Atividade 07 - Expressões Constantes 2 - Precedência

Andrei de Araújo Formiga

20 de março de 2025

Na linguagem EC1 é possível escrever qualquer expressão com as quatro operações e operandos constantes, mas é preciso usar parênteses para delimitar claramente qual é a ordem das operações. Isso simplifica a análise sintática mas piora a usabilidade para os programadores.

Programadores estão acostumados a escrever expressões em uma notação mais natural, seguindo as mesmas convenções da notação matemática. Esse aliás foi um dos maiores objetivos da linguagem FORTRAN (*FORmula TRANslator*), uma das primeiras linguagens de programação de alto nível: tornar possível, para os cientistas e outros usuários dos computadores na época, escrever e calcular fórmulas seguindo uma notação mais próxima da tradicional, ao invés de uma sequência de instruções de baixo nível como MOV, ADD, etc.

Para fazer corretamente a análise sintática de expressões usando o menor número possível de parênteses, precisamos entender e implementar os conceitos de precedência e associatividade de operadores.

1 Precedência e associatividade

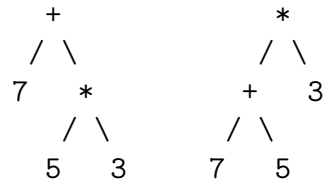
A gramática da linguagem EC1 tem as seguintes produções para expressões:

```
<exp> ::= <num> | '(' <exp> <op> <exp> ')'
```

Para retirar os parênteses obrigatórios, não podemos simplesmente retirar os parênteses da produção:

```
<exp> ::= <exp> <op> <exp>
```

Porque esta regra de produção é ambígua. Uma expressão como $7 + 5 * 3$ pode ser analisada de duas formas, gerando duas árvores possíveis:



A árvore da esquerda representa uma expressão em que a multiplicação é efetuada primeiro, e depois a soma. A árvore da direita representa a outra ordem possível: somar 7 com 3, depois multiplicar o resultado por 5.

Pelas convenções tradicionais de precedência, o esperado é que a multiplicação sempre seja calculada antes da soma. Ou seja, a árvore correta nessa situação seria a da esquerda. Mas com uma regra ambígua como `<exp> <op> <exp>` é mais difícil garantir que a análise produza a árvore correta.

A solução é incorporar a precedência na gramática de expressões. Infelizmente isso cria um problema para o analisador descendente recursivo que estamos usando, como veremos a seguir.

1.1 Precedência na gramática

Uma expressão como 7 + 5 * 3 não é ambígua para quem aprendeu aritmética na escola por causa da regra de precedência: aprendemos que as multiplicações e divisões devem ser efetuadas antes das somas e subtrações.

Para evitar ambiguidade na gramática, vamos codificar as regras de precedência nas produções. A forma mais comum de fazer isso é ter um não-terminal para cada nível de precedência. Para as quatro operações básicas, o exemplo comum é uma gramática com três não-terminais (normalmente chamados de *expressão*, *termo* e *fator*):

```
<exp>   ::= <exp> '+' <termo> | <exp> '-' <termo> | <termo>
<termo> ::= <termo> '*' <fator> | <termo> '/' <fator> | <fator>
<fator> ::= <num> | '(' <exp> ')'
```

`<exp>` representa o nível da adição e subtração, `<termo>` representa o nível da multiplicação e divisão, e `<fator>` representa constantes inteiras isoladas ou expressões entre parênteses, que possuem a precedência mais alta.

O problema de implementar essa gramática em analisador descendente recursivo está em produções como a primeira,

```
<exp> ::= <exp> '+' <termo>
```

Vamos lembrar que em um analisador descendente recursivo, cada não-terminal é reconhecido por uma função, e sempre que esse não-terminal aparece em uma produção, chamamos a função associada a ele. Para reconhecer a regra acima, escrevemos uma função que começa da seguinte forma:

```
exp():
  e1 = exp()
  ...
```

A primeira coisa que a função faz é chamar a si mesma, o que obviamente resulta em um *loop* infinito. A regra apresenta *recursividade à esquerda*, e esse tipo de regra é um problema para qualquer algoritmo de análise sintática descendente.

Felizmente, existe um algoritmo para retirar a recursividade à esquerda de uma gramática livre de contexto. Aplicando essa transformação na gramática de expressões, obtemos:

```
<exp> ::= <termo> <termo_aux>
<termo_aux> ::= '+' <termo> <termo_aux> | '-' <termo> <termo_aux> |
<termo> ::= <fator> <fator_aux>
<fator_aux> ::= '*' <fator> <fator_aux> | '/' <fator> <fator_aux> |
<fator> ::= <num> | '(' <exp> ')'
```

A transformação adiciona dois não-terminais à gramática: `<termo_aux>` e `<fator_aux>`. Note que ambos incluem uma produção vazia.

Vamos fazer outra transformação para deixar a gramática mais simples para análise. `<termo_aux>` e `<fator_aux>` representam zero ou mais repetições de um operador seguido de `<termo>` ou `<fator>`. Usando o operador `*` para representar zero ou mais repetições (assim como nas Expressões Regulares), ficamos com a seguinte gramática:

```
<exp>   ::= <termo> (('+' | '-') <termo>)*
<termo> ::= <fator> (('*' | '/') <fator>)*
<fator> ::= <num> | '(' <exp> ')'
```

Vamos fazer uma última mudança, de natureza mais cosmética: mudar os nomes dos não-terminais. Isso vai ser mais importante quando formos adicionar mais operadores e mais níveis de precedência. Os três níveis serão chamados de `<exp_a>` para expressões aditivas, `<exp_m>` para expressões multiplicativas, e `<prim>` para expressões primárias (ao invés de fator):

```
<exp_a> ::= <exp_m> (('+' | '-') <exp_m>)*
<exp_m> ::= <prim>  (('*' | '/') <prim>)*
<prim>  ::= <num> | '(' <exp_a> ')'
```

Essa gramática pode ser usada para criar um analisador sintático que pode reconhecer expressões aritméticas que levam em consideração as regras de precedência e associatividade dos operadores.

1.2 Associatividade

Talvez você tenha pensado que uma solução para a recursividade à esquerda em uma produção como `<exp> ::= <exp> '+' <termo>` seria inverter a ordem para

```
<exp> ::= <termo> '+' <exp>
```

Essa produção não é mais recursiva à esquerda, mas cria um problema com a associatividade dos operadores. Expressões como 7 + 5 + 3 e 10 - 8 - 2 também são ambíguas, pois é possível efetuar primeiro o operador mais à esquerda ou o outro, mais à direita. Na soma, o resultado final será o mesmo, em qualquer ordem. Mas na subtração a ordem muda o resultado: (10 - 8) - 2 = 0 enquanto 10 - (8 - 2) = 4.

Para esses casos usamos a convenção da associatividade, e nos quatro operadores básicos a associatividade é à esquerda, ou seja, em uma sequência de operadores de mesma precedência, a ordem deve ser da esquerda para a direita. Por essa convenção, 7 + 5 + 3 é o mesmo que (7 + 5) + 3, e 10 - 8 - 2 é o mesmo que (10 - 8) - 2.

A produção `<exp> ::= <termo> '+' <exp>` gera árvores com associatividade à direita, e por isso não pode ser utilizada. A gramática transformada que vimos acima (com `exp_a`, `exp_m` e `prim`) trata corretamente a associatividade de todos os operadores.

2 Implementação da gramática EC2

O objetivo desta atividade é mudar o analisador sintático para permitir que as expressões sejam escritas com menos parênteses, usando as regras de precedência e associatividade. Considerando isso, vimos que a gramática da linguagem EC2 deve ser:

```
<exp_a> ::= <exp_m> (('+' | '-') <exp_m>)*
<exp_m> ::= <prim>  (('*' | '/') <prim>)*
<prim>  ::= <num> | '(' <exp_a> ')'
```

Nessas regras, `<num>` é uma constante inteira.

O analisador sintático descendente recursivo terá três funções, uma para cada não-terminal. Mas as produções para `<exp_a>` e `<exp_m>` incluem o operador de repetição, *. Para implementar isso, é preciso usar algum *loop* na função correspondente.

Para `<exp_a>`, a função deve iniciar reconhecendo um `<exp_m>` (chamando a função correspondente). Em seguida vem a repetição: se o próximo *token* na entrada for + ou -, deve ser reconhecido outro `<exp_m>` e verificar novamente o *token* seguinte.

No momento de verificar se o próximo *token* é um operador, é importante não retirar esse *token* do fluxo de entrada, pois caso o *token* não seja um dos operadores esperados, esse *token* deve permanecer no fluxo de entrada, para ser visto por outra função de análise.

O que fazer quando houver um operador de soma e outro `exp_m`? Como as operações são associativas à esquerda, é possível já construir um nó de operação binária com os dois `exp_m` analisados e isso será o operando esquerdo de um próximo operador, se existir.

Por exemplo, seja a expressão 7 + 5 + 3. Na análise de `exp_a`, a constante 7 vai ser analisada como `exp_m`, e como o próximo *token* na entrada é um operador de soma, o analisador vai avançar o operador e analisar outra `exp_m`, que é a constante 5. Nesse ponto, a soma 7 + 5 será o operando esquerdo do próximo operador de soma. Portanto, podemos construir o nó da árvore de operação binária para 7 + 5 e continuar a análise.

Essas ideias estão resumidas no seguindo pseudo-código para a função de análise de `exp_a`:

```
exp_a():
  esq = exp_m()
  tok = olharProximoToken()
  while tok == '+' OR tok == '-':
    avancaToken()
    dir = exp_m()
    if tok == '+':
      operador = SOMA
    else:
      operador = SUB

    esq = BinOp(operador, esq, dir)
    tok = olharProximoToken()

  return esq
```

A função `olharProximoToken` é a função que verifica qual o próximo *token* sem avançar a entrada. Se o *token* for um operador esperado, `avancaToken` vai avançar o fluxo de entrada para que `exp_m` funcione corretamente.

Como a produção para `exp_m` tem a mesma forma da produção para `exp_a`, apenas trocando `exp_m` por `prim` e os operadores, a função para `exp_m` também terá a mesma forma da função para `exp_a`.

2.1 Eficiência

Além do problema com produções recursivas à esquerda, os analisadores descendentes recursivos têm um problema de eficiência na parte das expressões, causado pelo grande número de chamadas de função que são necessárias para fazer a análise sintática.

Por exemplo, suponha que o programa de entrada seja apenas uma constante inteira. Na gramática EC2, a análise vai resultar em três chamadas de função: a função para `exp_a` inicia a análise e chama a função para `exp_m`, que por sua vez vai chamar a função `prim`, que vai reconhecer a constante.

Nesse caso, três chamadas não parece muito, mas à medida que adicionamos mais operadores e mais níveis de precedência, o número de chamadas de função cresce rapidamente. Não é incomum que linguagens de programação tenham operadores em 10 ou mais níveis de precedência. Nesse caso, para reconhecer apenas uma constante o analisador precisa usar 10 ou mais chamadas de função. Isso resulta em analisadores sintáticos menos eficientes do que seria possível com outras técnicas.

Por isso, os compiladores que utilizam analisadores recursivos descendentes não fazem a análise sintática das expressões da mesma forma. O mais comum é trocar para outro algoritmo

de análise sintática para as expressões, geralmente alguma variação de análise de precedência de operadores (*operator-precedence parsing*). O algoritmo de escalada de precedência (*precedence climbing*) é uma opção comum, usado por exemplo no clang (compilador C/C++ baseado em LLVM e usado pela Apple).

3 Artefato para entrega

Cada grupo deve entregar o compilador para a linguagem EC2, que aceita expressões sem parênteses obrigatórios e segue corretamente as convenções de precedência e associatividade dos operadores. O projeto deve incluir testes que demonstram claramente a capacidade de reconhecer expressões sem parênteses. A geração de código não muda com relação à Atividade 07.