* (*assembly*) para instruções de linguagem de máquina:
  - numa relação de uma–para–uma, isto é, uma instrução de linguagem simbólica para uma instrução de máquina.

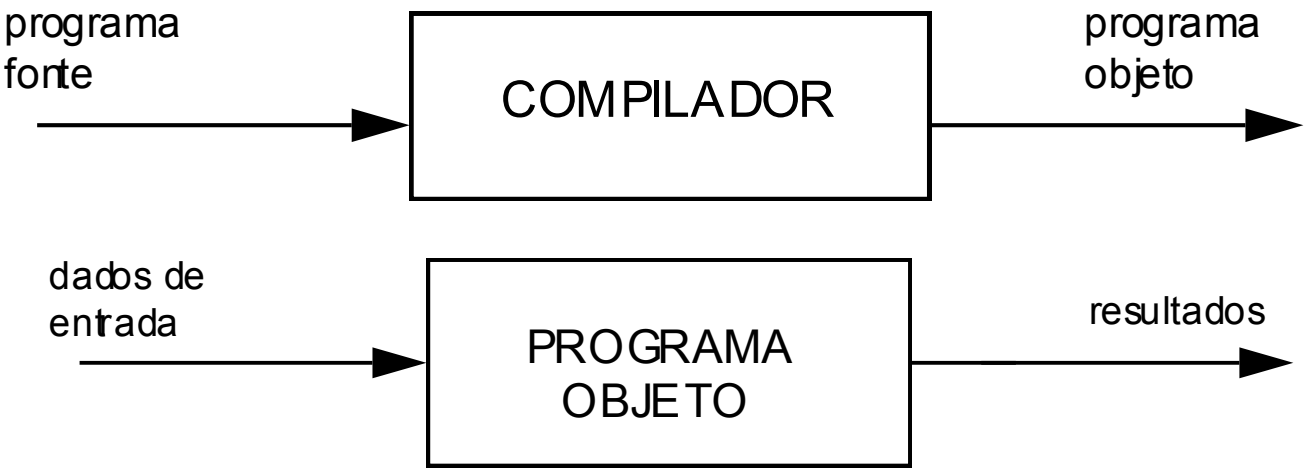
Macro–assemblers

- São tradutores que mapeiam instruções em linguagem simbólica para linguagem de máquina:
  - numa relação de uma–para–várias.

- Um comando macro é traduzido para uma seqüência de comandos simbólicos, antes de ser traduzida para linguagem de máquina:
  - "macros" são facilidades de expansão de texto em linguagem mnemônica.

Compiladores

- São tradutores que mapeiam programas escritos em linguagem de alto nível para programas equivalentes em linguagem simbólica ou linguagem de máquina.



Compiladores

- O espaço de tempo no qual ocorre a conversão de um programa fonte para um programa objeto é chamado de *tempo de compilação*.

- O programa objeto é executado no espaço de tempo chamado *tempo de execução*.

Pré–compiladores Pré–processadores ou Filtros

- São aqueles processadores que mapeiam instruções escritas numa linguagem de alto nível estendida para instruções da linguagem de programação original, ou seja, são tradutores que efetuam conversões entre duas linguagens de alto nível.

Pré–compiladores Pré–processadores ou Filtros

- Os pré–compiladores surgiram para facilitar a extensão de linguagens de alto nível existentes.
  - são usadas para atender aplicações específicas, cujo objetivo é aprimorar o projeto e escrita de algoritmos.

- Por exemplo, existem pré–processadores FORTRAN, BASIC e COBOL estruturados que mapeiam programas em versões estruturadas destas linguagens para programas em FORTRAN, BASIC e COBOL padrões.

Interpretadores

- São processadores que aceitam como entrada o código intermediário de um programa anteriormente traduzido e produzem o "efeito de execução" do algoritmo original sem mapeá-lo para linguagem de máquina.

Interpretadores

- Os interpretadores processam uma forma intermediária do programa fonte e dados ao mesmo tempo.
  - a interpretação fonte ocorre em tempo de execução, não sendo gerado um programa objeto.

Interpretadores x Compiladores

- Alguns interpretadores não utilizam um código intermediário, trabalhando diretamente sobre o programa fonte, analisando um comando fonte cada vez que este deve ser executado.
  - Tal enfoque consome muito tempo e é, raramente, utilizado.

- Uma estratégia mais eficiente envolve a aplicação de técnicas de compilação para traduzir o programa fonte para uma forma intermediária, que é, então, interpretada.

Interpretadores x Compiladores

- Os interpretadores são, geralmente, menores que os compiladores e facilitam a implementação de construções complexas de linguagens de programação.

- A principal desvantagem dos interpretadores é que o tempo de execução de um programa interpretado é maior que o tempo necessário para executar um programa objeto (compilado) equivalente.
  - Isto porque a "execução" do código intermediário tem embutido o custo do processamento de uma tradução virtual para código de máquina cada vez que uma instrução em código intermediário deve ser operada.

Interpretadores x Compiladores

- Uma grande vantagem dos sistemas interpretativos está na implementação de novas linguagens para diferentes equipamentos de computação.

- A utilização do sistema interpretativo permite uma implementação relativamente fácil, uma vez que é produzido um código intermediário padrão, independente de máquina, para o qual é programado um interpretador para cada máquina diferente.

- Tradutores que geram código para máquinas hospedeiras, isto é, máquinas nas quais eles próprios executam, são denominados auto–residentes (*self–resident translators*).

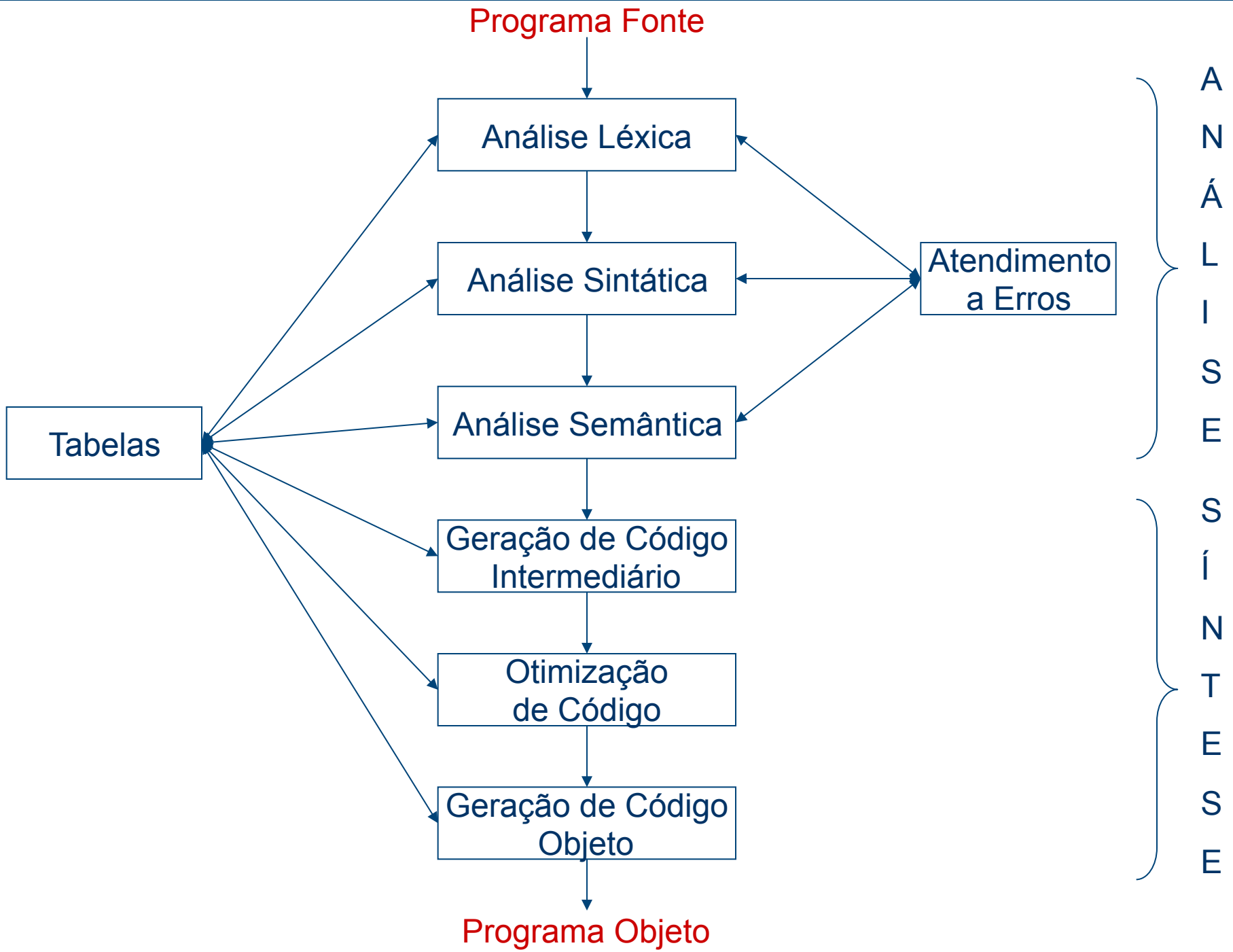
- Aqueles que geram código objeto para outras máquinas, que não as hospedeiras, são denominados compiladores cruzados (*cross–translators*).

Estrutura de um Tradutor

Estrutura de um Tradutor

- O processo de tradução é estruturado em fases onde cada fase se comunica com a seguinte através de uma linguagem intermediária adequada.

- Na prática (implementação), seguidamente, a distinção entre as fases, não é muito clara:
  - as funções básicas para a tradução podem não estar individualizadas em módulos específicos e, possivelmente, se apresentarem distribuídas em módulos distintos.


Programa Fonte
Análise Léxica
Análise Sintática
Análise Semântica
Geração de Código Intermediário
Otimização de Código
Geração de Código Objeto
Programa Objeto
Tabelas
Atendimento a Erros
ANÁLISE
SÍNTESE

Análise Léxica

- O objetivo principal desta fase é identificar seqüências de caracteres que constituem unidades léxicas (*tokens*).
- O analisador léxico lê caractere a caractere o texto fonte verificando se os caracteres lidos pertencem ao alfabeto da linguagem, identificando *tokens*, e desprezando comentários e brancos desnecessários.
- Em geral, o Analisador Léxico funciona sob o comando do Analisador Sintático, como subrotina.

Análise Léxica

- Os *tokens* constituem classes de símbolos
  - palavras reservadas, delimitadores, identificadores, etc.

- Podem ser representados, internamente:
  - pelo próprio símbolo (como no caso dos delimitadores e palavras reservadas);
  - por um par ordenado (a classe do símbolo e o índice para uma área onde o próprio símbolo foi armazenado – Ex: um identificador e sua entrada numa tabela de identificadores).

- A saída do analisador léxico é uma cadeia de *tokens* que é passada para a próxima fase, a Análise Sintática.

Análise Léxica – Exemplo

- Seja, por exemplo, o seguinte texto fonte em Pascal:

  while I < 100 **do** I **:=** J + I ;

- Após a Análise Léxica ter-se-ia a seguinte cadeia de *tokens*:

[whi,] [id, 7] [<,] [cte,13] [do,] [id, 7] [:=,] [id,12] [+,] [id,7] [; ,]

  - Palavras reservadas, operadores e delimitadores são representados pelos próprios símbolos,
  - Variáveis e constantes são representadas por um par [classe do símbolo, índice de tabela].

Análise Sintática

- Tem por função precípua verificar se a estrutura gramatical e o significado das construções declaradas no programa fonte estão de acordo com a especificação sintática da linguagem.

Análise Sintática

- O Analisador Sintático identifica seqüências de símbolos que constituem estruturas sintáticas (por exemplo, expressões, comandos), através de uma varredura ou *parsing* da representação interna (cadeia de *tokens*) do programa fonte.

- O Analisador Sintático produz (explícita ou implicitamente) uma estrutura em árvore, chamada árvore de derivação, que exibe a estrutura sintática do texto fonte, resultante da aplicação das regras gramaticais da linguagem.

- Em geral, a construção da árvore de derivação está implícita nas chamadas das rotinas recursivas que executam a análise sintática.

- Em muitos compiladores, a representação interna do programa resultante da análise sintática é a árvore compactada (sintaxe):
  - visa eliminar redundâncias e elementos supérfluos.

- Esta estrutura objetiva facilitar a geração do código que é a fase seguinte à análise.

- Outra função dos reconhecedores sintáticos é a detecção de erros de sintaxe identificando clara e objetivamente a posição e o tipo de erro ocorrido.

- Mesmo que erros tenham sido encontrados, o Analisador Sintático deve tentar recuperá-los prosseguindo a análise do texto restante

Exemplo – Análise Sintática

- Por exemplo, em Pascal, o comando while tem a seguinte sintaxe:

  while <expressão> **do** <comando> ;

- A estrutura <expressão> deve apresentar-se sintaticamente correta e sua avaliação deve retornar um valor do tipo lógico:
  - Isto é, a aplicação dos operadores (relacionais/lógicos) sobre os operandos (constantes/variáveis) deve resultar num valor do tipo lógico (verdadeiro/falso).

Exemplo – Análise Sintática

- As regras gramaticais que definem as construções da linguagem podem ser descritas através de produções (regras que produzem, geram), cujos elementos incluem *símbolos terminais* (aqueles que fazem parte do código fonte) e símbolos *não–terminais* (aqueles que geram outras regras).

- No exemplo que segue, os terminais aparecem em letras minúsculas e os símbolos não–terminais aparecem delimitados por “<” e “>”.

Exemplo – Gramática

- Os comandos while e de atribuição podem ser definidos (parcialmente) pelas seguintes produções:

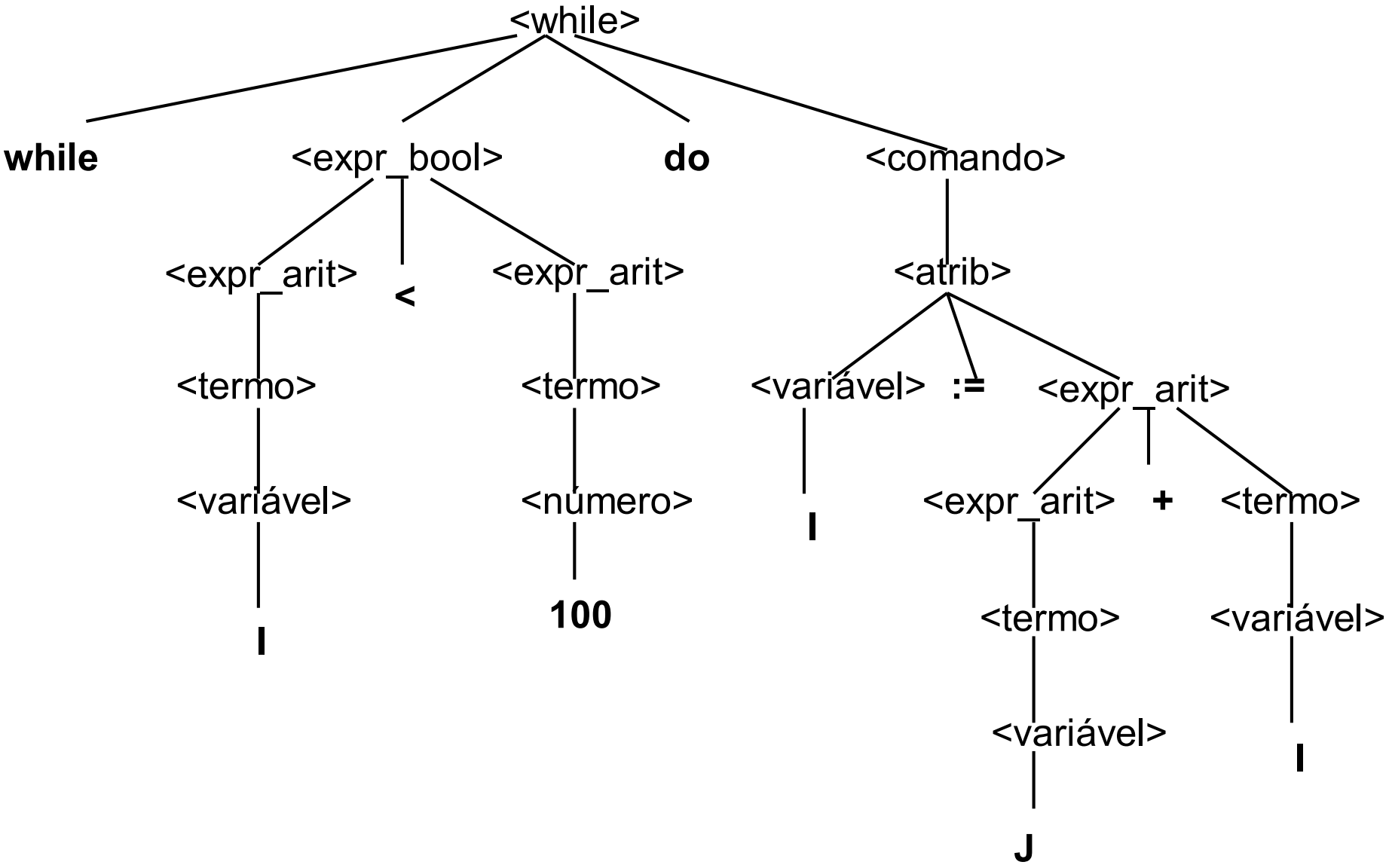
<comando>	→	<while> | <atrib> | ...
<while>	→	**while** <expr_bool> **do** <comando>
<atrib>	→	<variável> := <expr_arit> ;
<expr_bool>	→	<expr_arit> < <expr_arit> | ...
<expr_arit>	→	<expr_arit> + <termo> | <termo> | ...
<termo>	→	<número> | <variável>

Exemplo – Gramática

- Considere o comando while exemplificado anteriormente:

  while I < 100 **do** I := J + I ;

- A partir da seqüência de *tokens* liberada pelo Analisador Léxico, o Analisador Sintático produziria a árvore de derivação mostrada a seguir.


<while>
while
<expr_bool>
do
<comando>
<expr_arit>
<
<expr_arit>
<atrib>
<termo>
<termo>
<variável>
:=
<expr_arit>
<variável>
<número>
<expr_arit>
+
<termo>
I
I
100
<termo>
<variável>
<variável>
I
J

Analisador Semântico

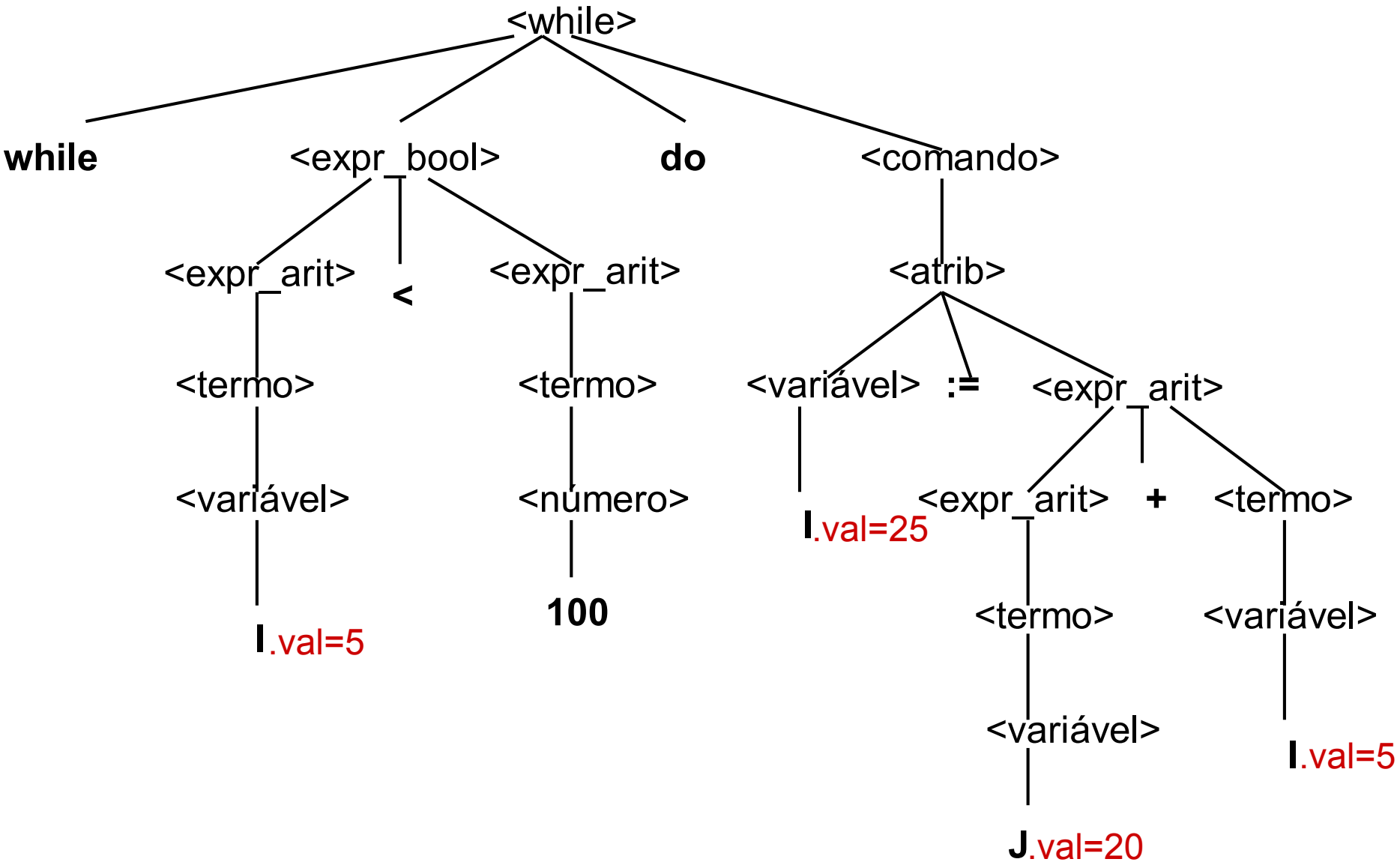
- Muitas vezes, o Analisador Sintático opera conjuntamente com o Analisador Semântico, cuja principal atividade é determinar se as estruturas sintáticas analisadas fazem sentido, ou seja, se um identificador declarado como variável é usado como tal; se existe compatibilidade entre operandos e operadores em expressões; etc

Exemplo

- Considere o comando while anterior:

  while I < 100 **do** I := J + I ;

- Para a total correção do código fonte, é necessário uma verificação semântica de toda a estrutura sintática.


<while>
while
<expr_bool>
do
<comando>
<expr_arit>
<
<expr_arit>
<atrib>
<termo>
<termo>
<variável>
:=
<expr_arit>
<variável>
<número>
<expr_arit>
+
<termo>
I.val=5
100
I.val=25
<termo>
<variável>
<variável>
I.val=5
J.val=20

- As fases até aqui descritas constituem módulos que executam tarefas analíticas.

- As fases seguintes trabalham para construir o código objeto:
  - geração de código intermediário,
  - otimização de código,
  - geração de código objeto.

Geração de Código Intermediário

- Esta fase utiliza a representação interna produzida pelo Analisador Sintático e gera como saída uma seqüência de código.

- Este código pode, eventualmente, ser o código objeto final mas, na maioria das vezes, constitui-se num código intermediário, pois a tradução de código fonte para objeto em mais de um passo.

Geração de Código Intermediário

- Vantagens:
  - possibilita a otimização de código intermediário, de modo a se obter o código objeto final mais eficiente;
  - resolve, gradualmente, as dificuldades da passagem de código fonte para objeto (alto–nível para baixo–nível), já que o código fonte pode ser visto como um texto condensado que "explode" em inúmeras instruções elementares de baixo nível.

Geração de Código Intermediário

- A geração de código intermediário pode estar estruturalmente distribuída nas fases anteriores (análise sintática e semântica) ou mesmo não existir (tradução direta para código objeto), no caso de tradutores bem simples.

- A grande diferença entre o código intermediário e o código objeto final é que o intermediário não especifica detalhes, tais como, que registradores serão usados, que endereços de memória serão referenciados, etc.

Geração de Código Intermediário

- Por exemplo, para o comando while anteriormente apresentado, o gerador de código intermediário, recebendo a árvore de derivação, mostrada acima, poderia produzir a seguinte seqüência de instruções:

```
L0      if  I  <  100  goto   L1
        goto   L2
L1      TEMP  :=  J  +  I
        I  :=  TEMP
        goto L0
L2      . . .
```

Geração de Código Intermediário

- Há vários tipos de código intermediário:
  - quádruplas,
  - triplas,
  - notação polonesa pós–fixada,
  - etc.

- A linguagem intermediária do exemplo acima é chamada código de três endereços, pois cada instrução tem no máximo três operandos.

Otimização de Código

- Esta fase tem por objetivo otimizar o código intermediário em termos de velocidade e espaço de memória.

- Considerando o código intermediário anterior, o seguinte código otimizado poderia ser obtido:

```
L0      if  I  ≥  100  goto   L2
        I  :=  J  +  I
        goto   L0
L2      . . .
```

Geração de Código Objeto

- É a fase mais difícil, pois requer uma seleção cuidadosa das instruções e registradores da máquina alvo a fim de produzir código objeto eficiente.

- Existem tradutores que ainda possuem mais uma fase, fase esta que realiza otimização do código objeto, isto é, otimização de código dependente de máquina.

Geração de Código Objeto

- Esta fase tem como objetivos:
  - produção de código objeto,
  - reserva de memória para dados e variáveis,
  - geração de código para acessar tais posições,
  - seleção de registradores,
  - etc.

Exemplo - Geração de Código Objeto

- A partir do código intermediário otimizado, anteriormente gerado, obter-se-ia o código objeto final baseado na linguagem simbólica de um microcomputador PC 8086.

```
L0      MOV  AX, I
        CMP  AX, 100
        JGE  L2
        MOV  AX, J
        MOV  BX, I
        ADD  BX
        MOV  I, AX
        JMP  L0
L2
```

Gerência de Tabelas

- Este módulo não constitui uma fase no sentido das anteriores, mas compreende um conjunto de tabelas e rotinas associadas que são utilizadas por quase todas as fases do tradutor.

- Algumas das tabelas usadas são fixas para cada linguagem, por exemplo, a tabela de palavras reservadas, tabelas de delimitadores, etc.

Gerência de Tabelas

- Entretanto, a estrutura que possui importância fundamental, é a que é montada durante a análise do programa fonte, com informações sobre:
  - declarações de variáveis;
  - declarações dos procedimentos ou sub-rotinas;
  - parâmetros de sub-rotinas; etc.

Gerência de Tabelas

- Estas informações são armazenadas na Tabela de Símbolos (às vezes chamada de tabela de nomes ou lista de identificadores).
- A cada ocorrência de um identificador no programa fonte, a tabela é acessada e o identificador é procurado na tabela.
- Quando encontrado, as informações associadas a ele são comparadas com as informações obtidas no programa fonte, sendo que, qualquer nova informação é inserida na tabela.

Gerência de Tabelas

- Os dados a serem coletados e armazenados na tabela de símbolos dependem da linguagem, do projeto do tradutor, do programa objeto a ser gerado.

- Entretanto, os atributos mais comumentemente registrados são:
  - para variáveis: classe (var), tipo, endereço no texto, precisão, tamanho;
  - parâmetros formais: classe (par), tipo, mecanismo de passagem;
  - procedimentos/sub-rotinas: classe (proc), número de parâmetros;

Tabela de Símbolos

- A Tabela de Símbolos deve ser estruturada de uma forma tal que permita rápida inserção e extração de informações, porém, deve ser tão compacta quanto possível.

Atendimento a Erros

- Esta fase tem por objetivo "tratar os erros" que são detectados em todas as fases de análise do programa fonte.
- Qualquer fase analítica deve prosseguir em sua análise, ainda que erros tenham sido detectados.
- Isto pode ser realizado, através de mecanismos de recuperação de erros, encarregados de re–sincronizar a fase com o ponto do texto em análise.
- A perda deste sincronismo faria a análise prosseguir de forma errada, propagando o efeito do erro.

Atendimento a Erros

- É fundamental que o tradutor prossiga na tradução, após a detecção de erros, de modo que o texto seja totalmente analisado.

Geradores de Compiladores

- Atualmente, a implementação de linguagens de programação é apoiada por sistemas geradores de compiladores (*compiler–compilers, compiler generators, translator–writing systems*).

- Estes sistemas classificam-se em três grupos:

Tipos de Geradores de Compiladores

a) geradores de analisadores léxicos: geram automaticamente reconhecedores para os símbolos léxicos (palavras-chave, identificadores, operadores, etc.) a partir de especificações de gramáticas ou expressões regulares.

Tipos de Geradores de Compiladores

b) geradores de analisadores sintáticos: produzem reconhecedores sintáticos a partir de gramáticas livres–do–contexto. Inicialmente, a implementação da análise sintática consumia grande esforço na construção de compiladores. Hoje, esta fase é considerada uma das mais fáceis de implementar.

Tipos de Geradores de Compiladores

c) geradores de geradores de código: estes sistemas recebem como entrada regras que definem a tradução de cada operação da linguagem intermediária para a linguagem de máquina. As regras devem incluir detalhes suficientes para possibilitar a manipulação de diferentes métodos de acesso a dados (por exemplo, uma variável pode estar em registradores, em memória ou na pilha da máquina). Em geral, instruções intermediárias são mapeadas para esqueletos que representam seqüências de instruções de máquina

Exercícios

1) No contexto de implementação de linguagens de programação, dê o significado dos seguintes termos: compilador, interpretador, montador, pré–compilador.

2) Aponte as vantagens e desvantagens dos interpretadores em relação aos compiladores.

3) Explique o processo de compilação: fases e seu inter-relacionamento.

4) Qual o significado de "passo" no processo de compilação? Quais as vantagens e desvantagens de implementar um compilador em vários passos?

Introdução à Compilação

Linguagens Formais A

Prof. Giovani Rubert Librelotto