Um Tradutor Dirigido por Sintaxe

Aula introdutória às técnicas de compilação. Ilustra técnicas desenvolvendo um programa Java que traduz instruções de uma linguagem de programação representativa para código de três endereços (representação intermediária).

A ênfase é dada ao *front-end* de um compilador, em particular, à análise léxica, sintática e à geração de código intermediário.

Um fragmento de código a ser traduzido:

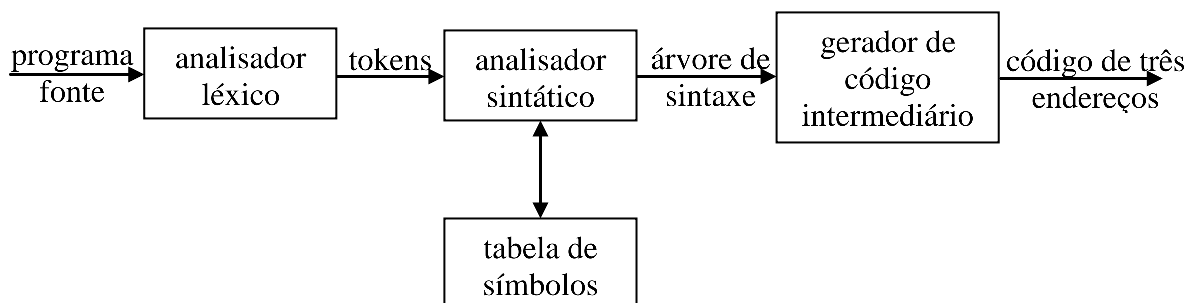
```
{
      int i; int j; float[100] a; float v; float x;
      while (true) {
            do i = i+1; while (a[i] < v);
            do j =  j-1; while (a[j] > v);
            if (i >= j) break;
            x = a[i]; a[i] = a[j]; a[j] = x;
      }
}
```

A fase de análise de um compilador subdivide um programa fonte em partes constituintes e produz uma representação interna para ele, chamada código intermediário.

A fase de síntese traduz o código intermediário para o programa objeto.

A análise é organizada em torno da *sintaxe* da linguagem a ser compilada. A sintaxe descreve a forma apropriada dos seus programas. A semântica define o que seus programas significam (o que cada programa faz quando é executado.

A sintaxe utiliza uma notação chamada gramáticas livre de contexto ou Backus-Naur Form (BNF).



Um analisador léxico permite que um tradutor trate as construções de múltiplos caracteres como identificadores, que são escritos como sequências de caracteres, mas são tratados como unidades chamadas *tokens* durante a análise sintática.

O analisador sintático produz uma árvore sintática (árvore sintática abastrata) que é traduzida em um código de três endereços.

Definição da sintaxe

A gramática livre de contexto, ou gramática, é usada para especificar a sintaxe de uma linguagem.

Uma gramática descreve naturalmente a estrutura hierárquica da maioria das construções de linguagens de programação.

Exemplo: um comando if-else em Java pode ter a forma

```
if (expressão) instrução else instrução
```

Um comando if-else é a concatenação da palavra-chave `if`, um parênteses de abertura, uma expressão, um parênteses de fechamento, um comando, a palavra-chave `else` e outro comando.

Usando a variável `expr` para representar uma expressão e a variável `stmt` para denotar um comando, essa regra de estruturação pode ser expressa como

`stmt → ` **`if`** ` (expr) stmt ` **`else`** ` stmt`

onde a seta pode ser lida como *pode ter a forma*. Essa regra é chamada de *produção*.

Em uma produção os elementos léxicos (palavra-chave e os parênteses) são chamados de *terminais*. Variáveis como expr e stmt representam sequências de terminais e são chamadas de *não-terminais*.

Um token consiste em um nome de token e um valor de atributo. Os nomes de token são símbolos abstratos usados pelo analisador para fazer o reconhecimento sintático. Normalmente nos referimos aos tokens e aos terminais como sinônimos.

Definição de gramáticas

Uma gramática livre de contexto possui quatro componentes:

1. Um conjunto de símbolos terminais (tokens).
2. Um conjunto de não-terminais (variáveis sintáticas).
3. Um conjunto de produções.
4. Uma designação de um dos não-terminais como o símbolo inicial da gramática.

Exemplo: expressões constituídas em dígitos e sinais de adição e subtração, como cadeias: `9 - 5 + 2`, `3 - 1` ou `7`. A gramática a seguir descreve a sintaxe dessas expressões. As produções são:

list → list + digit
list → list – digit
list → digit
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Os corpos das três produções com o não-terminal *list* como cabeça podem ser agrupados de foma equivalente como:

list → list + digit | list – digit | digit

De acordo com nossas convenções, os terminais dessa gramática são os símbolos `+ - 0 1 2 3 4 5 6 7 8 9`. Os não-terminais são os nomes list e digit, como list sendo o símbolo inicial.

Derivações

Uma gramática deriva cadeias começando com o símbolo inicial e substituindo repetidamente um não-terminal pelo corpo de uma produção para esse não-terminal. As cadeias de terminais que podem ser derivadas do símbolo inicial formam a *linguagem* definida pela gramática.

Exemplo 1: a linguagem definida pela gramática acima, consiste em listas de dígitos separados por sinais de adição e subtração. As dez produções para o não-terminal *digit* permitem que ele represente qualquer um dos terminais `0, 1, ..., 9`.

Podemos deduzir que `9 - 5 + 2` é uma lista da seguinte maneira:

a) `9` é uma lista, pois `9` é um dígito.
b) `9 - 5` é uma lista, pois `9` é uma lista e `5` é um dígito.
c) `9 - 5 + 2` é uma lista, pois `9 - 5` é uma lista e `2` é um dígito.

Exemplo 2: um tipo diferente de lista é a lista de parâmetros em uma chamada de função. Em Java, os parâmetros são delimitados por parênteses, como na chamada da função `max (x, y)`.

Podemos começar a desenvolver uma gramática para essas sequências com as produções:

call → id (optparams)
optparams → params | ∈
params → params, param | param

- *optparams* (lista de parâmetros opcionais) é ∈ (cadeia de zero símbolo de qualquer alfabeto), cadeia de símbolos vazia. *optparams* pode ser substituída pela cadeia vazia – uma chamada de função, *call*, pode consistir em um nome de função seguido pelos dois símbolos terminais ( ).
- *params* são semelhantes à *list*, com a vírgula no lugar do operador aritmético + ou –, e *param* no lugar de *digit*.

A análise sintática é um dos problemas mais importantes em toda a compilação. Ela consiste em, a partir de uma cadeia de terminais, tentar descobrir como derivá-la a partir do símbolo inicial da gramática, e, se possível, informar os erros de sintaxe dentro dessa cadeia.

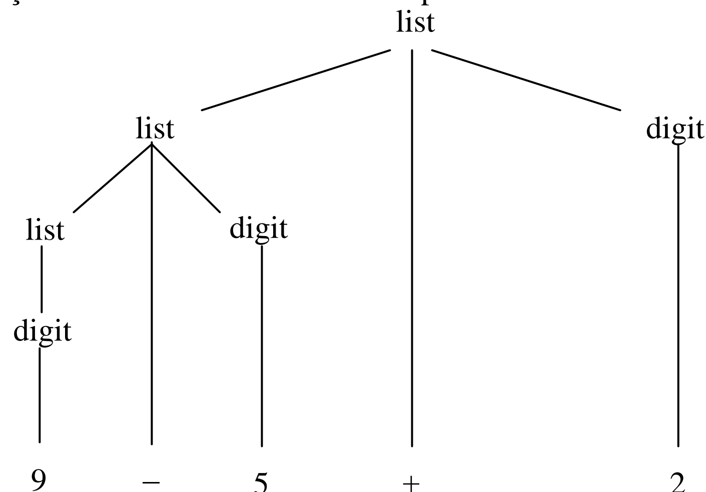
Árvores de derivação

Mostra de forma representativa como o símbolo inicial de uma gramática deriva uma cadeia na linguagem.

Dada uma gramática livre de contexto, uma árvore de derivação de acordo com a gramática é uma árvore com as seguintes propriedades:

1. A raiz é rotulada pelo símbolo inicial.
2. Cada folha é rotulada por um terminal ou por ∈.
3. Cada nó interior é rotulado por um não-terminal.
4. Se A é o não-terminal rotulando algum nó interior e X_1, X_2, ..., X_n são os rótulos dos filhos desse nó da esquerda para a direita, deve haver uma produção A → X_1, X_2, ..., X_n. Cada X_1, X_2, ..., X_n representa um símbolo que é um terminal ou um não-terminal. Se A → ∈ é uma produção, um nó rotulado com A pode ter um único filho rotulado com ∈.

Exemplo: a derivação de 9 – 5 + 2 é ilustrada pela árvore.



- Cada nó da árvore é rotulado por um símbolo da gramática.
- Um nó interno e seus filhos correspondem a uma produção.
- A raiz é rotulada como *list*, o símbolo inicial da gramática.
- Os filhos são rotulados, da esquerda para a direita
- O filho esquerdo da raiz é semelhante à raiz.
- Os três nós rotulados com *digit* possuem um filho que é rotulado por um dígito.

Da esquerda para a direita, as folhas de uma árvore de derivação formam o resultado da árvore, que é a cadeia gerada ou derivada do não-terminal na raiz da árvore de derivação

Outra definição da linguagem gerada por uma gramática é o conjunto de cadeias de terminais que podem ser geradas por alguma árvore de derivação. O processo de encontrar uma árvore de derivação para determinada cadeia de terminais é chamado de *análise* ou *reconhecimento* dessa cadeia.

Ambiguidade

Uma gramática pode ter mais de uma árvore de derivação gerando a determinada cadeia de terminais (gramática ambígua). Para uma gramática ser ambígua é preciso encontrar uma cadeia de terminais que seja o resultado de mais de uma árvore de derivação.

Exemplo: suponha que usemos um único não-terminal *string* e que não façamos distinção entre dígitos e listas. Poderíamos escrever a gramática

```
string → string + string | string - string | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

A figura abaixo mostra que uma expressão como `9 - 5 + 2` tem mais de uma árvore de derivação para essa gramática.

As duas arvores correspondem às duas maneiras de colocar parenteses na expressão: `(9 - 5) + 2` e `9 - (5 + 2)`.

Associatividade de operadores

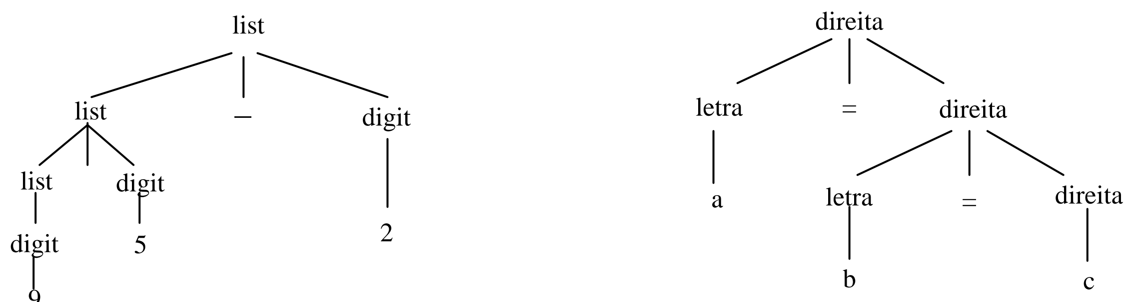
Por convenção, `9+5+2` é equivalente a `(9+5)+2` e `9 - 5 - 2` é equivalente a `(9 - 5) - 2`. Quando um operando como o `5` possui operadores à sua esquerda e direita, são necessários convenções para decidir qual operador se aplica a esse comando.

Na maioria das linguagens de programação, os quadro operadores aritméticos, adição, subtração, multiplicação e divisão, são associativos à esquerda.

Alguns operadores como exponenciação, são associativos à direita. Por exemplo, o operador de atribuição = em C e seus descendentes são associativos à direita. A expressão `a= b =c` é tratada da mesma forma que a expressão `a = (b = c)`.

```
direita → letra = direta | letra
letra → a | b | ... | z
```

Exemplo: a árvore de derivação para `9 - 5 - 2` cresce para baixo e à esquerda; a árvore de derivação para `a = b = c` cresce para baixo e à direita.



Precedência de operadores

Considere a expressão: `9 + 5 * 2`. Há duas interpretações possíveis para essa expressão: `(9 + 5) * 2` ou `9 + (5 * 2)`. As regras que definem a precedência relativa dos operadores são necessárias quando mais de um tipo de operador estiver presente.

Na aritmética comum, a multiplicação e a divisão possuem uma predecência maior do que a adição e a subtração.

Uma gramática para expressões aritméticas pode ser construída a partir de uma tabela mostrando a associatividade e precedência de operadores.

Os operadores na mesma linha possuem a mesma associatividade e precedência:

associativo à esquerda: + –
associativo à esquerda: * /

Exemplo: `fator → dígito | (expr)`

Considere os operadores * e /, que possuem precedência mais alta.

```
termo    →    termo * fator
         |    termo / fator
         |    fator
```

Da mesma forma, *expr* gera listas de termos separados pelos operadores de adição.

```
expr →    expr + termo
     |    expr - termo
     |    termo
```

A gramática resultante é:

```
expr      →    expr + termo | expr - termo | termo
termo     →    termo * fator | termo / fator | fator
fator     →    digito | (expr)
```

Exemplo: As palavras-chaves nos permitem reconhecer comandos. As exceções a essa regra incluem as atribuições e as chamadas de procedimento.

```
stmt      →    id = expressao;
          |    if (expressao) stmt
          |    if (expressao) stmt else stmt
          |    while (expressao) stmt
          |    do stmt while (expressao);
          |    {stmts}
stmts     →    stmts stmt
          |    ∈
```

Tradução dirigida por sintaxe

A tradução dirigida por sintaxe é feita anexando-s regras ou fragmentos de programa às produções de uma gramática.

Dois conceitos estão relacionados à tradução dirigida por sintaxe:

- *Atributos:* qualquer valor associado a uma construção de programação (tipos de dados de expressões, o número de instruções no código gerado, ou o endereço da primeira instrução no código gerado para uma instrução, etc.).
- *Esquemas de tradução:* notação para conectar fragmentos de programa às produções de uma gramática.

A combinação do resultado da execução de todos esses fragmentos, na ordem induzida pela análise sintática, produz a tradução do programa para o qual esse processo de análise e síntese é aplicado.

Notação pós-fixada

A notação pós-fixada para uma expressão E pode ser definida da seguinte forma:

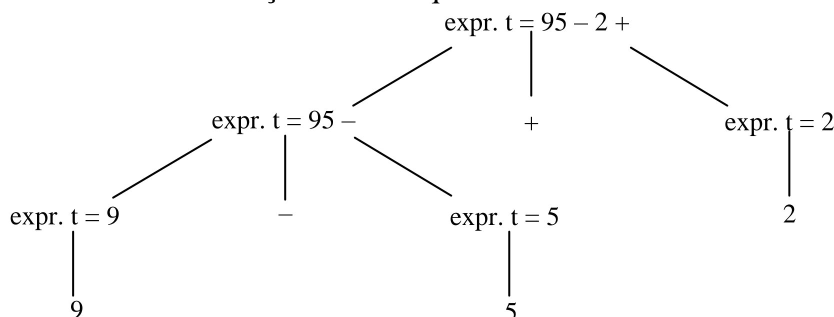
1. Se E é uma variável ou constante, então a notação pós-fixada para E é a própria E.
2. Se E é uma expressão da forma E_1 op E_2, onde op é qualquer operador binário, então a notação pós-fixada para E é E_1' E_2' op.
3. Se E é uma expressão entre parênteses da forma (E_1), então a notação pós-fixada para E é a mesma que a notação pós-fixada para E_1.

Exemplo: a notação pós-fixada para (9 – 5) + 2 é 95 – 2+

Atributos sintetizados

Um atributo é considerado sintetizado se o seu valor no nó da árvore de derivação N for determinado a partir dos valores dos atributos nos filhos de N e no próprio N. Os atributos sintetizados têm a propriedade desejável de poderem ser avaliados durante uma única travessia de baixo para cima na árvore de derivação.

Exemplo: a árvore de derivação anotada que mostra os valores dos atributos



é baseada na definição dirigida por sintaxe para traduzir expressões,

Produção	Regras semânticas
expr → $expr_1$ + term	expr.t = $expr_1$.t || term.t || '+'
expr → $expr_1$ – term	expr.t = $expr_1$.t || term.t || '–'
expr → term	expr = term.t
term → 0	term.t = '0'
term → 1	term.t = '1'
...	...
term → 9	Term = '9'

consistindo em dígitos separados por sinais de adição ou subtração em notação pós-fixada. Cada não-terminal possui um atributo t com valor string, que representa a notação pós-fixada para a expressão gerada por esse não-terminal em uma árvore de derivação. O símbolo || na regra semântica é o operador para concatenação das cadeias.

Caminhamento em árvore

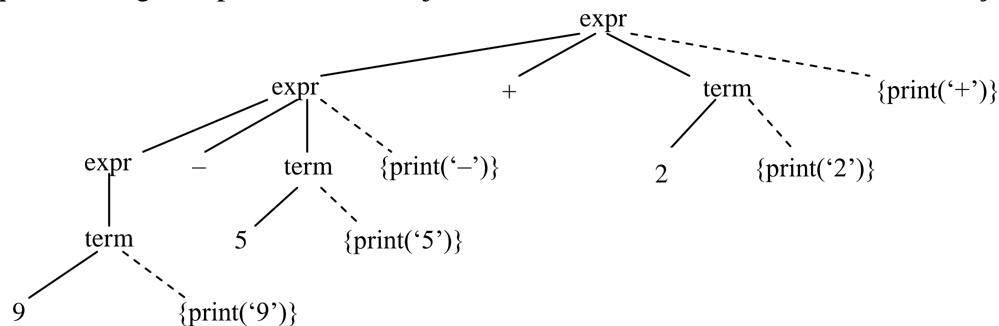
Os caminhamentos (travessias) de uma árvore começa na raiz e visita cada nó da árvore em alguma ordem.

Uma busca em *profundidade* começa na raiz e visita recursivamente os filhos de cada nó em qualquer ordem. A busca em profundidade visita os nós mais distantes (mais profundos) da raiz o mais rapidamente que puder.

Esquemas de tradução

Um esquema de tradução dirigido por sintaxe é uma notação para especificar uma tradução conectando fragmentos de programação a produções de uma gramática.

Um esquema de tradução é como uma definição dirigida por sintaxe, exceto pelo fato de que a ordem de avaliação das regras semânticas é especificada explicitamente.
Exemplo: a árvore de derivação abaixo possui comandos de impressão, *print*, em folhas extras, que estão ligadas por arestas tracejadas aos nós interiores da árvore de derivação.



O esquema de tradução aparece na abaixo.

```
expr →    expr1 + term   {print(' + ')}
expr →    expr1 - term   {print(' - ')}
expr →    term
term →    0          {print(' 0 ')}
term →    1          {print(' 1 ')}
          ...
term →    9          {print(' 9 ')}
```

A gramática mostrada gera expressões consistindo em dígitos separados pelos operadores de adição e subtração. As ações embutidas nos corpos das produções traduzem essas expressões para a notação pós-fixada, desde que realizemos uma busca em profundidade, da esquerda para a direita, e executemos cada comando *print* quando visitarmos sua folha.

Análise sintática

É o processo para determinar como uma cadeia de terminais pode ser gerada por uma gramática.

As maiorias dos métodos de análise sintática estão em uma de duas classes, chamadas métodos *descendentes* e *ascendentes*.

Esses termos referem-se à ordem em que os nós na árvore de derivação são construídos. Nos analisadores decrescentes a árvore é construída de cima para baixo (da raiz para as folhas). A análise ascendente pode tratar de uma classe maior de gramáticas e esquemas de tradução, de modo que as ferramentas de software para gerar analisadores diretamente a partir de gramáticas normalmente utilizam métodos ascendentes.

Análise sintática descentedente

A construção de uma árvore de derivação é feita iniciando-se na raiz, rotulada com o não-terminal inicial `stmt`, e realizando repetidamente os dois passos a seguir.

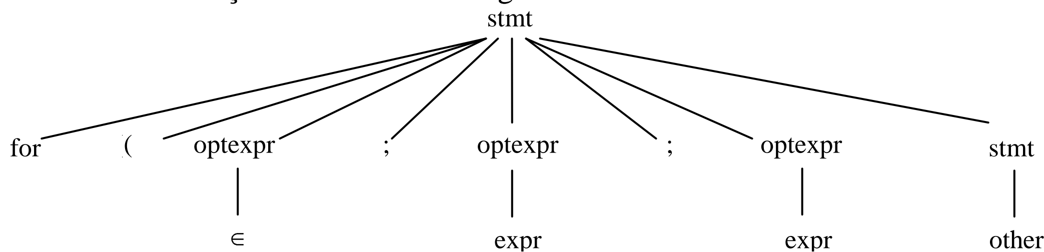
1. No nó N, rotulado com o não-terminal A, selecione uma das produções para A e construa filhos em N para os símbolos no corpo da produção.
2. Encontre o próximo nó em que uma subárvore deve ser construída, normalmente o não-terminal não expandido mais à esquerda da árvore.

Exemplo: uma gramática para alguns comandos em C e Java.

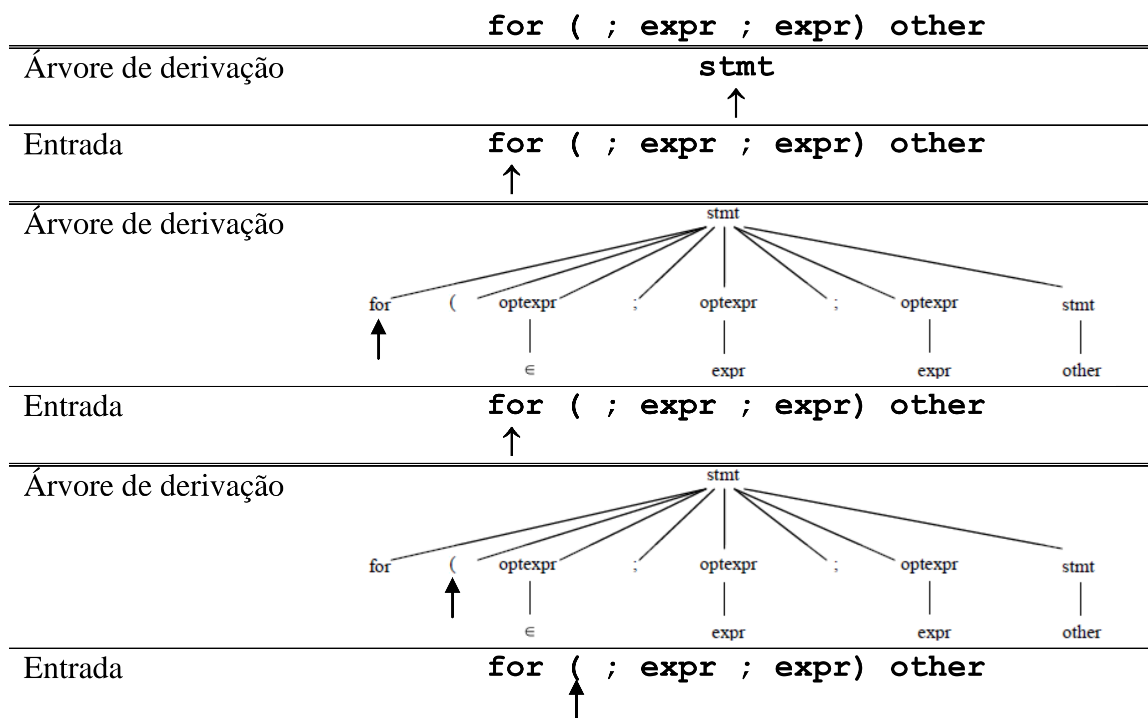
```
stmt      →    expr;
          |    if (expr) stmt
```

```
                |    for (optexpr; optexpr; optexpr) stmt
                |    other
optexpr         →    ∈
                |    expr
```

Uma árvore de derivação de acordo com a gramática acima.



Análise sintática descendente processando a entrada da esquerda para a direita.



Analisador sintático preditivo

O analisador preditivo no pseudocódigo abaixo mostra os procedimentos para os não-terminais `stmt` e `optexpr` da gramática e um procedimento adicional `match`, usado para simplificar o código para `stmt` e `optexpr`.

```
void stmt() {
      switch (lookahead) {
      case expr:
            match(expr);
            match('; ');
            break;
      case if:
            match(if);
            match('(');
            match(expr);
            match(')');
            stmt();
```

```
            break;
      case for:
            match(for);
            match('(');
            optexpr();
            match(';');
            optexpr();
            match(';');
            optexpr();
            match(')');
            stmt();
            break;
      case other:
            match (other);
            break;
      default:
            report("erro de sintaxe");
      }
}
void optexpr() {
      if (lookahead == expr) match(expr);
}
void match(terminal t) {
      if (lookahead == t) lookahead = nextterminal;
      else
            report("erro de sintaxe");
}
```

O procedimento `match(t)` compara seu argumento `t` com o símbolo `lookahead` e avança para o próximo terminal de entrada se eles casarem. Assim, `match` muda o valor da variável `lookahead`, uma variável global que mantém o terminal de entrada atualmente sendo lido.

O reconhecedor preditivo conta com as informações sobre os primeiros símbolos que podem ser gerados pelo corpo de uma produção.

Considere α como uma cadeia de símbolos da gramática (terminais e/ou não-terminais). Definimos FIRST(α) como sendo o conjunto de símbolos terminais que aparecem como primeiros símbolos de uma ou mais cadeias de terminais gerados a partir de α. Se α for $\in$ ou puder gerar $\in$, então $\in$ também está em FIRST(α).

Normalmente, α começa com um terminal, e, portanto é o único símbolo em FIRST(α), ou α começa com um não-terminal cujos corpos da produção começam com terminais, e, neste caso, esses terminais são os únicos membros de FIRST(α).

Exemplo: em relação à gramática apresentada, os seguintes são cálculos corretos de FIRST.

```
FIRST(stmt)    =    {expr, if, for, other}
FIRST (expr;)  =    {expr}
```

Nosso analisador preditivo utiliza uma produção-$\in$ como padrão quando nenhuma outra produção puder ser usada.

Exemplo: depois que os terminais `for` e `(` forem casados, o símbolo lookahead é `;`. Nesse ponto, o procedimento optexpr é chamado e o código

```
if (lookahead == expr) match(expr);
```

é executado.

O não-terminal optexpr possui duas produções, com corpos expr e ∈. O símbolo lookahead ; não casa com o terminal expr, de modo que a produção com corpo expr não pode ser aplicada.

O procedimento retorna sem alterar o símbolo lookahead ou sem fazer alguma outra ação. Não fazer nada corresponde a aplicar uma produção-∈.

Projetando um analisador preditivo

Uma analisador preditivo é um programa consistindo em um procedimento para cada não-terminal. O procedimento para o não-terminal *A* efetua duas ações:

1. Declara qual produção-A deve ser usada examinando o símbolo lookahead.
2. O procedimento em seguida imita o corpo da produção escolhida. Os símbolos são executados a partir da esquerda.

Assim como o esquema de tradução é formado estendendo-se uma gramática, um tradutor dirigido por sintaxe pode ser formando estendendo-se um analisador preditivo.

Recursão à esquerda

A produção recursiva à esquerda do tipo

`expr → expr + term`

onde o símbolo mais à esquerda do lado direito da produção é igual ao não-terminal do lado esquerdo da produção, faça com que o analisador de descida recursivo fique em um loop para sempre.

Uma produção recursiva à esquerda pode ser eliminada pela reescrita da produção problemática.

Exemplo: considere um não-terminal *A* com duas produções

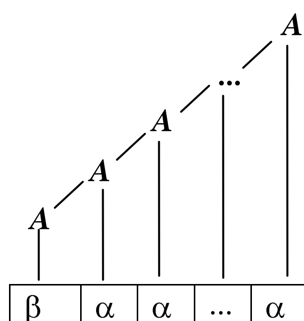
$$A \rightarrow A\alpha \mid \beta$$

onde α e β são sequências de terminais e não-terminais que não começam com *A*. Por exemplo:

`expr → expr + term | term`

o não-terminal `A = expr`, α `= + term`, e β `= term`.

O não-terminal *A* e sua produção são considerados recursivo à esquerda. A produção $A \rightarrow A\alpha$ tem o próprio *A* como símbolo mais à esquerda do lado direito.



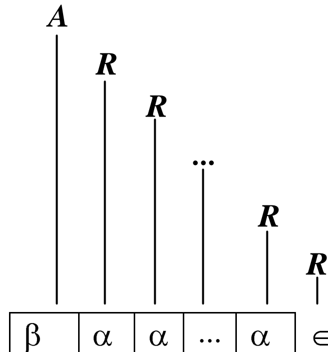
Recursão à direita

O mesmo efeito pode ser conseguido reescrevendo-se as produções para *A*, usando um novo não-terminal *R*:

$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R \mid \in$$

O não-terminal R e sua produção R → αR são recursivos à direita. Essa produção para R tem o próprio R como último símbolo no lado direito da produção.

As produções recursivas à direita produzem árvores que crescem para baixo e para a direita. As árvores que crescem dessa forma dificultam a tradução de expressões contendo operadores associativos à esquerdo, como o operador de subração.



Exercícios

1. Que linguagem é gerada pelas gramáticas a seguir? Justifique sua resposta.
   a) S → 0 S 1 | 0 1
   b) S → +SS | –SS | a
   c) S → S (S) S | ∈
   d) S → a S b S | b S a S | ∈
2. Escreva a notação pós-fixada para a expressão aritmética: `9 - (5 + 2)`.
3. Considere a notação pós-fixada `952+-3*`, escreva a expressão aritmética correspondente.
4. Construa analisadores de descida recursiva, começando com as seguintes gramáticas:
   a) S → +SS | -SS | a
   b) S → S (S) S | ∈
   c) S → 0 S 1 | 0 1