Compreendendo a Análise Sintática Descendente Recursiva

por Antonio Atta
atta@uneb.br

A Análise Sintática Descendente Recursiva (ASDR) é uma das formas mais usuais de se implementar compiladores de linguagens de programação pela aplicação do desenvolvimento e codificação manual do compilador. Nessa proposta, o modelo de tradução adotado ocorre em um passo, com o processo de compilação sendo dirigido pela execução da análise sintática – as etapas seguintes do projeto do compilador, quais sejam, análise semântica, geração de código intermediário etc. são executadas de forma concomitante com a análise sintática. Isto não quer dizer que o desenvolvimento do compilador não ocorra de forma modular. É possível escrever e testar todo o código do analisador sintático e somente depois seguir inserindo no mesmo o código referente às outras etapas do processo de compilação. Dessa forma, o código do analisador sintático finda por conter também, embutido nele, o código de todas as etapas restantes do compilador, sendo possível, no entanto, identificar claramente todas as etapas do processo de compilação, daí a noção de *Tradução Dirigida pela Sintaxe*. É importante compreender que no modelo ASDR a árvore sintática que é construída ao longo do processo da análise sintática é implícita, ou seja, associada às chamadas e retornos dos procedimentos que implementam as regras de produção da gramática (ou, mais especificamente, os seus componentes não-terminais). Portanto, essa árvore sintática implícita "cresce" e "decresce" dinamicamente à medida que a análise sintática ocorre, como veremos a seguir.

ASDR completa a partir de uma linguagem e gramática simples (mas funcional)

A fim de ilustrar de maneira prática o processo do desenvolvimento de um compilador que emprega a ASDR, tomemos como exemplo um compilador de expressões aritméticas, que gera o resultado calculado de uma expressão aritmética como código alvo do processo de tradução. As etapas a seguir ilustram o desenvolvimento de um ASDR para esse projeto.

A Gramática de Expressões Aritméticas

Tudo começa com a gramática da linguagem. Usemos como ponto de partida a gramática simples abaixo, onde **Expr** é o símbolo não-terminal de partida:

Gramática simples para Expressões Aritméticas (regras de produção)

Expr → **Expr** '+' **Expr** | **Expr** '–' **Expr** | **Expr** '*' **Expr** | **Expr** '/ ' **Expr** | **num**

Figura 1- Gramática simples para expressões aritméticas

O problema da ambiguidade

Apesar da gramática da Figura 1 gerar expressões aritméticas válidas, ela não é adequada para a aplicação da ASDR porque ela é ambígua. É possível montar mais de uma árvore de derivação para uma mesma expressão aritmética exemplo. Não detalharei o conceito de ambiguidade aqui – recomendo a consulta ao "livro do dragão" (AHO, 2008) para melhor compreensão deste aspecto. Além disso, essa gramática não dá suporte a outros elementos importantes característicos de uma linguagem de expressões aritméticas, a exemplo do respeito à ordem de precedência dos operadores na execução da expressão e da possibilidade do uso de parênteses para a "quebra" dessa precedência. Uma modificação possível na busca por uma gramática equivalente (que gera cadeias de uma mesma linguagem – no caso, expressões aritméticas válidas), mas não ambígua, nos leva à seguinte gramática:

Gramática simples equivalente para Expressões Aritméticas (regras de produção)

Expr → **Expr** '+' **Termo** | **Expr** '–' **Termo** | **Termo**
Termo → **Termo** '*' **Fator** | **Termo** '/' **Fator** | **Fator**
Fator → '(' **Expr** ')' | **num**

Figura 2 - Gramática equivalente sem ambiguidade

A gramática da Figura 2, não só resolve o problema da ambiguidade registrado na gramática da Figura 1, como também acrescenta o aspecto semântico do respeito à precedência dos operadores e à ocorrência de parênteses quando necessário. Isso é obtido a partir da inserção do não-terminal **Termo** que gera as árvores filhas de derivação, associadas às operações de maior precedência (multiplicação e divisão) sempre abaixo (ou internamente) às árvores filhas de derivação associadas às operações de menor precedência (soma e subtração). Essa característica garante que, no processo de tradução efetuado juntamente com a análise sintática, as operações de multiplicação e divisão gerem código antes da geração do código associado às operações de soma e subtração – em outras palavras, o resultado das operações de multiplicação e divisão passam a ser operandos para as operações de soma e subtração quando essas operações ocorrem simultaneamente em uma expressão aritmética. Efeito semelhante ocorre com o não-terminal **Fator** que trata a possível ocorrência de parênteses. Exercite essa percepção montando manualmente as árvores sintáticas para expressões como "2 + 3 * 5", "(2 + 3) * 5", "(2 * 10) / (2 + 3)" e outras que você mesmo crie (contanto que estejam sintaticamente corretas) a partir da gramática da Figura 2.

O problema da recursão à esquerda

A gramática da Figura 2, como visto, é mais completa e mais adequada, à uma processo de implementação de um tradutor usando o computador (já que não é ambígua), mas padece de um outro problema nocivo à implementação de compiladores de expressões aritméticas pela

técnica da ASDR: a recursão à esquerda. Esse problema ocorre quando a técnica ASDR é usada porque a ideia básica que caracteriza essa técnica é a montagem das árvores sintáticas, implicitamente, usando o próprio controle do fluxo de execução do programa, que efetua a análise sintática, com chamadas recursivas a procedimentos, como mecanismo de construção das árvores. Explicando melhor: na ASDR cada símbolo não-terminal da gramática (**Termo**, por exemplo) é associado a um procedimento na codificação do analisador sintático que deve ser chamado na ordem em que eles ocorrem nas regras de produção. Se tomarmos como exemplo a regra de produção " **Termo → Termo '*' Fator** " na gramática da Figura 2, temos que **Termo** é um não-terminal e, portanto, possui um procedimento "Termo()" associado no analisador sintático, cujo corpo de comandos [de Termo()] deve seguir a seguinte sequência:

a. chamar o procedimento Termo();
b. reconhecer a ocorrência do operador '*';
c. chamar o procedimento Fator().

O problema reside justamente no fato de Termo() "chamar" Termo(), recursivamente, como primeiro comando do seu próprio bloco de comandos, conforme indicado pela gramática, sem consumir antes qualquer símbolo (*token*) da expressão sendo analisada. Isso coloca o analisador sintático em laço de recursão infinita e inviabiliza a implementação computacional do analisador sintático pela técnica ASDR.

Para sanar essa recursão à esquerda e podermos aplicar a ASDR é necessário modificar uma segunda vez a gramática e encontrar uma gramática equivalente que resolva as ocorrências de recursão à esquerda. Novamente, o "livro do dragão" (AHO, 2008) descreve uma forma geral de executar essa transformação, que consiste basicamente na reescrita das regras de produção onde a recursão à esquerda se apresenta – recomenda-se fortemente que você pare por um momento agora e estude esse processo para melhor compreensão e motivação antes de seguir adiante.

A gramática da Figura 2, transformada então para uma gramática equivalente livre da recursão à esquerda (e que também não é ambígua e respeita a precedência de operadores), é apresentada na Figura 3.

Gramática simples para Expressões Aritméticas sem recursão à esquerda (regras de produção)

Expr → Termo Resto

Termo → Fator Sobra

Resto → ' +' Termo Resto | ' –' Termo Resto | ε

Sobra → ' *' Fator Sobra | ' /' Fator Sobra | ε

Fator → '(' Expr ')' | num

Figura 3 – Gramática equivalente sem recursão à esquerda

Note que a recursividade continua a existir nas regras de produção da gramática da Figura 3, mas o fenômeno da recursão à esquerda deixa de ocorrer porque antes das chamadas recursivas

algum símbolo terminal da expressão de entrada (*token*) será consumido antes da recursão ocorrer; com isso, como a expressão de entrada é sempre finita, fica garantida a não ocorrência de laço infinito de recursão, posto que, em algum momento, a expressão de entrada termina. A gramática da Figura 3 está adequada para a implementação do ASDR.

Partindo para a prática: implementando o ASDR

Como primeira etapa na implementação de qualquer Analisador Sintático, vamos precisar de um Analisador Léxico fornecedor de *tokens* (símbolos terminais da expressão de entrada) para alimentar o ASDR. A Figura 4 apresenta um AFD (Autômato Finito Determinístico) possível para a linguagem de expressões aritméticas apresentada. O código C das Figuras 5, 6 e 7 implementam esse AFD sob a forma de uma função denominada Analex(), que a cada chamada preenche um *token* (variável declarada globalmente) com o próximo símbolo terminal localizado na entrada padrão (*stdin*) onde a expressão de entrada deve ser digitada.

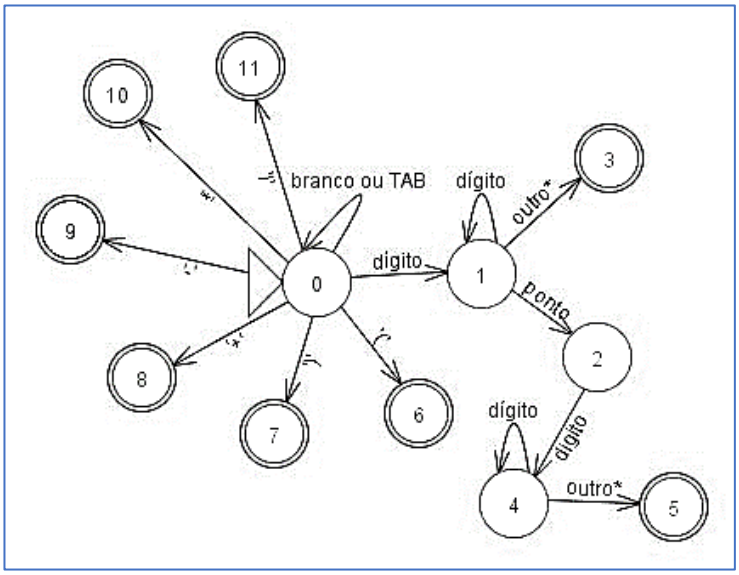

Figura 4 - AFD para tokens de Expressões Aritméticas

Na codificação de Analex(), a opção pela definição do *token* como variável global foi tomada levando em consideração a legibilidade e facilidade de compreensão da implementação da ASDR mais adiante – o *token* em análise a cada instante será massivamente utilizado por todas os procedimentos do ASDR.

A codificação apresentada do Analisador Léxico acrescenta ainda o reconhecimento de um *token* especial, o sinal de igual (=) que não ocorre no AFD e que deve ser usado, por conveniência, na implementação para indicar o final da expressão aritmética.

Como já apresentei, na implementação da ASDR a árvore sintática ou árvore de derivação é montada implicitamente a partir do fluxo de chamadas e retornos de procedimentos que ocorre na execução de um código escrito em uma linguagem de programação de alto nível típica como o C. Para que esse modelo funcione, associamos um procedimento a cada símbolo **não-terminal** da gramática, que deve ser executado de acordo com a ordem de ocorrência dos mesmos nas diversas regras de produção que compõem a gramática. Assim, na implementação do procedimento associado ao não-terminal **Fator,** a partir da regra de produção "**Fator** → '(' **Expr** ')' | **num**" da gramática da Figura 3, devemos verificar se o *token* atual é um número ou um abre parênteses; se for um número, reconhecemos esse *token* como válido chamando a função Analex() para acesso o próximo *token* e a função Fator() deve retornar; se for um abre parênteses, reconhecemos esse *token* como válido, novamente chamando a função Analex() para acesso ao próximo *token*, e em seguida chamamos a função Expr() para tratar o símbolo não-terminal **Expr** que irá processar a análise sintática relacionada com a expressão entre os parênteses; para

finalizar, verificamos se o *token* atual é um fecha parênteses, conforme indicado na regra de produção. Caso, ao chamar a função Fator(), o *token* atual não seja nem um número válido nem um abre parênteses, uma mensagem de erro sintático deve ser emitida pois **Fator** não possui nenhuma regra de produção com saída a partir do símbolo de cadeia vazia (**ε**), como é o caso das regras de produção do não-terminal **Resto**. Quando um símbolo não-terminal possui uma regra de produção com o símbolo de cadeia vazia (**ε**) no seu lado direito, devemos retornar sem emitir mensagem de erro quando não encontramos casamento para o *token* atual na derivação desse símbolo não-terminal; em outras palavras, deixamos que esse *token* atual seja tratado pelo procedimento associado a outro símbolo não-terminal no processo de montagem da árvores sintática.

```
#ifndef ANALEX
#define ANALEX

#include <stdio.h>

typedef
enum {OP=1, SN}
TCAT;

typedef
enum {SOMA=1, SUBT, MULT, DIVI, ABRE_P, FECHA_P, IGUAL, NOOP}
TCOD;

typedef
struct {
    TCAT cat;
    union {
       TCOD cod;
       float valor;
    };
} TOKEN;

/* Variaveis globais */
TOKEN t;
char digitos[15];
int indDig;

/* Assinaturas de funcoes  */
void Erro(int);
void Analex(FILE *);

#endif // ANALEX
```

Figura 5 - analex.h, arquivo header do Analisador Léxico

A Figura 8 apresenta o arquivo codificado em linguagem C com as assinaturas das funções que comporão o nosso ASDR para expressões aritméticas associados à gramática da Figura 3. Por enquanto, estaremos preocupados apenas com a ASDR sem nos atermos a aspectos de geração da tradução propriamente dita (ou seja, o cálculo do resultado da expressão).

A Figura 9 apresenta a codificação completa do analisador sintático de expressões aritméticas usando a técnica descendente recursiva de implementação – ASDR.

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "analex.h"

void Analex(FILE *fd) {

    char c;
    int estado;


    c=fgetc(stdin);
    estado=0;

    while (1) {
        switch (estado) {
        case 0:
            if (c==' ' || c=='\t') estado=0;
            else if (isdigit(c)) {
                estado=1;
                digitos[0]=c;
                indDig=1;
            }
            else if (c=='+') {t.cat=SN; t.cod=SOMA; return;}
            else if (c=='-') {t.cat=SN; t.cod=SUBT; return;}
            else if (c=='*') {t.cat=SN; t.cod=MULT; return;}
            else if (c=='/') {t.cat=SN; t.cod=DIVI; return;}
            else if (c=='(') {t.cat=SN; t.cod=ABRE_P; return;}
            else if (c==')') {t.cat=SN; t.cod=FECHA_P; return;}
            else if (c=='=') {t.cat=SN; t.cod=IGUAL; return;}
            else Erro(1);
            break;
        case 1:
            if (isdigit(c)) {
                estado=1;
                digitos[indDig++]=c;
            }
            else if (c=='.') {
                estado=2;
                digitos[indDig++]=c;
            }
            else {
                estado=3;
                ungetc(c, stdin);
                t.cat=OP;
                digitos[indDig]='\0';
                t.valor=atof(digitos);
                return;
            }
            break;
```

Figura 6 - Implementação do Analisador Léxico - Parte 1

```
        case 2:
            if (isdigit(c)) {
                estado=4;
                digitos[indDig++]=c;
            }
            else Erro(2);
            break;
        case 4:
            if (isdigit(c)) {
                estado=4;
                digitos[indDig++]=c;
            }
            else {
                estado=5;
                ungetc(c, stdin);
                t.cat=OP;
                digitos[indDig]='\0';
                t.valor=atof(digitos);
                return;
            }
        }
        c=fgetc(stdin);
    }
}
```

Figura 7 - Implementação do Analisador Léxico - Parte 2

```
#ifndef ANASINT
#define ANASINT

/* Variaveis globais  */
extern TOKEN t;

/* Assinaturas de funcoes  */
void Expr();
void Termo();
void Resto();
void Sobra();
void Fator();

#endif // ANASINT
```

Figura 8 - anasint.h, arquivo header C com as assinaturas das funções do Analisador Sintático

```c
#include <stdlib.h>
#include "analex.h"
#include "anasint.h"

void Expr() {    // EXPR

    Termo();
    Resto();
}

void Termo() {

    Fator();
    Sobra();
}

void Resto() {    // RESTO

    if ((t.cat==SN && t.cod==SOMA) || (t.cat==SN && t.cod==SUBT))  {
        Analex(stdin);
        Termo();
        Resto();
    }
    else ;        // saida por vazio
}

void Sobra() {    // SOBRA

    if ((t.cat==SN && t.cod==MULT) || (t.cat==SN && t.cod==DIVI))  {
        Analex(stdin);
        Fator();
        Sobra();
    }
    else ;       // saida por vazio
}

void Fator() {  // FATOR

    if (t.cat==SN && t.cod==ABRE_P) {
        Analex(stdin);
        Expr();
        if (t.cat!=SN || t.cod!=FECHA_P) {
            Erro(3);
        }
        Analex(stdin);
    }
    else if (t.cat==OP) {
        Analex(stdin);
    }
    else Erro(4);

}
```

Figura 9 - Implementação dos procedimentos do Analisador Sintático (ASDR)

Para permitir os testes das codificações dos analisadores léxico e sintático apresentados, as Figuras 10 e 11 abaixo apresentam a codificação da rotina de emissão de erros e um programa

principal exemplo (autoexplicativo), respectivamente, que podem ser usados na compilação do ASDR de expressões aritméticas.

```
#include <stdio.h>
#include <stdlib.h>

char erros[][50] =  {"Sem erro",
                     "Caracter invalido!",
                     "Formato de numero invalido!",
                     "Faltando fecha parenteses!",
                     "Operando invalido!"};

void Erro(int e) {

    printf("Erro: %s\n", erros[e]);
    exit(e);
}
```

Figura 10 - Arquivo de tratamento de erros

```
#include <stdio.h>
#include <stdlib.h>
#include "analex.h"
#include "anasint.h"


int main()
{
    printf("Digite expressões aritméticas no formato padrão (modo infixo).\n");
    printf("Termine cada expressão com o sinal de igualdade (=) seguido de
<enter>.\n");
    printf("Parênteses podem ser usados para quebrar a ordem de precedência dos
\n");
    printf("operadores (+, -, *, e /). Ao final de cada expressão, se ela
estiver correta,\n");
    printf("uma mensagem com essa indicação será emitida e, em seguida, uma
nova \n");
    printf("expressao pode ser digitada.\n");
    printf("Termine a entrada de expressões com um sinal de igualdade (=) sem
expressao.\n");
    printf("Ocorrencias de caracteres após o sinal de igualdade (=) serão
desprezadas.\n");
    printf("\nExpr> ");
    Analex(stdin);
    while (t.cat != SN || t.cod != IGUAL) {
        Expr();
        if (t.cat==SN && t.cod==IGUAL)
            printf("Resultado> Expressão OK!\n");
        else
            Erro(1);
        printf("\nExpr> ");
        fflush(stdin);
        Analex(stdin);
    }
    printf("Ate a proxima!");
    return 0;
}
```

Figura 11 - Programa principal exemplo de teste do ASDR de Expressões Aritméticas

Considerações Finais

Recomenda-se que os exemplos de codificação C associados à gramática da linguagem de expressões aritméticas presentes neste texto sejam estudados até o completo entendimento do processo de construção de analisadores sintáticos pela técnica ASDR. Esse exercício é fundamental para desenvolver a habilidade de aplicação da técnica ASDR em compiladores de linguagens com gramáticas mais amplas (uma linguagem de programação de alto nível, por exemplo). Para facilitar esse estudo e permitir a manipulação dos códigos C do ASDR apresentado, disponibilizamos a codificação C completa dos procedimentos e o respectivo projeto (IDE Code::Blocks) em:

https://github.com/antonioatta/ASDR-Expressoes-Aritmeticas-AL-AS

Conforme visto, é fundamental que o projeto da gramática esteja em conformidade com os requisitos da técnica ASDR sob pena de inviabilizar a aplicação da técnica e a codificação manual do analisador sintático.

Todas as codificações C dos analisadores léxico e sintático apresentadas estão funcionais e podem ser incluídas em um projeto C usando uma IDE que disponha de um compilador C, como o Code::Blocks, para fins de testes dessas codificações. Recomenda-se, portanto, como complemento ao estudo (ou paralelamente a esse estudo) que os programas e procedimentos apresentados como parte prática do texto sejam experimentados em execução. Alguns exemplos de expressões que podem ser inicialmente usadas para testar o programa gerado são (sem o uso das aspas): “2+3+4=”, “(5 + 3)/(1 +3)=”, “2=”, “((4.5 + 5.5) / (2+3)) – 2 =”; algumas expressões aritméticas *mal formadas* podem ser usadas também para a verificação de que as análises léxica e sintática estão ocorrendo com exatidão.

Referências

Aho, Alfred V. [et al.]; **Compiladores:** Princípios , Técnicas e Ferramentas. 2ª Edição. São Paulo: Pearson Addison-Wesley. 2008. ISBN 978-8588639249