Teoria da Computação e Compiladores

Análise Sintática

Prof. Dr. Fernando Kakugawa

fernando.kakugawa@animaeducacao.com.br

Universidade Anhembi Morumbi

Analisador Sintático

- Analisador sintático ou parser:
  - Entrada: sequência de tokens provenientes do analisador léxico;
  - Saída: árvore de parse da sequência, caso esteja de acordo com a gramática;
    - Muitas vezes esta árvore não é construída explicitamente e as fases seguintes executam-se em simultâneo com a análise sintática.

Análise Sintática

- Verifica se uma dada sequência de tokens constitui um programa válido;
- Compete ao analisador sintático encontrar erros na sequência de tokens e reportá-los o mais cedo possível, sem cessar a análise do resto da sequência;
- Caso a sequência esteja correta, extrair a sua estrutura (construir a árvore do parser) de acordo com as regras gramaticais que especificam a linguagem.

Análise Sintática

- Outras tarefas que podem ser realizadas conjuntamente com a análise sintática;
  - extrair, calcular e armazenar informações (atributos) relativamente aos símbolos gramaticais (terminais e não-terminais);
  - fazer verificação relativamente a esses atributos (p.ex. consistência de tipos);
  - gerar as “instruções” do código intermediário.

Obs: as primeiras duas fases enunciadas constituem a análise semântica.

Construção de um analisador Sintático

- Necessário
  - Um formalismo para especificar a linguagem;
  - Um método eficiente para determinar se uma dada sequência de tokens está ou não contida na linguagem;
- Especificação das linguagens de programação;
  - Gramáticas de contexto livre;
- Método;
  - Tentar uma derivação da sequência de tokens por aplicação das regras gramáticas (produções) a partir do símbolo inicial da gramática.

Construção de um analisador Sintático

- Análise **top-down**
  - Constrói-se a árvore de parser partindo da raiz (símbolo inicial) até se atingirem todos os terminais (tokens);
  - É o método preferido caso se implemente o analisador sintático “à mão”;
  - O percurso na árvore é feito em pré-ordem;
    - raiz, esquerda, direita.

Construção de um analisador Sintático

- Análise **top-down**
  - Utiliza-se derivação mais a esquerda;
  - Os algoritmos utilizados na análise top-down são o descendente recursivo e o LL(k);
  - O método LL(k) tem o seguinte significado:
    - Primeiro L: informa que o processamento é da esquerda para a direita;
    - Segundo L: o analisador utiliza a derivação à esquerda para a cadeia de entrada;
    - Símbolo k: significa que ele utiliza no máximo *k* símbolos a frente para decidir a direção da análise.

Construção de um analisador Sintático

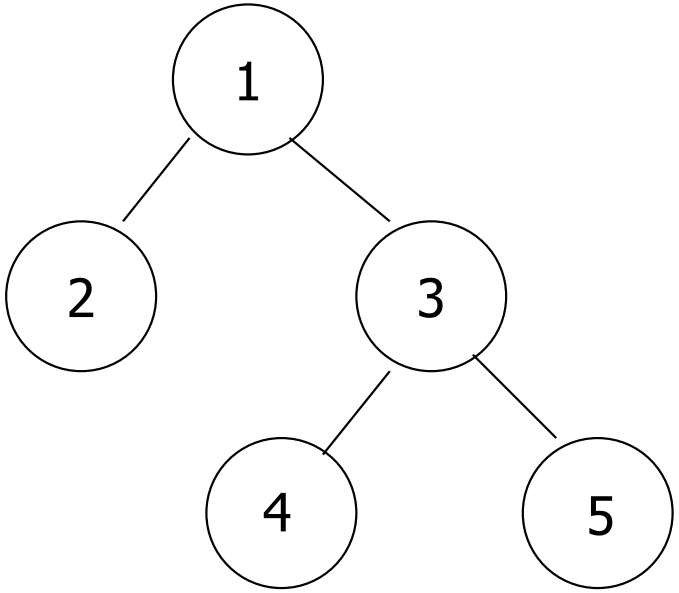
- Análise **bottom-up**
  - A construção da árvore de parser inicia-se nas folhas (tokens) e prossegue até se atingir o símbolo inicial, que constitui a raiz da árvore;
  - Constitui um método bastante poderoso, mas difícil de implementar, pelo que se recorre geralmente a ferramenta automáticas para gerar estes tipos de parsers;
  - O percurso na árvore é feito em pós-ordem;
    - esquerda, direita, raiz.

Construção de um analisador Sintático

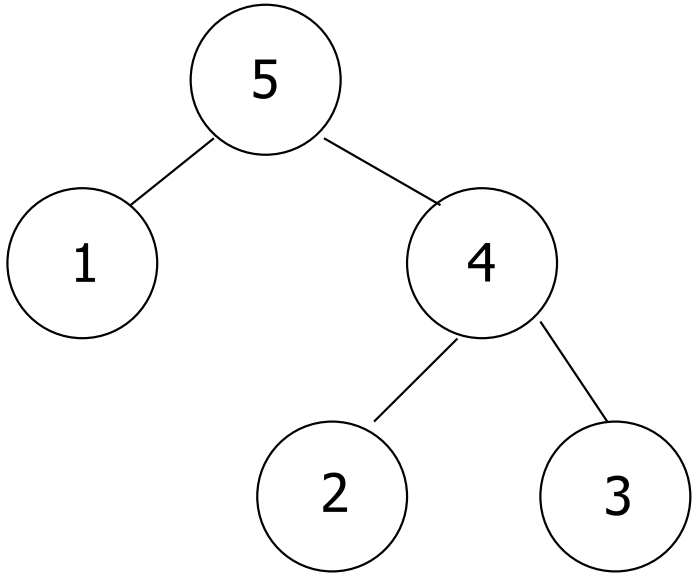
- Análise **bottom-up**
  - Utiliza-se derivação mais a direita;
  - Os algoritmos utilizados na análise bottom-up são a análise ascendente e o LR(1);
  - O método LR(1) tem o seguinte significado:
    - L: informa que o processamento é da esquerda para a direita;
    - R: o analisador utiliza a derivação à direita para a cadeia de entrada;
    - Símbolo 1: significa que ele utiliza apenas um símbolo para decidir a direção da análise.

Construção de um analisador Sintático

Pré-Ordem



Pós-Ordem



Exemplo

- Considere a gramática:

  Exp→ Exp Op Exp | ‘(’ Exp ‘)’ | num

  Op → ‘+’ | ‘-’ | ‘*’ | ‘/’

- E a sequência de tokens:

  (num - num) * num

- Em uma análise top-down partimos do símbolo inicial <Exp> e tentamos chegar à sequência de tokens através de uma derivação mais à esquerda:

Exemplo

Exp $\Rightarrow$ Exp Op Exp

$\Rightarrow$ (Exp) Op Exp

$\Rightarrow$ (Exp Op Exp) Op Exp

$\Rightarrow$ (num Op Exp) Op Exp

$\Rightarrow$ (num - Exp) Op Exp

$\Rightarrow$ (num - num) Op Exp

$\Rightarrow$ (num - num) * Exp

$\Rightarrow$ (num - num) * num

Exemplo

- Em uma análise bottom-up partimos da sequência de tokens que pretendemos analisar e vamos aplicando regras gramaticais (ao contrário), até se chegar ao símbolo inicial:

Exemplo

⇐ (num - num) * num

⇐ (Exp - num) * num

⇐ (Exp Op num) * num

⇐ (Exp Op Exp) * num

⇐ (Exp) *num

⇐ Exp*num

⇐ Exp Op num

⇐ Exp Op Exp

⇐ Exp

COMPILADORES

Análise sintática

top-down

Análise Sintática top-down

- Neste tipo de análise sintática, a árvore de parser é construída partindo da raiz, até se chegar à sequência de tokens de texto da entrada:
  - Gera sempre uma derivação mais à esquerda;
  - Os nodes da árvore de parser construída são visitados em pré-ordem;
    - Descida recursiva.

Análise Sintática top-down

- São usados essencialmente dois métodos para implementar a análise top-down:
  - O método da descida recursiva, em que se associa uma rotina a cada não-terminal da gramática que é chamada quando se pretende reescrever esse símbolo;
  - O método de análise preditiva não-recursiva (LL(1)), que necessita de uma pilha auxiliar e é guiada por uma tabela de parser.

Análise Sintática top-down

- O retrocesso pode ser necessário para decidir *qual produção usar em determinado ponto;*

- A recursividade implica na formação de uma pilha semântica que controla a utilização dos símbolos gramaticais.

Análise LL(1)

- Utiliza uma pilha explícita ao invés de ativações recursivas;
- Para ilustrar o funcionamento do método será utilizado uma gramática que gera cadeias de parênteses balanceados:
  - $S \rightarrow (S)S \mid \varepsilon$
- A pilha inicia com o símbolo $ marcando a pilha e o símbolo não-terminal inicial:
  - $S

Análise LL(1)

Este método gera uma tabela de ações sintáticas.

Passos	Pilha	Entrada	Ação
1	$S	()$	$S \rightarrow (S)S$
2	$S)S(	()$	casamento
3	$S)S	)$	$S \rightarrow \varepsilon$
4	$S)	)$	casamento
5	$S	$	$S \rightarrow \varepsilon$
6	$	$	aceita

Para verificar se a cadeia de entrada é válida, a pilha sintática e a entrada devem estar vazias ao final do processamento

Descendente recursivo

- Considere a seguinte gramática para uma linguagem:

<expr>→ <expr_c> EOF

<expr_c>→ <termo> {+<expr_c>}*

<termo>→ <fator> {*<termo>}*

<fator>→ <primario> {**<fator>}*

<primario> → IDENT | NUMERO | (<expr_c>)

- O Programa desta Gramática está no DescendenteRecursivo.txt

Descedente recursivo

- Na descida recursiva escreve-se uma rotina para cada não-terminal da gramática, começando-se pelo símbolo inicial da gramática.

Exemplo

- Construir um Analisador Sintático em pseudo-código para a seguinte gramática:

  M → a | (N)

  N → MX

  X → b | ε

- Símbolo Inicial: M

Descendente recursivo - dificuldades

- Considere agora a seguinte gramática para expressões aritméticas

$S \rightarrow E\$$	$T \rightarrow T * F$	$F \rightarrow id$
$E \rightarrow E + T$	$T \rightarrow T / F$	$F \rightarrow num$
$E \rightarrow E - T$	$T \rightarrow F$	$F \rightarrow (E)$
$E \rightarrow T$		

- Como escrever as rotinas relativas a E e T?
- Não sabemos como distinguir as várias produções.
- Mesmo sabendo, as primeiras 2 produções de E e T produziriam chamadas recursivas infinitas (loop infinito)

Descendente recursivo - dificuldades

- No entanto existem métodos para eliminar a recursividade à esquerda de uma gramática e para distinguir entre regras, quando um não-terminal possui várias produções que começam por não-terminais distintos, ou símbolos iguais.

Descendente recursivo - dificuldades

- São elas:
- Recursividade à esquerda: ($A \Rightarrow A\alpha$)
  - Fazer uma eliminação da recursividade à esquerda, transformando a gramática;
- Produções que comecem com o mesmo símbolo: ($A \Rightarrow X\alpha \mid X\beta$)
  - Fazer uma fatoração à esquerda

Eliminação da recursividade à esquerda

- Seja uma gramática que contém produções com recursividade à esquerda imediata da forma:

  $\mathbf{A} \Rightarrow \mathbf{A}\alpha_1 \mid \mathbf{A}\alpha_2 \mid ... \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_m$

  - onde β_1 a β_m não começam por A

- Então aquele conjunto de regras pode ser substituído, de forma equivalente, pelo seguinte conjunto:

  $\mathbf{A} \Rightarrow \beta_1 \mathbf{A'} \mid \beta_2 \mathbf{A'} \mid ... \mid \beta_m \mathbf{A'}$

  $\mathbf{A'} \Rightarrow \alpha_1 \mathbf{A'} \mid \alpha_2 \mathbf{A'} \mid ... \mid \alpha_n \mathbf{A'} \mid \varepsilon$

Eliminação da recursividade à esquerda

- Exemplo:
  - Sejam as regras para expressão do exemplo anterior:

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

Eliminação da recursividade à esquerda

- Exemplo:
  - Essas regras podem ser transformadas no conjunto

    $E \rightarrow TE'$

    Onde:

    $E' \rightarrow + TE'$

    $E' \rightarrow - TE'$

    $E' \rightarrow \varepsilon$

Fatorização à esquerda

- Quando mais do que uma regra, para o mesmo não-terminal, começam pelo mesmo símbolo é difícil escolher, examinando apenas o próximo token.
- Solução: atrasar a escolha até haver distinção

Fatorização à esquerda

- Algoritmo de fatorização à esquerda:

1. Para cada não terminal A da gramática encontrar o prefixo mais longo α, comum a duas ou mais das suas regras;
2. Se $\alpha \neq \varepsilon$, substituir todas as produções de

   A $\rightarrow \alpha\beta_1 | \alpha\beta_2 | ... | \alpha\beta_n | \gamma$, onde γ representa regras que não começam por α por:

   A $\rightarrow \alpha$**A'** $| \gamma$

   A' $\rightarrow \beta_1 | \beta_2 | ... | \beta_n$
3. Repetir o processo até $\alpha = \varepsilon$, para todas as regras

Fatorização à esquerda

- Exemplo:
  - Seja a gramática:

    S $\rightarrow$ if E then S | if E then S else S

  - Aplicando a fatorização à esquerda obtemos:

    S $\rightarrow$ if E then S **X**

    X$\rightarrow$ else S | ε

Exemplo

- Elimine a recursividade a esquerda das seguintes gramáticas:
  - $X \rightarrow Xb \mid c \mid d$
  - $M \rightarrow Mc \mid Mac \mid bd \mid \varepsilon$

Tratamento de Erros

- O tratamento de erros durante a análise sintática deve obedecer a certos princípios gerais:
  - O erro deve ser declarado o mais cedo possível
  - Continuar a análise, de modo a encontrar o número máximo de erros
  - Evitar que um único erro gere uma série de mensagens
  - Evitar ciclos infinitos, gerando sucessivamente as mesmas mensagens de erros

Tratamento de Erros

- Problemas com a determinação da extensão do erro são facilmente percebidos se tentarmos fazer o parsing de um programa como o dado abaixo:

```
main ()
{
    int a,b=1;
    float c,e,


    if (b)
       a=0;
    c= b-a;
    e= c**2;
}
```

Tratamento de Erros

- Após a detecção do erro ao lermos o token ***if***, temos que encontrar uma forma de retomar o processo de compilação, para irmos até o final do arquivo.
- Isso é feito mais por aspectos históricos do que por necessidade real. Acontece que nos primórdios da computação os programadores tinham que esperar muito tempo pelo resultado da compilação.
- Assim, se a cada erro o compilador fosse abortar todo o processo, teríamos que tentar a compilação muitas vezes, ocupando dias de atividade, até depurarmos a sintaxe do programa. Com isso, era necessário que o compilador sempre indicasse todos os erros que fossem identificáveis numa única passagem, para agilizar o processo de depuração.
- Isso persiste até hoje, muito embora os computadores atuais não nos forcem a ter tal atitude.

Tratamento de Erros

- O maior problema para fazer com que o compilador chegasse ao final do programa era determinar a partir de que token, o compilador teria que retomar a compilação, isto é, quando é que o atual erro deixaria de influenciar no restante do programa
  - A solução para esse problema pode ser encontrada de maneiras distintas

Tratamento de Erros

- Recuperação no nível de frase:
  - Embora este método não tenha muita aplicação prática, o mesmo serve como base para entendermos qual é, de fato, o processo de recuperação de erros.
  - O que se faz nesse método é tentar corrigir o fonte (de modo imaginário é claro), através de inserção, deleção ou modificação de tokens, para atingir um parsing correto.

Tratamento de Erros

- Recuperação no nível de frase:
  - Se o mesmo for feito no nível de frase, ou comando, o que se busca é a alteração da frase para que ela passe a ser considerada como gramaticalmente correta, num processo de minimização de erros ou alterações
  - O problema desse método é que o mesmo é complexo e ainda pode causar a entrada em loops, ao inserir tokens que acabem exigindo sempre a inserção de mais tokens para fazer a correção da frase

Tratamento de Erros

- Recuperação por pânico:
  - Aqui temos o mais simples de todos os métodos de recuperação, que vai simplesmente ignorar todos os tokens seguintes ao ponto em que o erro foi determinado, até encontrar um token que tenha, de fato, uma posição precisamente definida dentro de qualquer programa

Tratamento de Erros

- Recuperação por pânico:
  - Assim, tokens como **'{' , '}' , 'while' , 'for',** etc., que marcam precisamente ou o início ou o final de um bloco de comandos (frases) podem ser usados para determinar o momento de reinício do parsing.
  - A simplicidade desse modelo faz com que ele seja bastante usado quando fazemos a implementação do compilador, mas peca pelo fato de que pode acabar por ignorar alguns erros que estejam embutidos entre os tokens não analisados pelo parser.

Exercício

- G = ({S,T},{a,b,c,d},P, S), onde P é:
  - S -> baT | Sbc | c
  - T -> cdT | ε
- Elimine a recursão a esquerda
- Construa um analisador sintático em pseudo-código
- Elabore uma string que seja aceita por esta gramática com, pelo menos 6 caracteres
- Monte uma árvore de derivação para a string elaborada

Material elaborado por:

Prof. Dr. Augusto Mendes Gomes Jr.

augusto.gomes@animaeducacao.com.br

Prof. Dr. Fernando Kakugawa

fernando.kakugawa@animaeducacao.com.br