Compiladores

Capítulo 4. Gramáticas de Atributos

4.1 - Introdução

As técnicas de análise sintática vistas anteriormente descrevem basicamente reconhecedores, ou seja máquinas (algoritmos) que determinam se uma cadeia pertence ou não à linguagem reconhecida. Ou seja, os parsers descritos no capítulo anterior respondem a uma dada entrada *x* com "Sim", explicitando que a cadeia pertence à linguagem da gramática considerada, ou com "Não", isto é, não pertence. Entretanto, no caso de aceitação, a informação sobre a forma de derivação de *x* deve ser usada na geração do código para *x*, e, no caso de não aceitação, a informação disponível deve ser usada para tratamento de erros: sinalização (para que o programador possa corrigir o erro encontrado) e recuperação (para que o parser possa dar continuação à análise.

Supondo o sucesso da análise, a saída do analisador sintático pode descrever a forma de derivação da cadeia de entrada de diversas maneiras. Pode ser fornecida uma árvore de derivação da cadeia, ou, equivalentemente, pode ser fornecida a seqüência das regras utilizadas (parse esquerdo ou parse direito invertido, conforme o tipo do analisador).

Além dessa informação *sintática* sobre a forma de derivação da entrada, deve ficar disponível também a informação *léxica*, composta essencialmente pelas cadeias correspondentes aos chamados tokens variáveis (identificadores, literais inteiros, reais, cadeias, ...). Esta informação deve ser processada nas fases seguintes do compilador.

Este capítulo apresenta uma técnica geral de tratamento das informações obtidas nas fases de análise léxica e sintática, baseada em *gramáticas de atributos*. Usando essas gramáticas de atributos, podemos especificar qualquer forma de *tradução dirigida pela sintaxe*, associando valores - os *atributos* - aos símbolos terminais e nãoterminais de uma gramática livre de contexto.

Em sua forma mais geral, o uso de gramáticas de atributos implica na construção de árvores de derivação, em cujos nós são armazenados todos os valores de atributos de símbolos da árvore de derivação, e isso pode levar a um gasto considerável de tempo e de espaço no processo de compilação. Na realidade, não há necessidade de usar exatamente a árvore de derivação, podendo ser usadas árvores semelhantes, desde que contenham toda a informação relevante.

Vamos discutir inicialmente neste capítulo as técnicas mais gerais, e depois mostrar sua adaptação/restrição para aplicação aos casos particulares que permitem eficiência na construção e execução de compiladores.

Para ilustrar como uma árvore de derivação pode ser construída, vamos apresentar um exemplo em que a árvore é construída durante o processo de análise, por um analisador ascendente. O fato de que usamos um analisador ascendente fixa a ordem das ações (usualmente conhecidas como ações semânticas) associadas às regras usadas para as reduções. Por conveniência, rotulamos cada nó com o número da regra

aplicada, em vez do símbolo não terminal. Naturalmente, a partir do número da regra é possível obter diretamente o símbolo do nó (não terminal do lado esquerdo) e os símbolos dos filhos (terminais e não terminais do lado direito). Não incluímos, portanto, nós correspondentes a símbolos terminais. Esta forma de implementação foi escolhida para ser apresentada aqui, por ser mais compacta que uma árvore de derivação, apesar de conter a mesma informação, mas deve ficar claro que é apenas uma entre várias formas possíveis.

Exemplo 1: Seja a gramática G_0, dada por suas regras:

1. $E^0 \rightarrow E^1 + T$
2. $E \rightarrow T$
3. $T^0 \rightarrow T^1 * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Os índices superiores servem apenas para distinguir entre as várias ocorrências de um mesmo símbolo em uma regra. Assim, E^0 e E^1 são ocorrências distintas do nãoterminal E.

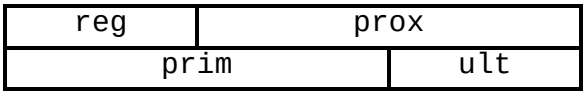
□

Vamos inicialmente definir o formato do nó da árvore. A árvore que vai ser construída neste exemplo terá nós com 0, 1 ou 2 filhos, mas, embora isso seja desnecessário aqui, vamos usar um formato de nó que permite nós com um número qualquer de filhos. Para isso vamos usar uma *lista de filhos*, em vez de um número fixo de campos para filhos. (Veja o tipo `no`, logo a seguir.) Em cada nó, um campo `prim` aponta para o primeiro filho desse nó; em cada nó o campo `prox` aponta para o próximo irmão, servindo para encadear a lista de irmãos.

O último filho não tem próximo irmão, e neste caso, o campo `prox`, em vez de apontar para o próximo irmão (inexistente), aponta para o nó pai. Para que fique claro qual é o conteúdo do campo `prox`, o fato de que este é o último filho é indicado pelo valor `TRUE` no campo booleano `ult`. Esta representação de árvores é às vezes conhecida como alinhavada (“*threaded*”). Um exemplo dessa implementação está representado na Fig. 1, na página 4-5.

```
type
   pt=^no;
   no=record
      reg:integer;          { número da regra }
      prim:pt;              { primeiro filho }
      prox:pt;              { próximo irmão ou pai }
      ult:boolean;          { último filho? }
   end;
```

Graficamente, temos o seguinte formato de nó:



O algoritmo utiliza uma pilha, chamada de *pilha semântica*. Este nome é usado em oposição ao nome de *pilha sintática*, que fica reservado para a pilha usada pelo

analisador sintático. Neste caso, a pilha semântica guarda os valores dos ponteiros para as árvores associadas aos nãoterminais que se encontram na pilha sintática. Essa pilha é implementada através do array `psem` e da variável (topo da pilha) `tsem`, e é manipulada pelos procedimentos/funções `push` e `pop`. O tratamento dos casos de *underflow* (`pop` na pilha vazia) e de *overflow* (`push` numa pilha que já atingiu o limite de tamanho) fica como exercício.

```
const maxpsem=100;                          { ou um valor apropriado }
var psem:array[1..maxpsem] of pt;
    tsem:integer;
                                            { iniciar com zero: pilha vazia }

function pop:pt;                        procedure push(p:pt);
begin                                   begin
  if tsem=0 then                          if tsem=maxpsem then
    underflow                               overflow
  else begin                              else begin
    pop:=psem[tsem];                        tsem:=tsem+1;
    tsem:=tsem-1;                           psem[tsem]:=p;
  end                                     end;
end;                                    end;
```

Os procedimentos `gera`*n* (apresentados a seguir) criam nós com *n*=0, 1, 2 filhos, e são os únicos necessários para o Exemplo 1, uma vez que as regras tem *n*=0, 1 ou 2 nãoterminais do lado direito, mas, claramente, a idéia pode ser estendida para outros valores de *n*. No momento da criação de um nó, *ponteiros* para seus *n* filhos já se encontram na pilha, de onde devem ser retirados na ordem invertida. Após a criação de um nó, o ponteiro correspondente deve ser empilhado.

No Exemplo 1, no caso de uma redução pela regra $E^0 \rightarrow E^1+T$, as árvores correspondentes a E^1 (a ocorrência de `E` do lado direito da regra) e `T` já se encontram construídas, e apontadores para elas estão no topo da pilha. Como a árvore correspondente a E^1 foi construída antes da árvore correspondente a `T` (porque a redução que introduziu E^1 na pilha sintática ocorreu antes da redução que introduziu `T`), o apontador para a árvore de `T` está no topo, e o apontador para a árvore de E^1 está na segunda posição, imediatamente abaixo.

O procedimento `gera2` recupera as duas árvores, através de duas chamadas a `pop`, constrói a árvore correspondente a E^0 (a ocorrência de `E` do lado esquerdo da regra), que tem como raiz um nó novo rotulado pela regra 1, e cujos filhos são as raízes das árvores correspondentes a E^1 e `T`. O código Pascal pode ser um pouco simplificado pelo uso do comando `with`. Os procedimentos `gera1` e `gera0` são semelhantes.

```
procedure gera2(r: integer);
   var p, p1, p2:pt;
begin                          { desempilha ... }
   p2:=pop;                    { ... primeiro, o segundo filho }
   p1:=pop;                    { ... depois, o primeiro filho }
   new(p);                     { cria nó }
   p^.reg:=r;                  { anota regra }
   p^.prim:=p1                 { primeiro filho }
   p1^.ult:=false;             { p1 não é o último filho ... }
   p1^.prox:=p2;               { ... e aponta para o irmão }
   p2^.ult:=true;              { p2 é o último filho ... }
   p2^.prox:=p;                { ... e aponta para o pai }
   push(p);                    { empilha o nó criado }
end; { gera2 }

procedure gera1(r: integer);
   var p, p1:pt;
begin
   p1:=pop;                    { desempilha  o primeiro filho }
   new(p);                     { cria nó }
   p^.reg:=r;                  { anota regra }
   p^.prim:=p1;                { primeiro filho }
   p1^.ult:=true;              { p1 é o último filho ... }
   p1^.prox:=p;                { ... e aponta para o pai }
   push(p);                    { empilha o nó criado }
end; { gera1 }

procedure gera0(r: integer);
   var p:pt;
begin
   new(p);                     { cria nó }
   p^.reg:=r;                  { anota regra }
   p^.prim:=nil;               { sem filhos }
   push(p);                    { empilha o nó criado }
end; { gera0 }
```

Exemplo 1 (continuação): Para a entrada `id*(id+id)`, o parse (direito, invertido, como gerado por um analisador ascendente de G_0) é

6-4-6-4-2-6-4-1-5-3-2.

As ações podem ser associadas às regras, por exemplo, como no código a seguir:

```
case r of
     1: gera2(1);
     2: gera1(2);
     3: gera2(3);
     4: gera1(4);
     5: gera1(5);
     6: gera0(6);
end;
```

A árvore da Fig. 1 foi gerada para a entrada considerada pela seqüência de chamadas

```
gera0(6); gera1(4); gera0(6); gera1(4); gera1(2);
gera0(6); gera1(4); gera2(1); gera1(5); gera2(3);
gera1(2);
```

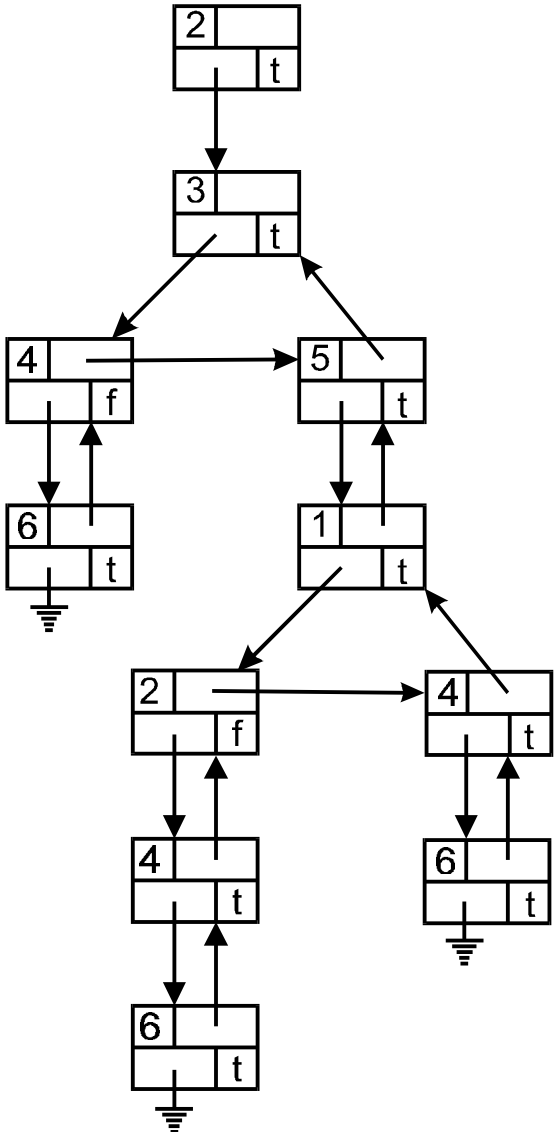


Figura 1: árvore de derivação de `id*(id+id)`

Note que, para a discussão das gramáticas de atributos (seção 4.2 e seguintes), o formato de representação da árvore de derivação é irrelevante. Para uso efetivo de árvores no cálculo de atributos, ou como representação intermediária de programas, a forma de representação é importante, podendo até ser previsto o uso de mais de um formato, em fases distintas do processo de compilação.

4.2. Gramáticas de atributos

Nesta seção vamos introduzir o conceito de gramáticas de atributos. Uma gramática de atributos é composta de duas partes: uma gramática livre de contexto, que descreve uma linguagem, e regras de cálculo, que permitem calcular valores — os atributos — associados aos símbolos terminais e não terminais da gramática que ocorrem na derivação de uma dada cadeia da linguagem. Por exemplo, em uma gramática de uma linguagem de programação, que tem símbolos não terminais `programa` e `comando`, podemos ter uma regra que mostra como calcular o atributo `código` do nãoterminal `programa` em função dos valores dos atributos `código` dos nãoterminais `comando` que compõem o programa (a cadeia de entrada).

O exemplo que vamos apresentar a seguir não tem relação direta com o processo de compilação, mas visa apenas apresentar os conceitos e as definições básicas em um exemplo relativamente pequeno. Apresentaremos posteriormente (no apêndice 4-A) um exemplo completo de geração de código para uma (mini-) linguagem de programação, usando apenas técnicas de gramáticas de atributos. Normalmente, essas técnicas são usadas de forma adaptada em compiladores reais.

Exemplo 2: A gramática de atributos a seguir mostra como calcular o valor de um número binário com parte fracionária (parte "decimal").

```
1. N → I¹ . I²                    2. N → I
     N.v := I¹.v + I².v;               N.v := I.v;
     I¹.p := 0;                        I.p := 0;
     I².p := - I².l;

3. I⁰ → I¹ B                      4. I → B
     I⁰.v := I¹.v + B.v;               I.v := B.v;
     I⁰.l := I¹.l +1;                  I.l := 1
     I¹.p := I⁰.p + 1;                 B.p := I.p;
     B.p := I⁰.p;

5. B → 0                          6. B → 1
     B.v :=0;                          B.v := 2^-B.p;
```

Os diversos componentes da gramática de atributos são discutidos a seguir:

(1). A gramática livre de contexto "subjacente".

```
1. N → I¹ . I²
2. N → I
3. I⁰ → I¹ B
4. I → B
5. B → 0
6. B → 1
```

Note que algumas ocorrências de símbolos na gramática estão diferenciadas por índices superiores (*superscritos*). Isto nos permite fazer referência ao "primeiro `I`" e ao "segundo `I`" da primeira regra, por exemplo. A numeração das regras (*regras sintáticas*) também serve apenas para facilidade de referência.

(2) As regras de cálculo permitem calcular os atributos `v` (valor), `p` (posição), `l` (comprimento). Note que só há referências ao valor de `N` (`N.v`), ao valor e à posição de `B` (`B.v` e `B.p`), mas existem referências aos três atributos, no caso de `I` (`I.v`, `I.p`, `I.l`). Para cada regra sintática, há *regras semânticas* que permitem o cálculo de alguns atributos, em função de outros atributos. Por exemplo, para a regra 1, temos (entre outras) a *regra semântica*

```
N.v := I¹.v + I².v;
```

que indica que

"o valor de `N` é obtido somando os valores do primeiro `I` e do segundo `I`".

Essas regras são criadas por um processo de programação; esta programação, entretanto, não especifica a seqüência em que as regras semânticas devem ser aplicadas. O momento para calcular N.v não é especificado, e a aplicação pode ser feita qualquer momento, *desde que* I^1.v e I^2.v tenham sido calculados antes. Dizemos que N.v *depende* de I^1.v e I^2.v. Isso vale para todas as regras semânticas: as regras podem ser aplicadas em qualquer ordem, desde que os valores usados nas regras tenham sido calculados antes, ou seja, obedecendo a relações de dependência entre atributos.

□

Vamos agora ilustrar o processo de cálculo dos atributos, usando a gramática de atributos do Exemplo 2. A idéia fundamental do processo é a de construir uma árvore de derivação para a cadeia cujos atributos queremos calcular, e percorrer a árvore calculando sempre atributos que só dependem de outros atributos já previamente calculados. O processo se inicia com atributos cujos valores são constantes. A ordem em que os atributos são calculados, exceto pela relação de dependência citada acima, pode ser qualquer.

Exemplo 2: (continuação): A Figura 2 mostra a árvore de derivação da cadeia x=101.011, que será usada como exemplo para o cálculo dos atributos. Em cada nó da árvore de derivação estão indicados o número do nó (para facilitar a referência), o símbolo correspondente, e os valores dos atributos, já calculados.

Descrevemos em seguida uma das possíveis ordens de cálculo dos atributos. A legenda explica o formato da tabela com a memória de cálculo dos atributos.

Legenda	
coluna	conteúdo
n / a	o atributo a vai ser calculado no nó n
n / r	a regra r foi aplicada no nó n; esta regra é usada para o cálculo do atributo na primeira coluna.
nós envolvidos	se a regra r da segunda coluna é $A \rightarrow X_1 X_2 \ldots X_m$, os nós correspondentes a A, X_1, X_2, ... X_m .
cálculo	apresenta a regra semântica aplicada e o resultado.

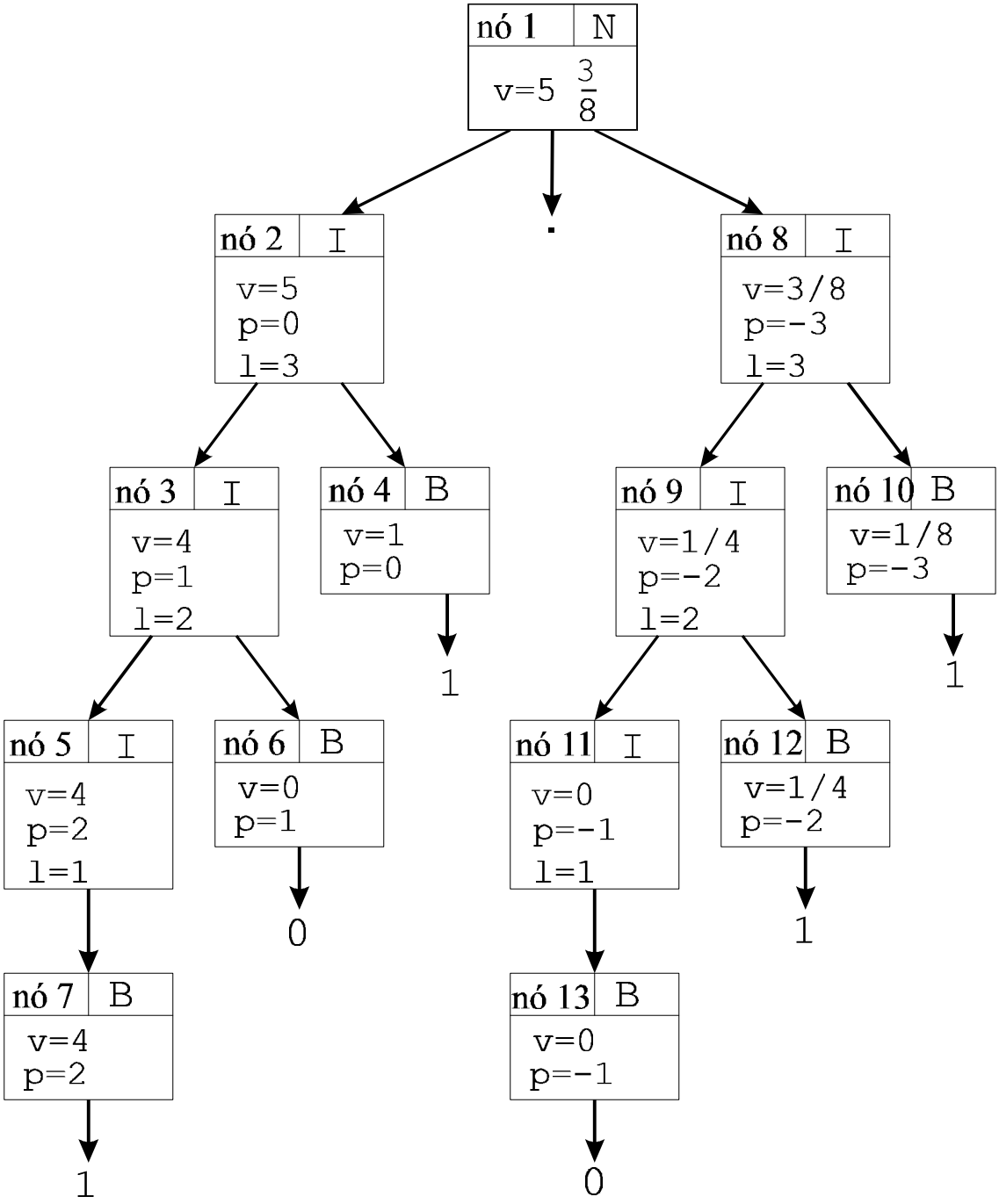


Fig. 2. Árvore de derivação (com atributos) para `101.011`

n / a	n / r	nós envolvidos	cálculo
5/l	5/4	5(I), 7(B)	`I.l := 1`
11/l	11/4	11(I), 13(B)	`I.l := 1`
3/l	3/3	3(I^0), 5(I^1), 6(B)	I^0`.l := `I^1`.l + 1 = 2`
9/l	9/3	9(I^0), 11(I^1), 12(B)	I^0`.l := `I^1`.l + 1 = 2`
2/l	2/3	2(I^0), 3(I^1), 4(B)	I^0`.l := `I^1`.l + 1 = 3`
8/l	8/3	8(I^0), 9(I^1), 10(B)	I^0`.l := `I^1`.l + 1 = 3`
2/p	1/1	1(N), 2(I^1), 8(I^2)	I^1`.p := 0`
8/p	1/1	1(N), 2(I^1), 8(I^2)	I^2`.p := - `I^2`.l = -3`

n / a	n / r	nós envolvidos	cálculo
3/p	2/3	2(I^0), 3(I^1), 4(B)	$I^1.p := I^0.p + 1 = 1$
4/p	2/3	2(I^0), 3(I^1), 4(B)	$B.p := I^0.p = 0$
9/p	8/3	8(I^0), 9(I^1), 10(B)	$I^1.p := I^0.p + 1 = -2$
10/p	8/3	8(I^0), 9(I^1), 10(B)	$B.p := I^0.p = -3$
5/p	3/3	3(I^0), 5(I^1), 6(B)	$I^1.p := I^0.p + 1 = 2$
6/p	3/3	3(I^0), 5(I^1), 6(B)	$B.p := I^0.p = 1$
11/p	9/3	9(I^0), 11(I^1), 12(B)	$I^1.p := I^0.p + 1 = -1$
12/p	9/3	9(I^0), 11(I^1), 12(B)	$B.p := I^0.p = -2$
7/p	5/4	5(I), 7(B)	$B.p := I.p = 2$
13/p	11/4	11(I), 13(B)	$B.p := I.p = -1$
7/v	7/6	7 (B)	$B.v := 2^{B.p} = 4$
13/v	13/5	13 (B)	$B.v := 0$
5/v	5/4	5 (I), 7(B)	$I.v := B.v = 4$
6/v	6/5	6 (B)	$B.v := 0$
11/v	11/4	11 (I), 13(B)	$I.v := B.v = 0$
12/v	12/6	12 (B)	$B.v := 2^{B.p} = 1/4$
3/v	3/3	3(I^0), 5(I^1), 6(B)	$I^1.v := I^1.v + B.v = 4$
4/v	4/6	4(B)	$B.v := 2^{B.p} = 1$
9/v	9/3	9(I^0), 11(I^1), 12(B)	$I^0.v := I^1.v + B.v = 1/4$
10/v	10/6	6(B)	$B.v := 2^{B.p} = 1/8$
2/v	2/3	2(I^0), 3(I^1), 4(B)	$I^0.v := I^1.v + B.v = 5$
8/v	8/3	8(I^0), 9(I^1), 10(B)	$I^0.v := I^1.v + B.v = 3/8$
1/v	1/1	1(N), 2(I^1), 8(I^2)	$N.v := I^1.v + I^2.v = 5\,3/8$

□

A ordem de cálculo usada no exemplo acima não é arbitrária. Primeiro, podemos classificar os atributos em *herdados* e *sintetizados*. A classificação se baseia na posição do símbolo na regra: se temos uma regra $r: A \rightarrow X_1\ X_2 \ldots X_m$, e uma regra semântica associada à regra r permite calcular um atributo x de A ($A.x := \ldots$), o atributo x de A é sintetizado. Se, na regra $r: A \rightarrow X_1\ X_2 \ldots X_m$, calculamos um atributo y de um dos símbolos X_i, ($X_i.y := \ldots$), o atributo y de X_i é herdado. Como o lado esquerdo da regra aparece em cima na árvore de derivação, em geral os atributos sintetizados são calculados em função de outros atributos correspondentes a nós inferiores na árvore, e portanto valores sucessivos do atributo podem ser calculados *subindo* na árvore. Semelhantemente, valores sucessivos de atributos herdados podem ser calculados *descendo* na árvore.

Para maior clareza, vamos fazer algumas restrições nas gramáticas de atributos com que vamos trabalhar aqui.

Primeiro, vamos exigir que cada atributo seja *ou* sintetizado *ou* herdado, isto é, um atributo x de um símbolo X não pode ser calculado uma vez quando o símbolo X se encontra do lado esquerdo de uma regra, e outra vez quando X se encontra do lado direito de outra regra. Entretanto, observamos que os atributos são associados aos símbolos, e podemos ter um atributo herdado $X.x$, e outro atributo $Y.x$ sintetizado.

Segundo, vamos exigir que, em todas as regras, sejam apresentadas regras semânticas para calcular todos os atributos que *podem* ser calculados: numa regra $A \rightarrow X_1 X_2 \ldots X_m$, devem estar presentes regras semânticas para cálculo de todos os atributos sintetizados de A, e de todos os atributos herdados de todos os X_i.

Uma gramática que satisfaz essas restrições é chamada uma *gramática normal*. Todas as gramáticas de atributos aqui consideradas serão gramáticas normais.

Exemplo 2 (continuação): A gramática de atributos deste exemplo é uma gramática normal. Os atributos N.v, I.v, I.l, B.v são sintetizados e os atributos I.p e B.p são herdados. Por exemplo, na regra 3 estão calculados os atributos sintetizados de I^0 (I^0.v e I^0.l), os atributos herdados de I^1 (I^1.p), e de B (B.p).

```
3. I⁰ → I¹ B
      I⁰.v := I¹.v + B.v;
      I⁰.l := I¹.l +1;
      I¹.p := I⁰.p + 1;
      B.p := I⁰.p;
```

A ordem de cálculo descrita na tabela tem três fases:

- l (sintetizado) é calculado subindo;
- p (herdado) é calculado descendo;
- v (sintetizado) é calculado subindo.

A ordem l-p-v é usada porque nenhum valor de l depende de algum valor de outro atributo, mas valores de p podem depender dos valores de l (na regra 1, temos I^2.p := -I^2.l) e valores de v podem depender dos valores de p (na regra 6, temos B.v := $2^{B.p}$).

□

Nem sempre é possível fazer uma análise simples e determinar uma ordem de cálculo para os atributos, como foi feito no exemplo 2. Um ponto a observar é que mesmo em uma gramática normal, pode não ser possível calcular todos os atributos, para algumas cadeias de entrada. Isso acontece quando existe *circularidade* na definição: para calcular um atributo A.x, precisamos, direta ou indiretamente, do valor de um atributo B.y; para calcular B.y, precisamos, direta ou indiretamente, do valor de A.x. Esquematicamente,

```
A.x := f1( ..., B.y, ... );
...
B.y := f2( ..., A.x, ... );
```

As condições para que uma gramática seja normal apenas garantem a presença de regras

```
A.x := ... ;
```

e

```
B.y := ... ;
```

mas não garantem a *não-circularidade* da gramática.

Dizemos que uma gramática de atributos é *circular* se para alguma entrada existe um ciclo de dependências como o exemplificado acima. Para fins deste trabalho,

as gramáticas circulares serão consideradas inúteis, mas vale a pena observar que em outros contextos, podemos considerar

$$\begin{cases} A.x = \ldots . ; \\ \qquad \ldots \\ B.y = \ldots . ; \end{cases}$$

como um sistema de equações simultâneas a serem resolvidas, que pode ter zero, uma ou mais soluções.

Existem algoritmos para a verificação da não-circularidade de uma gramática de atributos, mas são algoritmos lentos, de tempo exponencial no tamanho da gramática. Essa é uma das razões pela qual se considera mais importante uma propriedade mais forte (mais restritiva) do que a não-circularidade: a *não-circularidade absoluta*. A não-circularidade absoluta é bem mais fácil de testar que a circularidade.

Simplificadamente, diz-se que uma gramática é *absolutamente não circular* (*ANC*) se existe uma *receita* de cálculo dos atributos que funciona para qualquer entrada, ou seja, que não depende de alguma característica da árvore de derivação da entrada. Essa *receita* toma a forma de um procedimento que avalia os atributos em uma árvore de derivação qualquer. Esse procedimento é recursivo, e uma chamada na raiz da árvore de derivação em princípio pode levar a várias chamadas em todos os nós da árvore. Esses procedimentos são conhecidos como *avaliadores recursivos*, ou, às vezes como *tree-walking automata*: autômatos que caminham na árvore de derivação.

A classe das gramáticas ANC inclui todas as gramáticas que nos interessam na prática, uma vez que é mais fácil construir um compilador usando um procedimento ("receita") que se aplica a todos os programas fonte, do que deduzir, para cada programa fonte, a ordem de cálculo apropriada, após a construção da árvore de derivação correspondente.

As gramáticas ANC e os avaliadores recursivos são examinados na próxima seção.

4.3 Gramáticas absolutamente não circulares e avaliadores recursivos.

A análise simplificada que vamos fazer aqui se baseia na construção do avaliador recursivo para a gramática: se essa construção é possível, a gramática de atributos é ANC.

A idéia fundamental é a construção de vários grafos que indiquem as dependências entre os atributos. Em princípio, uma aresta de `A.x` para `B.y`, em um desses grafos indica que o valor de `A.x` é usado no cálculo do valor de `B.y`. Na realidade, como queremos uma receita que funcione para todos os casos, vamos introduzir essa aresta sempre que *existir a possibilidade* de que `A.x` seja necessário para calcular `B.y`.

Os grafos a serem construídos são:

1. para cada regra *r*, um grafo de dependências *D(r)* entre as ocorrências de atributos na regra. Esse grafo é chamado *Grafo de Dependências da Regra r*, e tem como nós todas as ocorrências de todos os atributos dos símbolos da regra. Suas arestas refletem as dependências existentes naquela regra: se tivermos
   `B.y:=f(..., A.x, ...)`,
   teremos uma aresta de `A.x` para `B.y`.

2. para cada regra *r*, um *Grafo de Dependências Induzidas da Regra, DI(r)*. Este grafo tem os mesmos nós do grafo D(r), mas, além das arestas de D(r), tem arestas que refletem o efeito das dependências decorrentes de aplicações de outras regras aos símbolos não terminais do lado direito da regra *r*. (Ver algoritmo a seguir.)

3. para cada símbolo `A` da gramática, um *Grafo de Dependências Induzidas DI(A)* para o símbolo. Esse grafo tem um nó para cada atributo de `A`, e suas arestas refletem as dependências induzidas entre atributos de `A` presentes nos grafos DI(r), para as regras com `A` do lado esquerdo. Para um símbolo (terminal ou nãoterminal) que não tem atributos, o grafo é vazio, e não precisa ser construído.

Vamos apresentar o algoritmo para construção desses grafos em paralelo com a construção dos grafos no caso do exemplo 2.

Construção de D(r), para todas as regras r.

Seja a regra r: $X_0 \rightarrow X_1 X_2 \ldots X_m$. Para cada ocorrência de símbolo X_i, e para cada atributo `a` de X_i, acrescente um nó X_i`.a` em D(r). Para cada regra semântica X_i`.a:=f(...` X_j`.b, ...)`, inclua uma aresta de cada X_j`.b` que ocorre do lado direito para X_i`.a`.

Exemplo 2 (continuação): Considere a regra `1:` $N \rightarrow I^1.I^2$. Os nós de D(1) são $N.v$, $I^1.v$, $I^1.p$, $I^1.l$, $I^2.v$, $I^2.p$, $I^2.l$. As regras semânticas são

```
N.v := I¹.v + I².v;
I¹.p := 0;
I¹.p := - I².l;
```

e, portanto, as arestas são

de $I^1.v$ para $N.v$
de $I^2.v$ para $N.v$
de $I^2.l$ para $I^2.p$

O grafo D(1) está indicado na Fig. 3, juntamente com os demais grafos D(2), ... D(6), que podem ser obtidos de forma semelhante. Para maior clareza, os nós associados a atributos de uma mesma ocorrência de nãoterminal foram reunidos.

Construção dos grafos DI(r) e DI(A)

A construção dos grafos DI(*r*) e DI(`A`), para as diversas regras r e para os diversos símbolos `A` é feita de forma conjunta, pela repetição dos dois passos α e β a

seguir, *até que nenhuma aresta possa ser acrescentada a qualquer dos grafos*. Inicialmente, DI(r) tem os mesmos nós e arestas de D(r), e DI(A) tem um nó para cada atributo de A, e nenhuma aresta.

α. Para cada grafo DI(r), onde r é $X_0 \rightarrow X_1 X_2 \ldots X_m$, e X_0 é uma ocorrência do nãoterminal A, identifique todos os nós $X_0.a$ e $X_0.b$ entre os quais existe um caminho (possivelmente passando por atributos de outros símbolos da regra). Para todos esses pares, acrescente uma aresta entre A.a e A.b em DI(A) (se a aresta não existir).

β. Para cada aresta de A.a para A.b de um grafo DI(A), acrescente arestas de $X_i.a$ para $X_i.b$, em todos os grafos DI(r) correspondentes a regras r em que A tem uma ocorrência X_i do lado direito da regra (quando essas arestas ainda não existirem).

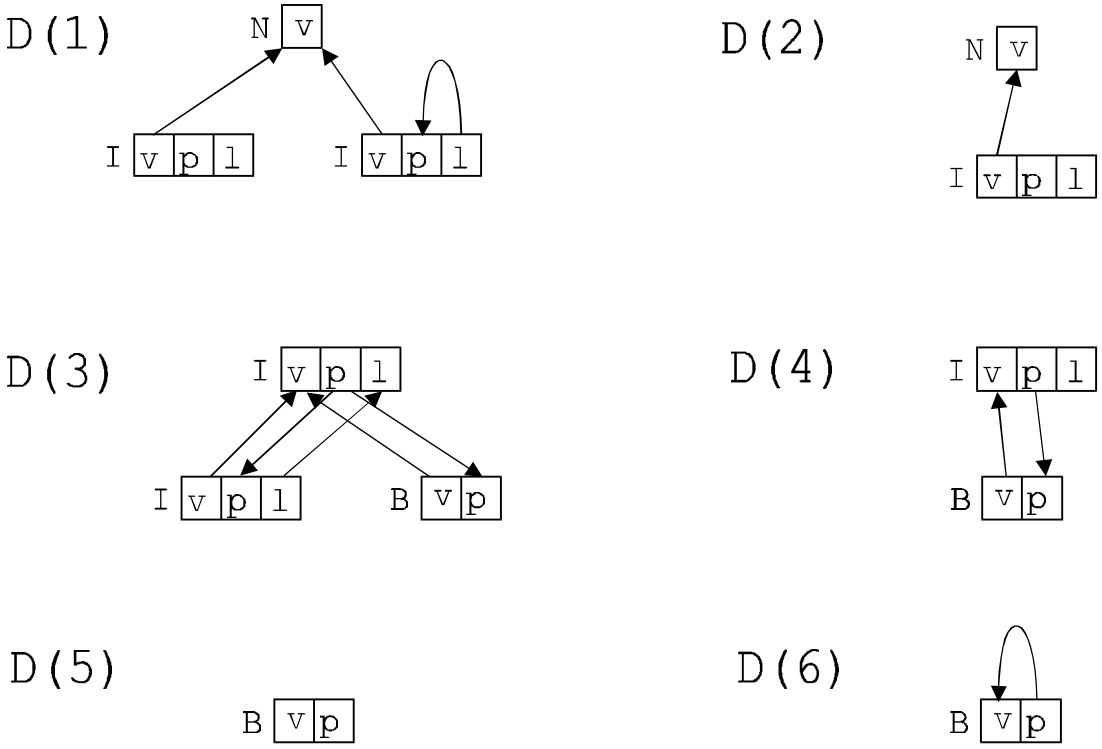


Fig. 3. Grafos D(*r*) para o Exemplo 2.

□

Exemplo 2 (continuação): Inicialmente, verificamos que não existem caminhos entre atributos de N nas regras 1 e 2 (mesmo porque N só tem um atributo), nem entre atributos de I nas regras 3 e 4, ou atributos de B na regra 5. Em DI(6) existe um caminho entre B.p e B.v. Por essa razão, acrescentamos (passo α) uma aresta de B.p para B.v em DI(B) (ver Fig. 4). Em consequência dessa inclusão acrescentamos (passo β) arestas de B.p para B.v nos grafos DI(3) e DI(4). Verificamos agora que há caminhos novos de I^0.p para I^0.v, em DI(3), e de I.p para I.v em DI(4), e, por essa razão, acrescentamos (passo α) uma aresta de I.p para I.v em DI(I). Pelo passo β, acrescentamos arestas de I^1.p para I^1.v em DI(3), de I^1.p para I^1.v e de I^2.p para I^2.v em DI(1), e de I.p para I.v em DI(2). Como não há nenhum

caminho novo, o processo termina. Os grafos resultantes são mostrados na Fig. 4. As arestas novas dos grafos DI(r), ou seja, as que não pertencem a D(r), estão indicadas por linhas interrompidas.

□

Construídos os grafos DI(*r*), para todas as regras *r,* podemos verificar se algum desses grafos contém um ciclo. Dizemos que a gramática é *absolutamente não circular* (*ANC*) se não existem ciclos nesses grafos. Na prática, isto quer dizer *(1)* que a gramática não é circular, e *(2)* que é possível, para cada regra *r*, determinar (pelo menos) uma ordem de cálculo dos atributos que satisfaz todas as dependências, tanto as presentes na regra *r* (as arestas provenientes de D(*r*), quanto as provenientes das aplicações de outras regras aos símbolos que ficam do lado direito de *r* (as arestas tracejadas). Vamos a seguir mostrar como construir um procedimento que avalia os atributos de acordo com uma ordem escolhida.

No caso contrário de serem encontrados um ou mais ciclos em alguns dos grafos DI(*r*), a gramática é dita *não absolutamente não circular*. Isto pode acontecer por uma de duas razões: *(1)* a gramática é circular, *ou* *(2)* a gramática não é circular, mas apenas *parece* circular, porque cada ciclo encontrado é composto de arestas provenientes de árvores de derivação diferentes, e o ciclo não existe completo em nenhuma das árvores de derivação. Apresentaremos posteriormente (Exemplo 3) uma gramática de atributos que não é circular, mas não passa no teste de não-circularidade absoluta. Para essas gramáticas (não circulares, mas não ANC), seria necessário calcular os atributos cada vez (para cada entrada) usando uma ordem de cálculo diferente. Essas gramáticas não são úteis como instrumentos auxiliares na construção de compiladores. Na prática, repetimos, só nos vão interessar as gramáticas ANC.

Exemplo 2 (continuação): Este exemplo satisfaz a condição de ANC, uma vez que não há ciclos em D(1), ..., D(6). Vamos ver para cada regra as ordens de cálculo possíveis.

Regra 1: há três atributos que podem ser calculados inicialmente (nós sem arestas entrando). Isso pode ser visto com mais facilidade se o grafo for "desenrolado", e desenhado de maneira diferente, como na Figura 4 (III).

São os atributos $I^1.l$, $I^2.l$, $I^1.p$. Supondo que o valor que realmente nos interessa é o valor de `v` na raiz, o atributo $I^1.l$ é inútil, uma vez que nenhum outro atributo depende dele. (Claro, se quisermos calculá-lo, por alguma razão especial, o cálculo poderá ser feito de forma inteiramente semelhante à de $I^2.l$.) Entre as várias ordens de cálculo possíveis vamos escolher a seguinte:

$I^1.p,\ I^1.v,\ I^2.l,\ I^2.p,\ I^2.v,\ N.v$

Note que os atributos *sintetizados* $I^1.v$, $I^2.l$, e $I^2.v$ só podem ser calculados quando o símbolo `I` correspondente está à esquerda (regras 3 e 4). Portanto é necessário descer aos nós I^1 e I^2 (e possivelmente aos descendentes destes). O nó I^2 precisa ser visitado duas vezes: na primeira, $I^2.l$ é calculado; só na segunda, depois do cálculo de $I^2.p$, é que $I^2.v$ pode ser obtido. Os atributos sintetizados dos símbolos do lado direito só podem ser calculados em visitas aos nós-filhos

correspondentes; entre essas visitas pode ser necessário subir para obter valores de atributos herdados.

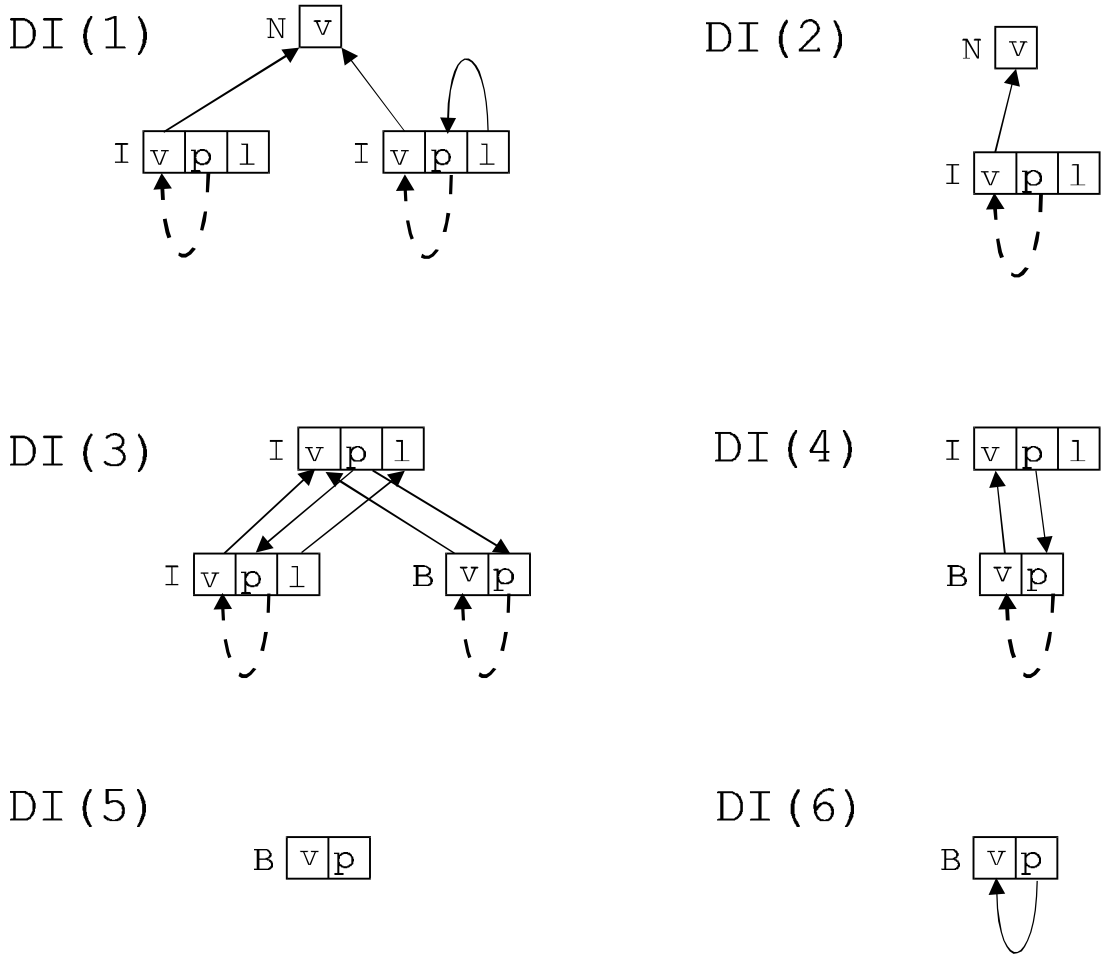


Fig. 4(I). Grafos de dependências induzidas para o Exemplo 2

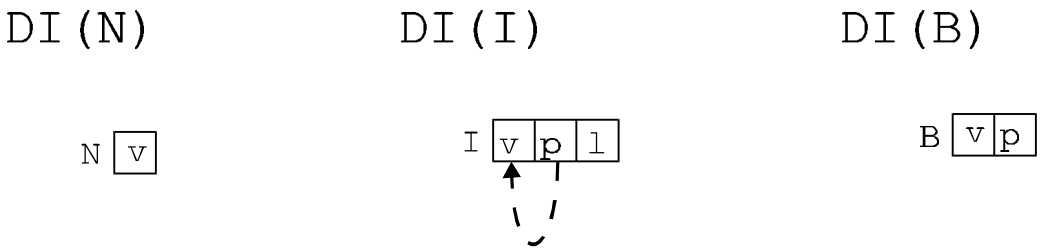


Fig. 4 (II). Grafos de dependências induzidas para o Exemplo 2

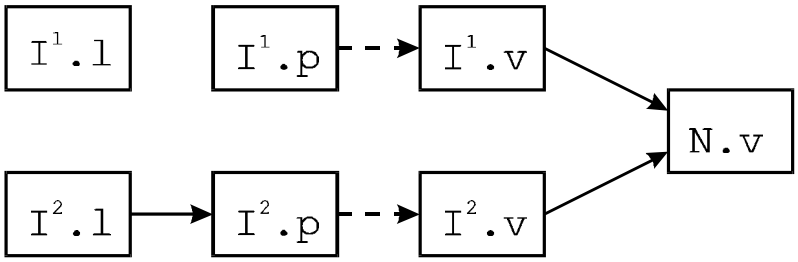


Fig. 4 (III)- Outra maneira de desenhar `DI(1)`

Regra 2: Para a regra 2, só há uma ordem de cálculo possível:

```
I.l, I.p, I.v, N.v
```

De forma semelhante, o nó `I` deve ser visitado duas vezes, uma para cálculo de `I.l` e outra, depois do cálculo de `I.p`, para cálculo de `I.v`.

Regra 3: Para a regra 3, é necessário considerar dois casos diferentes, correspondentes às duas visitas mencionadas na análise das regras 1 e 2. Se nenhum atributo de I^0 é conhecido, só podemos calcular I^1`.l` (visita a I^1) e I^0`.l`; entretanto, se I^0`.p` é conhecido, é possível calcular I^1`.p`, I^1`.v` (visita a I^1), `B.p`, `B.v` (visita a `B`) e I^0`.v`.

Regra 4: Como para a regra 3, temos dois casos a considerar. Se nenhum atributo de `I` é conhecido, só podemos calcular `I.l`; entretanto, se `I.p` é conhecido, é possível calcular `B.p`, `B.v` (visita a `B`) e `I.v`.

Regras 5, 6: Note que nas visitas a nós `B`, sempre o atributo `B.p` será conhecido, e sempre o atributo `B.v` será calculado. No caso da regra 5, o atributo `B.p` não será utilizado.

□

A implementação do esquema ordenado de cálculo dos atributos pode ser feita através de um *avaliador recursivo*, implementado através de um procedimento `eval` (de *evaluate*, avaliar) correspondente a uma visita a um filho, e que é chamado a primeira vez na raiz da árvore. Em princípio, cada chamada de `eval` em um nó calcula todos os atributos que já podem ser calculados naquele nó, se necessário fazendo chamadas (recursivas) a `eval` nos filhos do nó considerado. A efetiva implementação de `eval` pode levar em consideração muitas possibilidades de economia de tempo ou de espaço em memória; em nosso exemplo, procuramos apenas mostrar que `eval` pode ser escrito de forma mecânica a partir dos grafos DI(*r*) de uma gramática ANC.

Dada uma gramática de atributos ANC, o procedimento `eval` é construído da seguinte maneira:

- `eval` recebe dois parâmetros: o primeiro é um ponteiro para o nó de trabalho; o segundo é um *estado*, que descreve a informação disponível naquele nó, para aquela chamada. Em princípio, o valor do estado deve ser o conjunto dos atributos já calculados, mas também podem ser usados números inteiros (1 para a primeira visita, 2 para a segunda, etc.).
- `eval` tem um formato de `case`, em que cada caso corresponde a uma regra. Se há vários valores possíveis para o estado cada caso deve ser subdividido, para levar em consideração os diferentes valores do estado.

Para cada regra e para cada valor do estado, deve ser escolhida uma ordem de cálculo possível. Os comandos correspondentes calculam os atributos, obedecendo à ordem de cálculo escolhida, e podem ser de dois tipos:

- cálculo local de atributos (atributos herdados de símbolos do lado direito, e atributos sintetizados do símbolo do lado esquerdo)
  Neste caso, o atributo é calculado imediatamente, em função dos demais atributos presentes, no nó e em seus filhos.

- visita para cálculo de atributos (atributos sintetizados dos símbolos do lado direito)
  Neste caso, uma chamada de `eval` é feita, tendo como argumentos o ponteiro para o nó cujo atributo se deseja calcular, e o estado correspondente a esse nó.

Duas alterações podem ser feitas sem dificuldade: atributos inúteis não precisam ser calculados, e dois ou mais atributos podem ser calculados em uma única visita.

Exemplo 2 (continuação): Vamos supor o seguinte formato para os nós da árvore de derivação:

```
type pt = ^no;
     no = record
          reg:integer;     { valor da regra aplicada neste nó }
          v:real;          { atributo sintetizado valor }
          p:integer;       { atributo herdado posição }
          l:integer;       { atributo sintetizado comprimento }
          f1, f2:pt;       { ponteiros para o primeiro filho }
                           { (ou filho único) e para o segundo }
     end;
```

Os valores do estado serão representados por 1 e 2 para a primeira e a segunda visitas aos nós I. Os nós `N` e `B` só serão visitados uma vez, e portanto para as regras 1, 2, 5 e 6, o estado será sempre 1, e não há necessidade de testar o valor de `s`. Note também que na regra 1, a única visita prevista a I^1 é a *segunda*.

```
procedure eval(q:pt; s:integer);
begin
  case q^.reg of
    1: begin
         q^.f1^.p:=0;                  { I1.p }
         eval(q^.f1, 2);               { I1.v }
         eval(q^.f2, 1);               { I2.l }
         q^.f2^.p := - q^.f2^.l;       { I2.p }
         eval(q^.f2^, 2);              { I2.v }
         q^.v := q^.f1^.v + q^.f2^.v; { N.v }
       end;
    2: begin
         q^.f1^.p:=0;                  { I.p }
         eval(q^.f1, 2);               { I.v }
         q^.v := q^.f1^.v              { N.v }
       end;
    3: if s=1 then begin               { primeira visita }
         eval(q^.f1, 1);               { I1.l }
         q^.l := q^.f1^.l + 1;         { I0.l }
       end else begin                  { segunda visita }
         q^.f1^.p := q^.p + 1;         { I1.p }
         eval(q^.f1, 2);               { I1.v }
         q^.f2^.p := q^.p;             { B.p }
         eval(q^.f2, 1);               { B.v }
         q^.v := q^.f1^.v+q^.f2^.v;  { I0.v }
       end;
```

```
    4: if s=1 then                      { primeira visita }
         q^.l := 1                      { I.l }
       else begin                       { segunda visita }
         q^.f1^.p := q^.p;              { B.p }
         eval(q^.f1, 1);                { B.v }
         q^.v := q^.f1^.v;              { I.v }
       end;
    5: begin
         q^.v:= 0;                      { B.v }
       end;
    6: begin
         q^.v:= exp(2, q^.p);           { B.v }
       end;
  end;
end;
```

Fica como exercício a simulação do procedimento `eval` para a entrada `101.011`.

□

Exercício 1: Considere a gramática de atributos abaixo.

1. $S \rightarrow A\ B$
   $S.ok := A.ok \text{ and } B.ok;$
   $A.s := B.s;$
   $B.n := A.n;$
2. $A^0 \rightarrow a\ A^1$
   $A^0.ok := A^1.ok;$
   $A^0.n := A^1.n + 1;$
   $A^1.s := A^0.s;$
3. $A \rightarrow 0$
   $A.ok := A.s=0;$
   $A.n := 0;$
4. $A \rightarrow 1$
   $A.ok := A.s=1;$
   $A.n := 0;$
5. $B^0 \rightarrow b\ B^1$
   $B^0.ok := B^1.ok$
   $B^0.s := B^1.s;$
   $B^1.n := B^0.n;$
6. $B \rightarrow 0$
   $B.ok := B.n=0;$
   $B.s := 0;$
7. $B \rightarrow 1$
   $B.ok := B.n=0;$
   $B.s := 1;$

1. Verifique que a gramática é normal, indicando os atributos herdados e sintetizados.
2. Verifique a gramática é absolutamente não circular, e construa um avaliador recursivo para seus atributos.

□

O exemplo seguinte mostra uma gramática que não é circular, mas também não é absolutamente não circular.

Exemplo 3: Considere a gramática de atributos abaixo:

1. $S \rightarrow A$
   $S.s := A.a;$
   $A.b := A.a;$
   $A.d := A.c;$
2. $A \rightarrow B$
   $A.a := A.e;$
   $A.c := A.g;$
   $B.f := A.b;$
   $B.h := A.d;$
3. $B \rightarrow 0$
   $B.e := B.h;$
   $B.g := 0;$
4. $B \rightarrow 1$
   $B.e := 1;$
   $B.g := B.f;$

A linguagem desta gramática é composta pelas duas cadeias `0` e `1`, e o cálculo dos atributos nestes dois casos não oferece problemas: a gramática não é circular,

como se pode ver nas duas árvores de derivação, correspondentes às duas cadeias, apresentadas na Fig. 5. Pelos grafos de dependências induzidas (Fig. 6 e 7), entretanto, podemos ver que a gramática não é absolutamente não circular: existe um ciclo (`A.a-A.b-A.c-A.d-A.a`) em DI(1). Dessas, a aresta `A.b-A.c` indica a possível dependência de `A.c` em `A.b`, decorrente da dependência de `B.g` em `B.f` (regra 3), e a aresta `A.d-A.a` indica a possível dependência de `A.c` em `A.b`, decorrente da dependência de `B.e` em `B.h` (regra 4). Note que em nenhuma árvore de derivação as duas arestas podem ocorrer simultaneamente, uma vez que as regras 3 e 4 não podem ser usadas ao mesmo tempo.

O ponto importante é que para gramáticas como esta, não é possível escrever um avaliador recursivo. Podemos ver que as ordens de cálculo dos atributos são completamente diferentes, para as duas entradas possíveis.

□

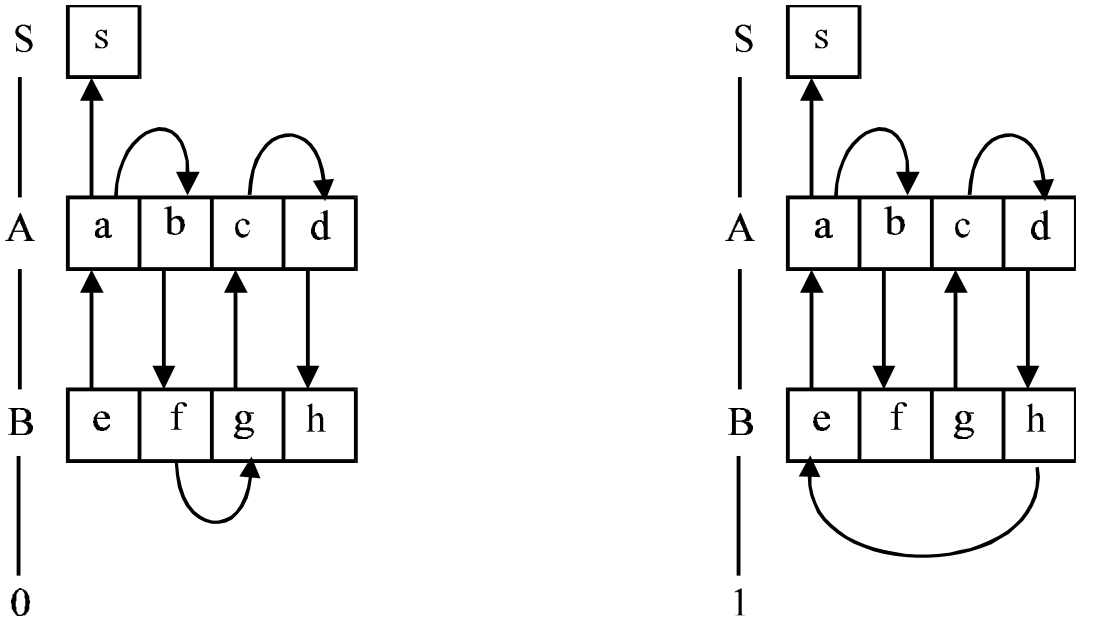


Fig. 5. - Árvores de derivação e dependências (Exemplo 3).

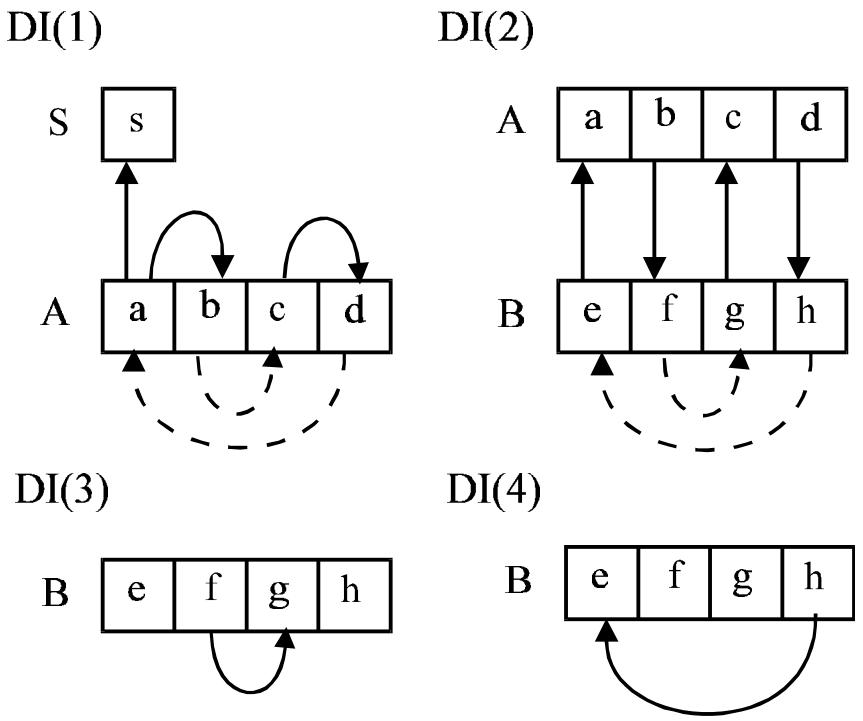


Fig. 6 - Grafos de dependências induzidas das regras *1-4*

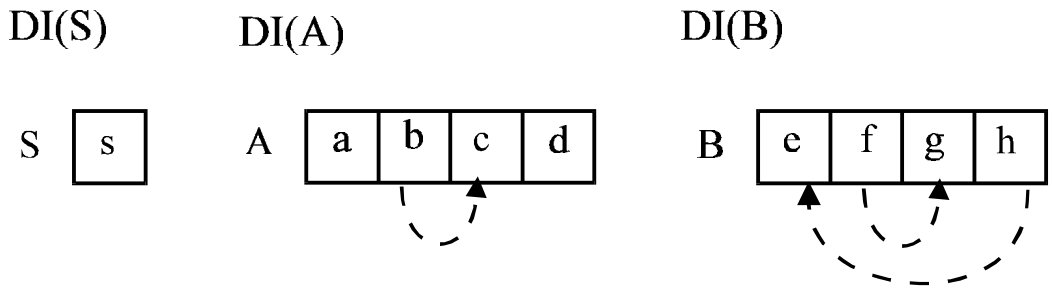


Fig. 7 - Grafos de dependências induzidas para S**,** A**,** B

4.4. Cálculo de atributos da esquerda para a direita.

O método de cálculo apresentado na seção anterior é bastante geral, uma vez que se aplica a qualquer gramática absolutamente não circular. Entretanto, embora a geração da árvore de derivação seja simples, como visto na seção 1, e a construção do avaliador possa ser feita de forma automática, a construção da árvore de derivação é trabalhosa e ocupa espaço de memória considerável, e por essas razões é evitada sempre que possível. Na prática, situações como as dos Exemplo 2 e 3 (em que é necessário fazer mais de uma visita à mesma sub-árvore da árvore de derivação) são relativamente raras, e em muitos casos é suficiente fazer uma visita a cada sub-árvore sempre em ordem, recursivamente, da esquerda para a direita. Normalmente, o projeto de uma linguagem de programação procura fazer com que as informações sejam incluídas no código fonte antes do ponto em que se tornam necessárias. Por exemplo, em geral as declarações precedem os comandos, uma vez que as informações fornecidas pelo programador nas declarações são usadas, posteriormente, para a compilação dos comandos.

Supondo algumas restrições na gramática de atributos, é possível, fazendo apenas uma visita à árvore de derivação, da esquerda para a direita, guardar os valores dos atributos em uma pilha (*pilha de atributos, pilha semântica*). Como acontece com o processo de análise sintática, a árvore de derivação é usada apenas como referência, na descrição da ordem em que as operações são realizadas, sem ser efetivamente construída. Esse esquema de *avaliação de atributos da esquerda para a direita*[1] permite prever a ordem em que os atributos são calculados, e permite a liberação imediata do espaço ocupado por atributos que não serão mais utilizados. Como uma vantagem adicional, a verificação da aplicabilidade deste esquema é mais simples do que a verificação da nãocircularidade absoluta: basta testar cada regra separadamente.

Suponha uma regra *r*: $A \rightarrow X_1 X_2 \ldots X_m$. As condições a serem verificadas para a regra *r* são:

- os atributos (herdados) de X_i podem depender de
  - todos os atributos herdados de A.
  - todos os atributos herdados e sintetizados de X_j, para $j=1,\ldots,i-1$.
  - todos os atributos herdados de X_i, já considerados anteriormente.
- os atributos (sintetizados) de A podem depender de
  - todos os atributos herdados de A.
  - todos os atributos herdados e sintetizados de todos os X_j, para $j=1,\ldots, n$.
  - todos os atributos sintetizados de X_i, já considerados anteriormente.

Podemos descrever a situação de outra forma, dizendo que a ordem de cálculo é a seguinte:

- atributos herdados de A (antes da visita ao nó onde a regra *r* se aplica).
- atributos herdados de X_1 (antes da visita à sub-árvore de X_1, calculados pela regra *r*).
- atributos sintetizados de X_1 (durante a visita à sub-árvore de X_1, calculados por alguma regra de X_1)
- atributos herdados de X_2 (antes da visita à sub-árvore de X_2, calculados pela regra *r*).
- atributos sintetizados de X_2 (durante a visita à sub-árvore de X_2, calculados por alguma regra de X_2)

....

- atributos herdados de X_n (antes da visita à sub-árvore de X_n, calculados pela regra *r*).
- atributos sintetizados de X_n (durante a visita à sub-árvore de X_n, por alguma regra de X_n)
- atributos sintetizados de A, calculados pela regra *r*

Cada atributo pode depender de qualquer outro atributo calculado anteriormente. Note que após o cálculo dos atributos sintetizados de A, os valores de todos os atributos de X_1, ..., X_n não poderão mais ser utilizados, e o espaço ocupado por eles pode ser liberado.

[1] *left-attributed* ou *l-attributed translations*, segundo Aho, Sethi e Ullman [Dragão].

A alocação de espaço para os atributos herdados de um símbolo X_i é feita (no topo da pilha) imediatamente antes do cálculo desses atributos. A alocação de espaço para os atributos sintetizados de A é feita após a remoção de todos os atributos dos símbolos X_i do lado direito da regra. (Note que os atributos dos símbolos X_i podem ser usados para o cálculo dos atributos sintetizados de A, e devem, portanto ser guardados até que esse cálculo seja efetuado.)

Se um símbolo terminal tem atributos sintetizados, considera-se que esses atributos são pré-calculados, ou seja, calculados de alguma maneira especificada fora da gramática de atributos; o mesmo acontece com atributos herdados do símbolo inicial. Na prática, os atributos sintetizados de terminais são informações obtidas pelo analisador léxico, que pode inseri-los no nó correspondente da árvore de derivação, ou na posição correta da pilha de atributos.

A visita à árvore pode ser simulada com facilidade por um analisador descendente, bastando para isso acrescentar algumas informações à pilha (sintática) do analisador. Toda vez que um nãoterminal é encontrado no topo, o analisador descendente escolhe uma regra daquele nãoterminal, e empilha os símbolos correspondentes ao lado direito dessa regra. O que é feito é acrescentar aos símbolos do lado direito da regra símbolos novos, associados às ações a serem executadas, e as ações correspondentes são executadas cada vez que um símbolo correspondente a uma ação é encontrado no topo da pilha sintática. Essas ações fazem então o tratamento correto da pilha de atributos.

Na regra *r*: $A \rightarrow X_1\ X_2\ ...\ X_n$ podemos indicar as ações a serem realizadas: a_1, a_2, ... a_n, b, entre os símbolos do lado direito:

r: $A \rightarrow a_1\ X_1\ a_2\ X_2\ ...\ a_n\ X_n\ b$. [2]

As ações a_i alocam espaço na pilha para todos os atributos de X_i, e calculam seus atributos herdados; a ação b calcula os atributos sintetizados de A e desaloca o espaço ocupado pelos atributos de X_1, ..., X_n.

O exemplo a seguir mostra como construir um avaliador de atributos para uma gramática cujos atributos podem ser avaliados da esquerda para a direita.

Exemplo 4: Considere a gramática de atributos abaixo.

```
1.   S → A B C
     B.m := A.n;
     C.m := B.n;
     S.ok := C.n = 0;
2.   A⁰ → a A¹
     A⁰.n := A¹.n + 1;
3.   A → ε
     A.n := 0;
4.   B⁰ → b B¹
     B⁰.n := B¹.n - 1;
     B¹.m := B⁰.m;
5.   B → ε
     B.n := B.m;
6.   C⁰ → c C¹
     C⁰.n := C¹.n - 1;
     C¹.m := C⁰.m;
7.   C → ε
     C.n:=C.m;
```

O atributo S.ok terá o valor TRUE quando o número de a's for a soma do número de b's com o número de c's. Os lados direitos das regras são alterados para

[2] A notação utilizada em [Dragão] insere diretamente o código das ações entre os símbolos da regra.

1. S → A r_1 B r_2 C r_3
2. A^0 → a A^1 r_4
3. A → r_5
4. B^0 → b r_6 B^1 r_7
5. B → r_8
6. C^0 → c r_9 C^1 r_{10}
7. C → r_{11}

Para implementar as ações, vamos supor disponíveis três procedimentos:

`push(i)`	empilha o valor `i`
`pop(i)`	retira `i` elementos da pilha
`top(i)`	devolve um elemento da pilha, contando a partir do topo. (`top(0)` devolve o elemento do topo.)

As ações a serem executadas são as seguintes:

```
r1:  push(top(0));     { empilha B.m = A.n }
r2:  push(top(0));     { empilha C.m = B.n }
r3:  t:=top(0)=0;
     pop(5);           { desempilha A.n, B.m, B.n, C.m, C.n }
     push(t);          { empilha S.ok =C.n=0 }
r4:  t:=top(0)+1;
     pop(1);           { desempilha A¹.n }
     push(t);          { empilha A⁰.n = A¹.n +1 }
r5:  push(0);          { empilha A.n =0 }
r6:  push(top(0));     { empilha B¹.m = B⁰.m }
r7:  t:=top(0)-1;
     pop(2);           { desempilha B¹.m, B¹.n }
     push(t);          { empilha B⁰.n = B¹.n -1 }
r8:  push(top(0));     { empilha B.n = B.m }
r9:  push(top(0));     { empilha C¹.m = C⁰.m }
r10: t:=top(0)-1;
     pop(2);           { desempilha C¹.m, C¹.n }
     push(t)           { empilha C⁰.n = C¹.n -1 }
r11: push(top(0);      { empilha C.n }
```

Note que algumas ações desnecessárias não foram incluídas. São ações antes de símbolos (terminais ou nãoterminais) que não tem atributos (herdados) para serem calculados na ocasião. Pelas características do exemplo, todos os acessos à pilha de atributos foram para obter o valor do atributo do topo da pilha.

Para exemplificar o cálculo dos atributos, vamos usar a tabela da página seguinte, que mostra o andamento em paralelo do processo de análise, através do conteúdo da pilha da análise sintática, e andamento do processo de cálculo dos atributos, através do conteúdo da pilha de atributos. Para maior clareza, os nomes e os símbolos correspondentes aos atributos estão também indicados, na pilha de atributos, junto a seus valores. A cadeia de entrada considerada é `aaaabbcc`.

A construção de um analisador descendente para a gramática dada e a construção do restante do avaliador de atributos fica como exercício.

Entrada	Pilha sintática	Pilha de atributos
aaaabbcc	S	
Aaaabbcc	$Ar_1Br_2Cr_3$	
aaaabbcc	$aAr_4r_1Br_2Cr_3$	
aaabbcc	$Ar_4r_1Br_2Cr_3$	
aaabbcc	$aAr_4r_4r_1Br_2Cr_3$	
aabbcc	$Ar_4r_4r_1Br_2Cr_3$	
aabbcc	$aAr_4r_4r_4r_1Br_2Cr_3$	
abbcc	$Ar_4r_4r_4r_1Br_2Cr_3$	
abbcc	$aAr_4r_4r_4r_4r_1Br_2Cr_3$	
bbcc	$Ar_4r_4r_4r_4r_1Br_2Cr_3$	
bbcc	$r_5r_4r_4r_4r_4r_1Br_2Cr_3$	
bbcc	$r_4r_4r_4r_4r_1Br_2Cr_3$	A.n=0
bbcc	$r_4r_4r_4r_1Br_2Cr_3$	A.n=1
bbcc	$r_4r_4r_1Br_2Cr_3$	A.n=2
bbcc	$r_4r_1Br_2Cr_3$	A.n=3
bbcc	$r_1Br_2Cr_3$	A.n=4
bbcc	Br_2Cr_3	A.n=4 B.m=4
bbcc	$br_6Br_7r_2Cr_3$	A.n=4 B.m=4
bcc	$r_6Br_7r_2Cr_3$	A.n=4 B.m=4
bcc	$Br_7r_2Cr_3$	A.n=4 B.m=4 B.m=4
bcc	$br_6Br_7r_7r_2Cr_3$	A.n=4 B.m=4 B.m=4
cc	$r_6Br_7r_7r_2Cr_3$	A.n=4 B.m=4 B.m=4
cc	$Br_7r_7r_2Cr_3$	A.n=4 B.m=4 B.m=4 B.m=4
cc	$r_8r_7r_7r_2Cr_3$	A.n=4 B.m=4 B.m=4 B.m=4
cc	$r_7r_7r_2Cr_3$	A.n=4 B.m=4 B.m=4 B.m=4 B.n=4
cc	$r_7r_2Cr_3$	A.n=4 B.m=4 B.m=4 B.n=3
cc	r_2Cr_3	A.n=4 B.m=4 B.n=2
cc	Cr_3	A.n=4 B.m=4 B.n=2 C.m=2
cc	$cr_9Cr_{10}r_3$	A.n=4 B.m=4 B.n=2 C.m=2
c	$r_9Cr_{10}r_3$	A.n=4 B.m=4 B.n=2 C.m=2
c	$Cr_{10}r_3$	A.n=4 B.m=4 B.n=2 C.m=2 C.m=2
c	$cr_9Cr_{10}r_{10}r_3$	A.n=4 B.m=4 B.n=2 C.m=2 C.m=2
	$r_9Cr_{10}r_{10}r_3$	A.n=4 B.m=4 B.n=2 C.m=2 C.m=2
	$Cr_{10}r_{10}r_3$	A.n=4 B.m=4 B.n=2 C.m=2 C.m=2 C.m=2
	$r_{11}r_{10}r_{10}r_3$	A.n=4 B.m=4 B.n=2 C.m=2 C.m=2 C.m=2
	$r_{10}r_{10}r_3$	A.n=4 B.m=4 B.n=2 C.m=2 C.m=2 C.m=2 C.n=2
	$r_{10}r_3$	A.n=4 B.m=4 B.n=2 C.m=2 C.m=2 C.n=1

Entrada	Pilha sintática	Pilha de atributos
	r_3	A.n=4 B.m=4 B.n=2 C.m=2 C.n=0
		S.ok=TRUE

□

Observamos que um analisador ascendente não pode ser diretamente utilizado, porque só anuncia a regra utilizada ao final da análise do trecho correspondente. Assim, o uso direto de um analisador ascendente só permite o uso de atributos sintetizados. Por exemplo, numa regra

$X \rightarrow r_1\ A\ r_2\ B\ r_3\ C\ r_4$,

a única ação que poderia ser executada seria a ação r_4, que calcula atributos sintetizados de X. Isso limita as ações semânticas ao uso apenas de atributos sintetizados, e a ações disparadas pelas reduções pelas regras da gramática.

Como um analisador descendente sempre decide qual a regra a ser utilizada em um ponto da cadeia de entrada anterior ao ponto onde um analisador ascendente faz a escolha da (mesma) regra, um analisador descendente pode simular com facilidade um analisador ascendente de forma *on-line*: basta colocar marcadores como r_4 acima, no fim de cada regra. Entretanto, um analisador ascendente só pode simular um analisador descendente de maneira *off-line*: a partir de um parse direito (invertido) é possível construir um parse esquerdo. Essa construção é feita da seguinte maneira:

- o analisador ascendente lê toda a cadeia de entrada, e constrói uma árvore de derivação, usando técnicas semelhantes às vistas na seção 4.1.
- a árvore de derivação é então visitada em pré-ordem (primeiro a raiz, depois as sub-árvores da esquerda para a direita), e o parse esquerdo é construído.

O exemplo a seguir ilustra o processo.

Exemplo 5: Considere a gramática de atributos do Exemplo 4, e a mesma cadeia de entrada aaaabbcc. Com essa entrada, um analisador ascendente encontraria as regras na ordem 3 2 2 2 2 5 4 4 7 6 6 1. Isso significa, que a árvore de derivação (com apenas os números das regras nos nós, como visto na seção 4.1, seria a árvore apresentada na Fig. 8. Percorrendo a árvore em pré-ordem, temos a ordem em que as regras seriam anunciadas pelo analisador ascendente: 1 2 2 2 2 3 4 4 5 6 6 7.

Outras maneiras de tratar o problema levam à alteração da gramática para inserir pontos de redução adicionais onde se fizerem necessários. O problema de criar pontos de redução adicionais pode ser exemplificado pela regra

$A \rightarrow B\ C\ D\ E$

onde suporemos a necessidade de inclusão de uma ação semântica r entre os símbolos C e D. Uma primeira maneira de alterar a gramática é através da introdução de um nãoterminal F, de forma que a regra dada é substituída por

(1) $A \rightarrow F\ D\ E$
$F \rightarrow B\ C$

e a ação desejada pode ser associada à redução pela regra de F. Outra maneira, possivelmente mais intuitiva leva à substituição da regra dada por

(2) A → B C F D E
F → ε

e, novamente, a ação semântica pode ser associada à redução pela regra de F.

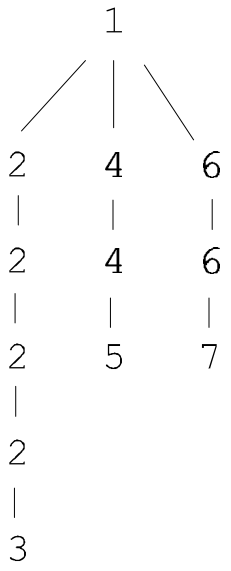


Fig. 8 - Árvore de derivação de aaabbcc

□

Uma dificuldade que pode ser encontrada é que alguns atributos poderão não estar disponíveis, devido à rearrumação da gramática. Por exemplo, os atributos de F, na regra F → ε (2), devem ser calculados a partir dos atributos sintetizados dos símbolos do lado direito da regra. O problema é que a regra não tem símbolos do lado direito. Veremos no Cap. 5 algumas maneiras de contornar estes problemas.

4.5 - Cálculo de atributos sintetizados

A maneira mais comum de tratamento de atributos usa apenas atributos sintetizados. Neste caso, não há necessidade de várias visitas às sub-árvores de uma árvore de derivação, uma vez que não existe a alternância de atributos herdado/sintetizado (subir/descer), e todos os atributos podem ser calculados de uma vez, subindo na árvore de derivação. Podemos usar um analisador ascendente, e tratar a pilha de atributos de forma mais simples. Para uma regra, $A \rightarrow X_1\ X_2\ \ldots\ X_m$, só pode haver uma ação, no fim do lado direito da regra, que é executada no momento em que o analisador ascendente anuncia a redução pela regra. Como no caso da seção anterior, a ação consiste em

- buscar na pilha os atributos (sintetizados) de X_m, ..., X_2, X_1, nessa ordem;
- calcular os atributos (sintetizados) de A em função dos atributos de X_1, X_2, ..., X_m;
- empilhar os atributos de A.

Exemplo 6: Considere a gramática de atributos a seguir.

1. $N \rightarrow I^1 \; . \; I^2$
   $N.v := I^1.v + I^2.v * 2^{-I2.l}$;
2. $N \rightarrow I$
   $N.v := I.v$;
3. $I^0 \rightarrow I^1 \; B$
   $I^0.v := I^1.v * 2 + B.v$;
   $I^0.l := I^1.l + 1$;
4. $I \rightarrow B$
   $I.v := B.v$;
   $I.l := 1$;
5. $B \rightarrow 0$
   $B.v := 0$;
6. $B \rightarrow 1$
   $B.v := 1$;

Como se pode ver, esta gramática é equivalente à gramática do Exemplo 2, no sentido de que os valores de `N.v` calculados são os mesmos, e que esta gramática só tem atributos sintetizados. Vamos calcular o valor de `N.v` para a mesma cadeia `101.011` usada no Exemplo 2. O processo está descrito na tabela seguinte, que mostra o conteúdo das pilhas sintática e de atributos durante a avaliação.

Pilha sintática	Entrada	Pilha de atributos
	101.011	
1	01.011	
B	01.011	B.v=1
I	01.011	I.v=1 I.l=1
I0	1.011	I.v=1 I.l=1
IB	1.011	I.v=1 I.l=1 B.v=0
I	1.011	I.v=2 I.l=2
I1	.011	I.v=2 I.l=2
IB	.011	I.v=2 I.l=2 B.v=1
I	.011	I.v=5 I.l=3
I.	011	I.v=5 I.l=3
I.0	11	I.v=5 I.l=3
I.B	11	I.v=5 I.l=3 B.v=0
I.I	11	I.v=5 I.l=3 I.v=0 I.l=1
I.I1	1	I.v=5 I.l=3 I.v=0 I.l=1
I.IB	1	I.v=5 I.l=3 I.v=0 I.l=1 B.v=1
I.I	1	I.v=5 I.l=3 I.v=1 I.l=2
I.I1		I.v=5 I.l=3 I.v=1 I.l=2
I.IB		I.v=5 I.l=3 I.v=1 I.l=2 B.v=1
I.I		I.v=5 I.l=3 I.v=3 I.l=3
N		N.v=$5\frac{3}{8}$

□

Naturalmente, em vez de incluir diretamente os valores dos atributos na pilha, podemos empilhar apontadores para registros onde ficam os valores. Em geral, a programação fica mais simples, porque o tipo do elemento da pilha é fixo e conhecido, e, além disso, o programador fica dispensado de prestar atenção na ordem de empilhamento dos atributos de cada símbolo, uma vez que os campos dos registros tem nomes.

Exercício 2: Considere a gramática de atributos descrita a seguir

```
1. S→T
     S.ok:=(T.bs=0);
     T.ad:=0;
     T.bd:=T.as;

2. T⁰→(T¹;T²)
     T¹.ad:=T⁰.ad;
     T².ad:=T¹.as;
     T⁰.as:=T².as;
     T².bd:=T⁰.bd;
     T¹.bd:=T².bs;
     T⁰.bs:=T¹.bs;

3. T→a
     T.as:=T.ad+1;
     T.bs:=T.bd

4. T→b
     T.as:=T.ad;
     T.bs:=T.bd-1;
```

1. determine quais são os atributos herdados e sintetizados, e quais as dependências entre eles.
2. caracterize as cadeias para as quais `S.ok=true`;
3. construa um avaliador recursivo para a gramática;
4. construa uma gramática de atributos que tenha apenas atributos sintetizados, e que seja equivalente à gramática dada, no sentido de que `S.ok=true` para exatamente as mesmas cadeias da linguagem.
5. construa um avaliador para esta última gramática.

Apêndice 4.A
Exemplo de gramática "pura" de atributos (sem variáveis globais)

A gramática aqui descrita permite a geração de código para programas escritos em uma linguagem bastante restrita, e é apresentada para justificar a afirmação de que é possível descrever o processo de geração de código através de uma gramática de atributos, embora seja mais freqüente o uso de gramáticas de atributos que se encarregam apenas da verificação semântica e da geração de código em alguma representação intermediária adequada.

1. Linguagem

A linguagem fonte aqui considerada é descrita pela gramática a seguir:

```
prog → bloc "."
bloc → BEGIN decs coms END
decs[0] → decs[1] dec ";"
decs → ε
dec → INTEGER idens
idens[0] → idens[1] "," iden
idens → iden
iden → ID
coms[0] → coms[1] ";" com
coms → com
com → bloc
com → var ":=" expr
com[0] → IF expr THEN com[1] ELSE com[2]
com → GOTO rot
com[0] → def ":" com[1]
com → ε
expr[0] → expr[1] "+" term
expr → term
term[0] → term[1] "*" fat
term → fat
fat → ( expr )
fat → var
fat → num
var → ID
def → ID
rot → ID
num → NUM
```

Como se pode ver, permite apenas blocos com declarações de variáveis simples inteiras e comandos. Comandos podem ser blocos (recursivamente), comandos de atribuição, comandos `if` e comandos `goto`. Os comandos podem ser rotulados para referência em comandos `goto`, mas rótulos só podem ser usados no bloco/nível em que foram definidos. As expressões da linguagem só podem ter parênteses, soma e produto;

A separação dos identificadores em `var`, `def` e `rot` auxilia o tratamento semântico dado.

2. Atributos

Os atributos utilizados estão na tabela a seguir, classificados em herdados e sintetizados.

	Atributos	
Símbolos	Herdados	Sintetizados
`prog`	—	`cod`
`decs dec idens iden`	`pos`	`tss esp`
`bloc`	`ini tsh pos`	`cod tam`
`coms com`	`ini tsh trh pos`	`cod tam trs`
`expr term fat num`	`tsh pos`	`cod tam end`
`var`	`tsh`	`end`
`rot`	`trh`	`loc`
`def`	`ini`	`trs`
`ID`	`nom`	—
`NUM`	`val`	—

A tabela a seguir descreve os valores de cada atributo.

Atributo	Descrição
`pos`	primeira posição livre na área (segmento) de dados
`ini`	primeira instrução do código correspondente
`tsh`	tabela de símbolos válida no local (herdada)
`trh`	tabela dos rótulos visíveis no local (herdada)
`nom`	seqüência de símbolos associada a um identificador (fornecido pelo scanner)
`val`	valor associado a um número (fornecido pelo scanner)
`cod`	código executável (seqüência de instruções)
`tss`	tabela de símbolos parcial (sintetizada)
`esp`	espaço ocupado por uma declaração ou fragmento (supõe 2 bytes por inteiro)
`tam`	tamanho (número de instruções) de código
`trs`	tabela de rótulos parcial (sintetizada)
`end`	endereço de variável (declarada ou temporária)
`loc`	primeira instrução do código correspondente

3. Instruções

A máquina para a qual o código é gerado é extremamente simplificada, dispondo apenas de um acumulador para uso com instruções aritméticas. As instruções usadas no código fonte são

transferência	`L`	Load
	`Li`	Load imediato
	`S`	Store
aritmética	`A`	Add
	`M`	Multiply
desvio	`J`	Jump
	`JF`	Jump if False

4. Funções auxiliares usadas na gramática de atributos

Algumas das funções devem fazer verificações semânticas adicionais, por exemplo para tratar identificadores declarados mais de uma vez.

`junta(ts1,ts2)`
: faz a união das tabelas de símbolos `ts1` e `ts2`, dando preferência à entrada na tabela de ts2, quando um identificador aparece nas duas tabelas.

`une(ts1,ts2)`
: faz a união das tabelas de símbolos `ts1` e `ts2`, indicando erro, quando um identificador aparece nas duas tabelas.

`consulta(ts, nome)`
: procura nome na tabela `ts`, e devolve o endereço correspondente, indicando erro, caso nome não ocorra na tabela.

5. Gramática de atributos

`prog → bloc "."`

`prog.cod := bloc.cod`	`bloc.ini := 0`
	`bloc.tsh := ∅`
	`bloc.pos := 0`

`bloc → BEGIN decs coms END`

`bloc.cod := coms.cod`
`bloc.tam := coms.tam`
	`decs.pos := bloc.pos`
	`coms.ini := bloc.ini`
	`coms.tsh :=`
	`  junta(decs.tss, bloc.tsh)`
	`coms.trh := coms.trs`
	`coms.pos :=`
	`  bloc.pos + decs.esp`

$decs^0 \rightarrow decs^1$ `dec ";"`

$decs^0$`.tss :=`
`  une(`$decs^1$`.tss, dec.tss)`
$decs^0$`.esp :=`
`  `$decs^1$`.esp + dec.esp`
	$decs^1$`.pos :=` $decs^0$`.pos`
	`dec.pos :=`
	`  `$decs^0$`.pos + `$decs^1$`.esp`

`decs →` ε

`decs.tss := ∅` `decs.esp := ∅`

`dec → INTEGER idens`

`dec.tss := idens.tss` `dec.esp := idens.esp`	`idens.pos := dec.pos`

$idens^0 \rightarrow idens^1$ `"," iden`

$idens^0$`.tss :=`
`  une(`$idens^1$`.tss, iden.tss)`
$idens^0$`.esp :=`
`  `$idens^1$`.esp + iden.esp`
	$idens^1$`.pos :=` $idens^0$`.pos`
	`iden.pos :=`
	`  `$idens^0$`.pos + `$idens^1$`.esp`

idens → iden

idens.tss := iden.tss idens.esp:=iden.esp	iden.pos:=idens.pos

iden → ID

iden.tss := {[ID.nom, iden.pos]} iden.esp := 2

$coms^0$ → $coms^1$ ";" com

$coms^0$.cod := $coms^1$.cod ⊕ com.cod $coms^0$.tam := $coms^1$.tam + com.tam $coms^0$.trs := une($coms^1$.trs, com.trs)	$coms^1$.ini := $coms^0$.ini $coms^1$.tsh := $coms^0$.tsh $coms^1$.trh := $coms^0$.trh $coms^1$.pos := $coms^0$.pos com.ini := $coms^0$.ini com.tsh := $coms^0$.tsh com.trh := $coms^0$.trh com.pos := $coms^0$.pos

coms → com

coms.cod := com.cod coms.tam := com.tam coms.trs := com.trs	com.ini := coms.ini com.tsh := coms.tsh com.trh := coms.trh com.pos := coms.pos

com → bloc

com.cod := bloc.cod com.tam := bloc.tam com.trs := ∅	bloc.ini := com.ini bloc.tsh := com.tsh bloc.pos := com.pos

com → var ":=" expr

com.cod := expr.cod ⊕ [L expr.end] ⊕ [S var.end] com.tam := expr.tam + 2 com.trs := ∅	var.tsh := com.tsh expr.tsh := com.tsh expr.pos := com.pos

com^0 → IF expr THEN com^1 ELSE com^2

com^0.cod := expr.cod ⊕ [JF expr.end com^2.ini] ⊕ com^1.cod ⊕ [J com^2.ini + com^2.tam] ⊕ com^2.cod com^0.tam := expr.tam + com^1.tam + com^2.tam + 2 com^0.trs := 0	expr.tsh := com^0.tsh expr.pos := com^0.pos com^1.ini := com^0.ini + expr.tam + 1 com^1.tsh := com^0.tsh com^1.trh := com^0.trh com^1.pos := com^0.pos com^2.ini := com^1.ini + com^1.tam + 1 com^2.tsh := com^0.tsh com^2.trh := com^0.trh com^2.pos := com^0.pos

com → GOTO rot

com.cod := [J rot.loc] com.tam := 1 com.trs := 0	rot.trh:=com.trh

com^0 → def ":" com^1

com^0.cod := com^1.cod com^0.tam := com^1.tam com^0.trs := une(com^1.trs, def.trs)	def.ini:=com^0.ini com^1.ini := com^0.ini com^1.tsh := com^0.tsh com^1.trh := com^0.trh com^1.pos := com^0.pos

com → ε

com.cod := ε com.tam := 0 com.trs := ∅

$expr^0$ → $expr^1$ "+" term

$expr^0$.cod := $expr^1$.cod ⊕ term.cod ⊕ [L $expr^1$.end] ⊕ [A term.end] ⊕ [S $expr^0$.end] $expr^0$.tam := $expr^1$.tam + term.tam + 3 $expr^0$.end := $expr^0$.pos	$expr^1$.tsh := $expr^0$.tsh $expr^1$.pos := $expr^0$.pos term.tsh := $expr^0$.tsh term.pos := $expr^0$.pos + 2

expr → term

expr.cod := term.cod expr.tam := term.tam expr.end := term.end	term.tsh := expr.tsh term.pos := expr.pos

$term^0$ → $term^1$ "*" fat

$term^0$.cod := $term^1$.cod ⊕ fat.cod ⊕ [L $term^1$.end] ⊕ [M fat.end] ⊕ [S $term^0$.end] $term^0$.tam := $term^1$.tam + fat.tam + 3 $term^0$.end := $term^0$.pos	$term^1$.tsh := $term^0$.tsh term.pos := term.pos fat.tsh := $term^0$.tsh fat.pos := $term^0$.pos + 2

term → fat

term.cod := fat.cod term.tam := fat.tam term.end := fat.end	fat.tsh := term.tsh fat.pos := term.pos

fat → (expr)

fat.cod := expr.cod fat.tam := expr.tam fat.end := expr.end	expr.tsh := fat.tsh expr.pos := fat.pos

fat → var

fat.cod := ε fat.tam := 0 fat.end := var.end	var.tsh := fat.tsh

fat → num

fat.cod := num.cod fat.tam := num.tam fat.end := num.end	num.tsh := fat.tsh num.pos := fat.pos

var → ID

```
var.end :=
   consulta(var.tsh, ID.nom)
```

def → ID

```
def.trs:={[iden.nom, def.ini]}
```

rot → ID

```
rot.loc :=
   consulta(rot.trh, ID.nom)
```

num → NUM

num.cod := [Li NUM.val] ⊕ [S num.end] num.tam := 2	num.end := num.pos

(rev. abr 99)