Interpreter

Padrões Comportamentais

O padrão *Interpreter* busca definir uma representação para a gramática de uma determinada linguagem, juntamente com um interpretador que usa tal representação para interpretar sentenças dessa linguagem.

Motivação (Por que utilizar?)

Se um determinado tipo de problema ocorre com muita frequência, pode valer a pena expressar instâncias do problema como sentenças de uma linguagem simples. Em seguida, pode-se criar um interpretador que resolva tal problema interpretando essas sentenças.

Como exemplo iremos criar um interpretador de **notação polonesa inversa** (ou RPN na sigla em inglês, de Reverse Polish Notation). Tal notação define expressões matemáticas onde os operadores sucedem seus operandos, por exemplo, para somar 3 e 4, a expressão é escrita **3 4 +** ao invés de **3 + 4**. Se existirem várias operações, os operadores sempre vêm imediatamente após seu segundo operando, portanto, a expressão escrita **3 - 4 + 5** na notação convencional seria escrita **3 4 - 5 +** na notação polonesa inversa: Primeiro 4 é subtraído de 3, então 5 é somado ao resultado.

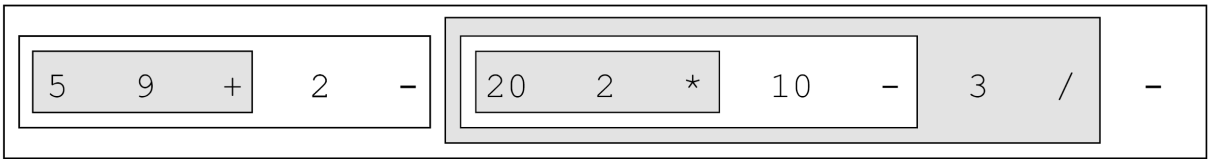
A. 5 5 +
B. 3 4 -
C. 5 9 + 2 - 20 2 * 10 - 3 / -

Exemplos de expressões seguindo a notação polonesa inversa

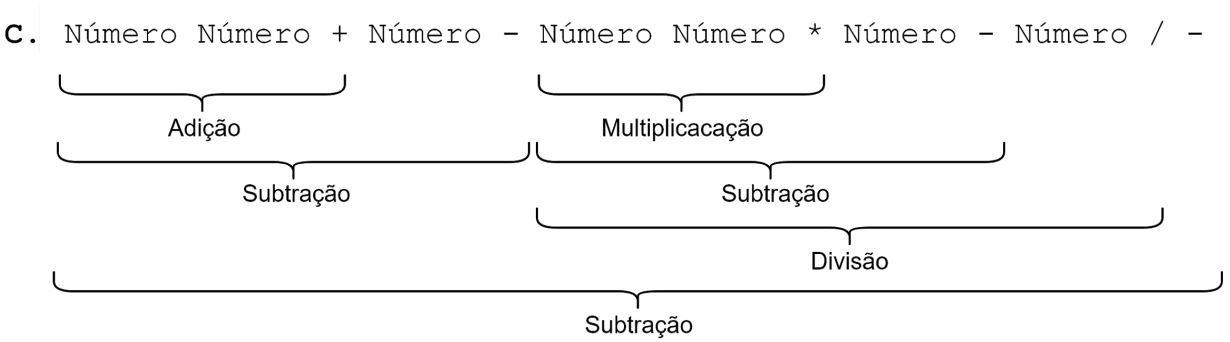
Repare nas expressões acima que sempre existe um padrão.

A. Número Número + (É uma adição)
B. Número Número - (É uma subtração)

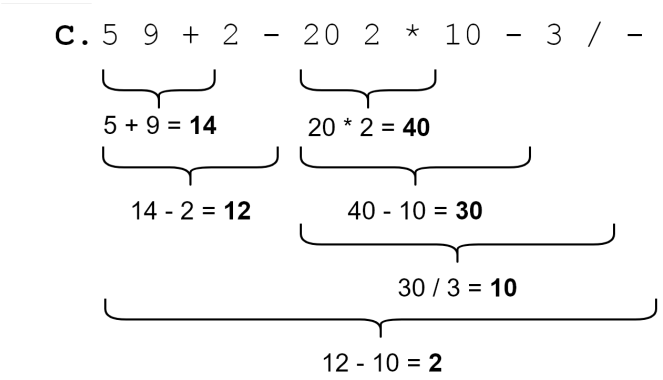
A expressão **C** é uma expressão composta por várias expressões, observe:



Expressão composta por outras expressões



O cálculo completo da expressão **C** fica da seguinte forma:



Conforme diz a definição, o padrão *Interpreter* busca definir uma representação para a gramática de uma determinada linguagem. Podemos considerar que as expressões aceitas por uma calculadora de notação polonesa inversa que realiza operações de adição, subtração, multiplicação e divisão fazem parte de uma linguagem simples.

É importante ressaltar que estamos tratando de linguagens no contexto da teoria da computação. De maneira informal, entenda as linguagens como uma forma de comunicação. Trata-se de um conjunto de elementos (símbolos) e um conjunto de métodos (regras) para combinar estes elementos, que são usados e entendidos por uma determinada comunidade. São exemplos as "linguagens naturais" (ou idiomas), "linguagens de programação" e os "protocolos de comunicação". A definição formal de uma linguagem exige uma série de conceitos de base que não serão abordados aqui.

Para um maior entendimento do padrão *Interpreter* é interessante que se conheça o conceito de gramáticas. **Uma gramática é um conjunto de regras de produção de cadeias em uma linguagem formal, ou seja, um objeto que permite especificar uma linguagem.**

A definição acima não é simples, por isso vamos por partes. Para isso considere a gramática a seguir que define a linguagem de nossa calculadora:

```
EXPRESSAO      ::= ADICAO | SUBTRACAO | MULTIPLICACAO | DIVISAO | VARIAVEL | NUMERO
ADICAO         ::= EXPRESSAO EXPRESSAO '+'
SUBTRACAO      ::= EXPRESSAO EXPRESSAO '-'
MULTIPLICACAO ::= EXPRESSAO EXPRESSAO '*'
DIVISAO        ::= EXPRESSAO EXPRESSAO '/'
NUMERO         ::= DIGITO | DIGITONUMERO
VARIAVEL       ::= 'a' | 'b' | 'c' | ... | 'z'
DIGITO         ::= '0' | '1' | '2' | ... | '9'
```

O lado esquerdo de '::=' na gramática acima nomina as regras de produção. Elas dizem que:

1. Uma EXPRESSAO pode ser qualquer combinação de ADICAO, SUBTRACAO, MULTIPLICACAO, DIVISAO, VARIAVEL, ou NUMERO.

2. Uma ADICAO sempre será uma EXPRESSAO, seguida de outra EXPRESSAO e logo depois um **'+'**.

3. Uma SUBTRACAO sempre será uma EXPRESSAO, seguida de outra EXPRESSAO e logo depois um **'-'**.

4. Uma MULTIPLICACAO sempre será uma EXPRESSAO, seguida de outra EXPRESSAO e logo depois um **'*'**.

5. Uma DIVISAO sempre será uma EXPRESSAO, seguida de outra EXPRESSAO e logo depois um **'/'**.

6. Um NUMERO sempre será um DIGITO ou uma combinação deles.

7. Uma VARIAVEL sempre será um caractere de a-z.

8. Um DIGITO sempre será um caractere de 1-9.

As Regras de produção são compostas por símbolos terminais e não terminais.

Símbolos Terminais: não podem ser quebrados em unidades menores. De forma simples, ao chegar em um símbolo terminal não tem mais para onde ir. No caso de nossa gramática, as variáveis (a, b, c … z), dígitos (1, 2, 3 … 9) e operadores (+, -, *, /) são símbolos terminais.

Símbolos não Terminais: São os símbolos que podem ser substituídos. Ao chegar em um símbolo não terminal é possível seguir para outra regra da gramática. Um símbolo EXPRESSAO pode ser substituído por uma ADICAO, SUBTRACAO, MULTIPLICACAO, DIVISAO, VARIAVEL ou NUMERO. Ao encontrar DIVISAO não

podemos alterar a barra "/" que é um símbolo terminal mas podemos trocar as expressões. Por exemplo:

```
DIVISAO =  EXPRESSAO EXPRESSAO /
           NUMERO NUMERO /
           NUMERO VARIAVEL /
           ADICAO NUMERO /
           DIVISAO SUBTRACAO /
```

Um NUMERO pode ser trocado por um DIGITO ou então por um DIGITO seguido de outro NUMERO. Se quisermos criar o número 5 por exemplo, basta trocar o NUMERO pelo DIGITO 5:

```
NUMERO = DIGITO = 5
```

Caso precisássemos do número 125 teríamos o seguinte:

```
NUMERO =   DIGITONUMERO = 1NUMERO
           DIGITODIGITONUMERO = 12NUMERO
           DIGITODIGITODIGITO = 125
```

Sempre que existir um NUMERO na cadeia ele poderá ser substituído por um DIGITO ou por um dígito seguido de outro número DIGITONUMERO. Por ser um símbolo não terminal, enquanto existir um NUMERO na cadeia será possível adicionar ao menos 1 DIGITO.

O símbolo EXPRESSAO é o símbolo inicial. A interpretação de toda regra de produção da nossa gramática começa a partir da regra EXPRESSAO.

Para ilustrar vamos alterar um pouco a expressão **C** inserindo duas variáveis x e y onde x = 2 e y = 3. Tais variáveis irão compor o contexto da interpretação. Ao alterar o valor de uma variável estaremos alterando o contexto.

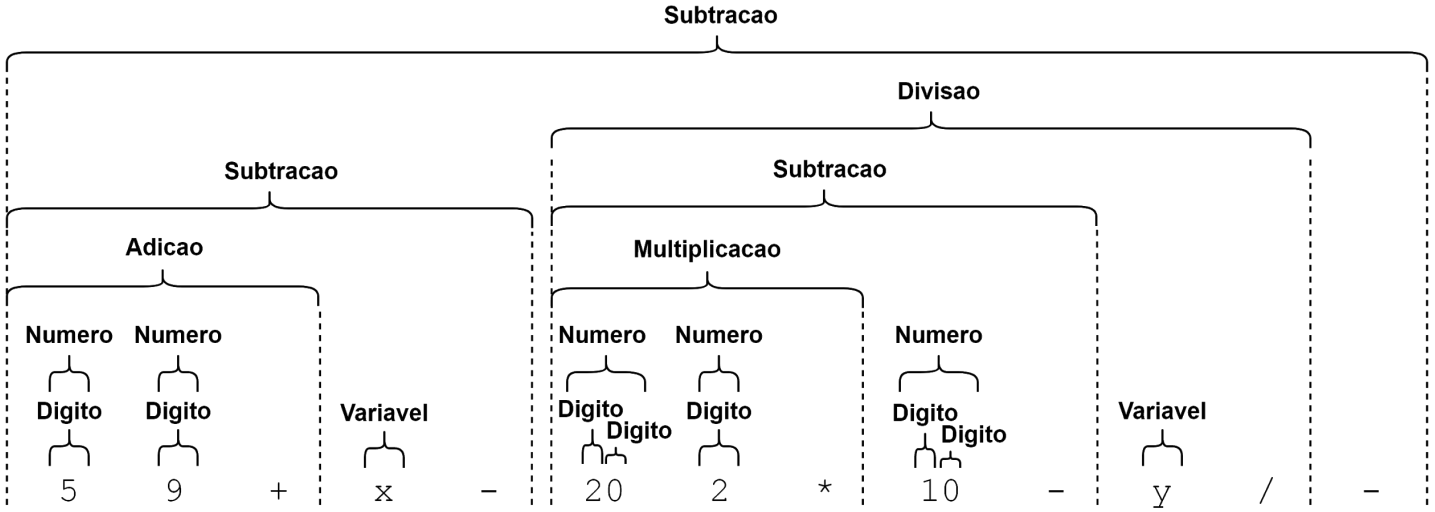
```
5 9 + x - 20 2 * 10 - y / -
```

Expressao C com variáveis

De acordo com a regra 1 da gramática acima, um NUMERO é uma EXPRESSAO da mesma forma que uma ADICAO também é uma EXPRESSAO.

Pela regra 2, ADICAO é uma EXPRESSAO seguida de outra EXPRESSAO e logo depois um **+**, deste modo podemos ter uma ADICAO no formato NUMERO NUMERO **+** já que NUMERO é uma EXPRESSAO. Por isso podemos ter 5 9 + como parte da expressão **C**.

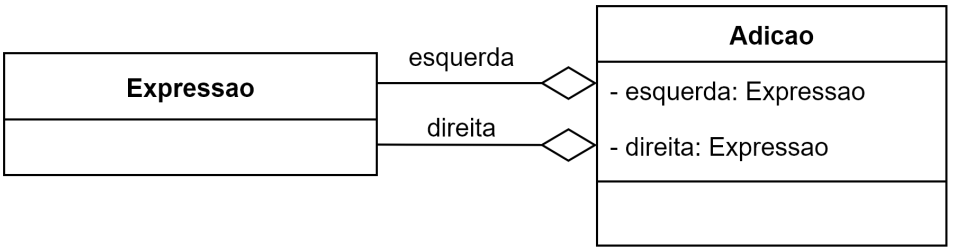
Seguindo as regras da gramática a expressão **C** seria interpretada da seguinte forma:



Interpretação da expressão C de acordo com a gramática fornecida

O padrão *Interpreter* nos diz que devemos usar uma classe para definir cada regra de produção da gramática. Temos 8 regras de produção, portanto teremos ao menos 8 classes já que esse número pode crescer caso seja feita alguma subclassificação.

De modo geral os símbolos não terminais presentes no lado direito de uma regra de produção serão variáveis de instância (atributos) na classe da regra. Por exemplo a regra **ADICAO ::= EXPRESSAO EXPRESSAO '+'** possui dois símbolos não terminais do lado direito, duas expressões, portando a classe Adicao poderia ser como a seguir:



Exemplo de classe para a regra ADICAO

Repare que a classe Adicao possui dois atributos do tipo Expressao que por sua vez também é uma classe. Embora essa configuração de classe seja comum, não é uma regra. Em nosso exemplo a regra Expressao será representada por uma interface. Mesmo que existam diversos símbolos não terminais do lado direito da regra de de produção EXPRESSAO nossa interface Expressao não terá atributos, já que em PHP isso nem seria possível.

Veja a seguir o diagrama de classes da implementação do padrão interpreter para nossa gramática.

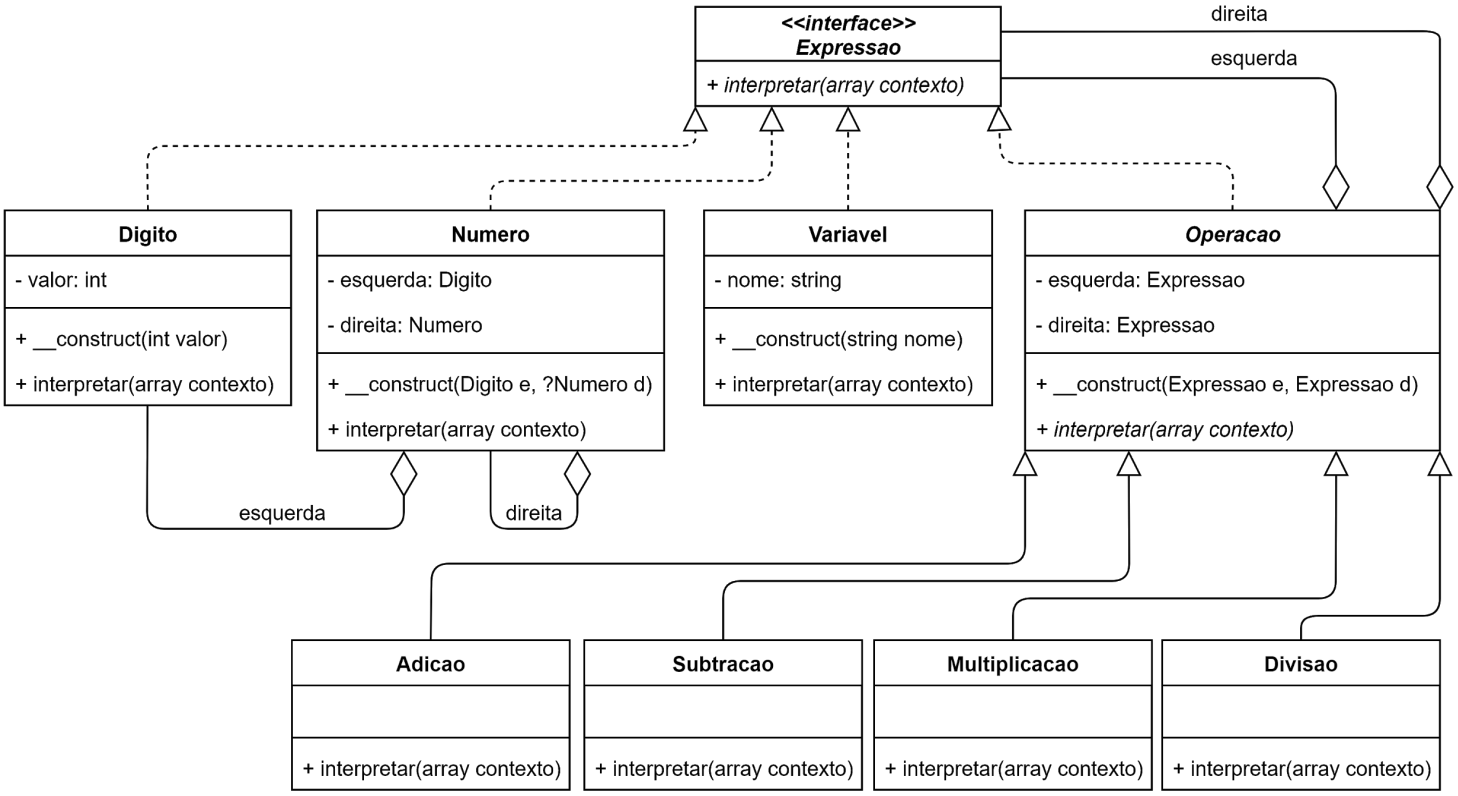


Diagrama de classes da gramática da calculadora utilizando o padrão *Interpreter*

O papel da interface **Expressao** é fornecer um supertipo e forçar que todas as classes de tal supertipo implementem o método **interpretar()**.

```
interface Expressao
{
    public function interpretar(array $contexto): int;
}
```

Em nossa gramática todas as operações são realizadas em dois operandos, no caso duas expressões. Em nosso código as expressões são representadas pela interface **Expressao**. Para que possamos fazer reutilização de código vamos criar a classe abstrata **Operacao** que manterá referência às duas expressões de uma operação.

```php
abstract class Operacao implements Expressao
{
    protected Expressao $esquerda; //Expressão da esquerda
    protected Expressao $direita; //Expressão da direita

    //As duas expressões são esperadas no construtor de uma operação
    public function __construct(Expressao $esquerda, Expressao $direita)
    {
        $this->esquerda = $esquerda;
        $this->direita = $direita;
    }

    /*O método interpretar se mantém abstrato para que as subclasses
    de operação o implemente*/
    abstract public function interpretar(array $contexto): int;
}
```

Temos a classe operação, agora vamos implementar as operações propriamente ditas, que serão subclassses de **Operacao**, portanto, também serão do tipo **Expressao** definido por nossa interface.

O método **interpretar()** de **Adicao** retornará a **Expressao** da esquerda somada a **Expressao** da direita.

```php
class Adicao extends Operacao
{
    public function interpretar(array $contexto): int
    {
        return $this->esquerda->interpretar($contexto) +
               $this->direita->interpretar($contexto);
    }
}
```

O método **interpretar()** de **Subtracao** retornará a **Expressao** da esquerda subtraída da **Expressao** da direita.

```php
class Subtracao extends Operacao
{
    public function interpretar(array $contexto): int
    {
        return $this->esquerda->interpretar($contexto) -
               $this->direita->interpretar($contexto);
    }
}
```

O método **interpretar()** de **Multiplicacao** retornará a **Expressao** da esquerda multiplicada pela **Expressao** da direita.

```php
class Multiplicacao extends Operacao
{
    public function interpretar(array $contexto): int
    {
        return $this->esquerda->interpretar($contexto) *
               $this->direita->interpretar($contexto);
    }
}
```

Por fim, o método **interpretar()** de **Divisao** retornará a parte inteira da **Expressao** da esquerda dividida pela **Expressao** da direita.

```php
class Divisao extends Operacao
{
    public function interpretar(array $contexto): int
    {
        return intval($this->esquerda->interpretar($contexto) /
               $this->direita->interpretar($contexto));
    }
}
```

Digito, **Numero** e **Variavel** também são expressões mas não são operações. Deste modo, elas implementam diretamente a interface **Expressao** já que não são subclasses de **Operacao**.

```php
class Digito implements Expressao
{
    private string $digito;

    //Espera uma string no construtor
    public function __construct(string $digito)
    {
        $this->digito = $digito;
    }

    //Somente digitos de 0 a 9 são aceitos
    private function validarDigito()
    {
        $digitos = [
            '0', '1', '2', '3', '4',
            '5', '6', '7', '8', '9'
        ];

        return in_array($this->digito, $digitos);
    }

    public function interpretar(array $contexto): int
    {
        //Antes de interpretar valida o dígito.
        if ($this->validarDigito()) {
            //É uma expressão terminal, então apenas retorna seu próprio valor.
            //Faz a conversão explícita para int.
            return intval($this->digito);
        }

        //Lança uma exceção caso a variável não esteja definida no contexto.
        throw new \Exception('Todo dígito deve ser um inteiro entre 0 e 9');
    }
}
```

```php
class Numero implements Expressao
{
    //Todo número tem ao menos um dígito
    private Digito $esquerda;

    //Caso um número tenha mais de um dígito
    //ele será composto por outro Numero.
    private ?Numero $direita;

    //Pode receber uma ou duas expressões em seu construtor.
    //Digito e Numero são do supertipo Expressao
    public function __construct(Digito $esquerda, ?Numero $direita = null)
    {
        $this->esquerda = $esquerda;
        $this->direita = $direita;
    }

    public function interpretar(array $contexto): int
    {
        //Se a direita for vazia (numero de apenas um dígito)
        if (is_null($this->direita)) {
            //Interpreta somente o lado esquerdo
            return $this->esquerda->interpretar($contexto);
        }

        //Interpreta o lado esquerdo e o concatena a interpretação do lado direito.
        //Retorna o resultado da concatenação convertido em inteiro.
        return intval(
            $this->esquerda->interpretar($contexto) .
            $this->direita->interpretar($contexto)
        );
    }
}
```

```php
class Variavel implements Expressao
{
    private string $variavel;

    //Espera uma string no construtor
    public function __construct(string $variavel)
    {
        $this->variavel = $variavel;
    }

    //Somente uma letra de a-z é aceita
    private function validarVariavel()
    {
        $alfabeto = [
            'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
            'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'
        ];
        return in_array($this->variavel, $alfabeto);
    }

    public function interpretar(array $contexto): int
    {
        //Se a variável for válida e estiver definida no contexto.
        if ($this->validarVariavel() && isset($contexto[$this->variavel])) {
            //Retorna o valor atribuído à variável no contexto.
            return $contexto[$this->variavel];
        }

        //Lança uma exceção caso a variável não esteja definida no contexto.
        throw new \Exception(
            'A variável precisa ser uma letra de a-z e estar definida no contexto!'
        );
    }
}
```

A definição do padrão *Interpreter* nos diz que *"O padrão Interpreter busca definir uma representação para a gramática de uma determinada linguagem, juntamente com um interpretador que usa tal representação para interpretar sentenças dessa linguagem"*. As classes que criamos nos fornecem a representação e as chamadas recursivas ao método **interpretar()** nos fornece o interpretador da representação.

A tabela a seguir relaciona as regras da gramática com as classes que as representam. Toda vez que chegarmos em uma dessas regras uma instância apropriada de uma das classes da representação será criada. Por exemplo, ao chegar na regra **ADICAO** criaremos um objeto da classe **Adicao** que precisa de duas expressões em seu construtor.

ADICAO ::= EXPRESSAO EXPRESSAO '+'

é equivalente a

new Adicao(**EXPRESSAO, EXPRESSAO**);

Onde **EXPRESSAO** deve ser uma instância de qualquer classe que implemente a interface **Expressao**.

Regra	Criação de instância de representação
EXPRESSAO	Sem instância já que Expressao é uma interface.
ADICAO	new Adicao(**EXPRESSAO**, **EXPRESSAO**);
SUBTRACAO	new Subtracao(**EXPRESSAO**, **EXPRESSAO**);
MULTIPLICACAO	new Multiplicacao(**EXPRESSAO**, **EXPRESSAO**);
DIVISAO	new Divisao(**EXPRESSAO**, **EXPRESSAO**);
NUMERO	new Numero(**DIGITO**); **OU** new Numero(**DIGITO**, **NUMERO**);
VARIAVEL	new Variavel(string);
DIGITO	new Digito(string);

Equivalências entre regras da gramática e suas respectivas classes

Antes de irmos para o teste, vejamos como representar uma expressão por meio das classes. Tomando a expressão **5 10 +** podemos ter a seguinte representação:

Começando pelo **5**:

5	10	+

Olhando para as regras da gramática:

```
EXPRESSAO     ::= ADICAO | SUBTRACAO | MULTIPLICACAO | DIVISAO | VARIAVEL | NUMERO
NUMERO        ::= DIGITO | DIGITONUMERO
DIGITO        ::= '0' | '1' | '2' | ... | '9'
```

Iniciamos em **EXPRESSAO**, por 5 ser um numero vamos para a regra **NUMERO**. De acordo com a tabela de equivalências ao chegar na regra **NUMERO** devemos criar uma instância da classe **Numero**.

```
$arvore = new Numero(DIGITO);
```

Agora precisamos determinar o **Digito** que é requisitado pelo construtor da classe **Numero**. Tal instância é alcançada por meio da regra **DIGITO** da gramática. 5 é um número de apenas 1 dígito então podemos ir direto para a regra **DIGITO**. Segundo a tabela de equivalências devemos criar uma instância de **Digito**.

```
$arvore = new Numero(new Digito(5));
```

Indo para o **10**:

5	**10**	+

Olhando para as regras da gramática:

```
EXPRESSAO     ::= ADICAO | SUBTRACAO | MULTIPLICACAO | DIVISAO | VARIAVEL | NUMERO
NUMERO        ::= DIGITO | DIGITONUMERO
DIGITO        ::= '0' | '1' | '2' | ... | '9'
```

Iniciamos em **EXPRESSAO**, por 10 ser um número vamos para a regra **NUMERO**. De acordo com a tabela de equivalências ao chegar na regra **NUMERO** devemos criar mais uma instância da classe Numero.

```
$arvore = new Numero(new Digito(5))
          new Numero(DIGITO, NUMERO);
```

A sintaxe do nosso código está errada mas não se preocupe com isso por enquanto.

Repare que para o número 10 nós caímos na segunda opção da regra **NUMERO**, que é **DIGITONUMERO**. Nessa regra primeiro resolvemos **DIGITO** e depois ainda precisaremos resolver o **NUMERO** que fica faltando. Tal regra é utilizada para produzir números com mais de um dígito, como é o caso do número 10. Enquanto existir uma forma de voltar para a regra **NUMERO** sempre será possível produzir um novo dígito.

Precisamos determinar a instância do primeiro Digito que é requisitada pelo construtor da classe Numero já instanciada. Tal instância é alcançada por meio da regra **DIGITO** da gramática.

```
$arvore = new Numero(new Digito(5))
          new Numero(new Digito(1), NUMERO);
```

Voltando para a gramática, ainda precisamos resolver o **NUMERO** que ficou pendente. No momento temos algo parecido com **1NUMERO** e precisamos do número **10**. Como não precisamos de nenhum outro dígito depois do **0** podemos a partir de **NUMERO** de ir direto para **DIGITO**.

```
EXPRESSAO     ::= ADICAO | SUBTRACAO | MULTIPLICACAO | DIVISAO | VARIAVEL | NUMERO
NUMERO        ::= DIGITO | DIGITONUMERO
DIGITO        ::= '0' | '1' | '2' | ... | '9'
```

O dígito 0 deve compor o número 10. Por isso deve ser o segundo parâmetro da instância de Numero que já criamos, repare que a classe Numero possui um atributo **$direita** que é do tipo Numero e é opcional.

Passamos novamente pela regra **NUMERO**, portanto, uma nova instância de Numero deve ser criada como segundo parâmetro da primeira instância de Numero.

```
$arvore = new Numero(new Digito(5))
          new Numero(new Digito(1), new Numero(DIGITO));
```

Resolvendo o **DIGITO**:

```
$arvore = new Numero(new Digito(5))
          new Numero(new Digito(1), new Numero(new Digito(0)));
```

Por fim vamos para a adição:

5	10	+

Olhando para as regras da gramática:

EXPRESSAO ::= **ADICAO** | SUBTRACAO | MULTIPLICACAO | DIVISAO | VARIAVEL | NUMERO
ADICAO ::= EXPRESSAO EXPRESSAO '+'

Iniciamos em **EXPRESSAO**, por se tratar de uma adição caímos na regra **ADICAO**. De acordo com a tabela de equivalência tal regra implica na criação de uma instância da classe Adicao que tem como requisitos duas expressões para ser criada. A classe Numero implementa a interface Expressao, então instâncias de Numero podem ser passadas para o construtor de Adicao.

```
$arvore = new Adicao(
              new Numero(new Digito(5)),
              new Numero(new Digito(1), new Numero(new Digito(0)))
          );
```

Agora a sintaxe do nosso código está correta. Vamos pedir para que a árvore acima seja interpretada. Para isso precisamos de um contexto, como neste exemplo não utilizamos variáveis vamos criar um contexto vazio.

```
$arvore = new Adicao(
              new Numero(new Digito(5)),
              new Numero(new Digito(1), new Numero(new Digito(0)))
          );

$contexto = []; //O contexto é um array que define quanto vale cada variável.
echo $arvore->interpretar($contexto); //Início da recursão de interpretação.
```

Saída:

```
15
```

Essa representação por meio de classes que criamos para nossa expressão é uma **árvore sintática abstrata**. Ela verifica se a expressão em questão está montada de forma correta, ou seja, verifica a sintaxe da expressão.

O padrão *Interpreter* não explica como criar uma árvore sintática abstrata, em outras palavras, não verifica sintaxe. Tal árvore pode ser criada diretamente pelo cliente (como foi feio para expressão acima) ou gerada por meio de um *parser*.

Vamos realizar um teste um pouco mais elaborado, agora com a expressão **C** a qual já analisamos anteriormente. Por ser uma expressão maior do que a que acabamos de ver, teremos também uma árvore sintática maior que foi construída com base nas regras da gramática.

C: 5 9 + x - 20 2 * 10 - y / -

```
//Criação manual da árvore sintática.
$arvore = new Subtracao(
                new Subtracao(
                    new Adicao(
                        new Numero(new Digito('5')),
                        new Numero(new Digito('9'))
                    ),
                    new Variavel('x')
                ),
                new Divisao(
                    new Subtracao(
                        new Multiplicacao(
                            new Numero(new Digito(2)), new Numero(new Digito(0))),
                            new Numero(new Digito(2))
                        ),
                        new Numero(new Digito(1), new Numero(new Digito(0)))
                    ),
                    new Variavel('y')
                )
            );

//O contexto é um array que define quanto vale cada variável.
$contexto = ['x' => 2, 'y' => 3];

//Início da recursão de interpretação.
echo $arvore->interpretar($contexto);
```

Saída:

```
2
```

No exemplo acima a árvore de análise sintática abstrata também foi gerada pelo cliente. **Considere a próxima classe como sendo um bônus,** nela criaremos um *parser* que irá gerar automaticamente a árvore de análise sintática abstrata de qualquer expressão fornecida, desde que ela obedeça a notação polonesa inversa.

```
class Parser
{
    private array $pilha = [];
    private string $expressao;

    public function __construct(string $expressao)
    {
        $this->expressao = trim($expressao);
    }

    private function isOperacao(string $token): bool
    {
        $operacoes = ['+', '-', '*', '/'];
        return in_array($token, $operacoes);
    }

    private function parseNumero(string $token): Numero
    {
        $pilhaNumero = [];
        $tamanhoOperando = strlen($token);
        $direita = null;

        for ($i = 0; $i < $tamanhoOperando; $i++) {
            $parteToken = substr($token, $i, 1);

            if ($i == $tamanhoOperando - 1) {
                $pilhaNumero[] = new Digito($parteToken);

                while (count($pilhaNumero) > 0) {
                    $topoPilha = end($pilhaNumero);

                    if ($topoPilha instanceof Digito) {
                        if (is_null($direita)) {
                            $direita = new Numero($topoPilha);
                            array_pop($pilhaNumero);
                        } else {
                            $direita = new Numero($topoPilha, $direita);
                            array_pop($pilhaNumero);
                        }
                    }
                }
            } else {
                $pilhaNumero[] = new Digito($parteToken);
            }

        }

        return $direita;
    }
```

```
    public function parse(): Expressao
    {
        $expressao = explode(' ', $this->expressao);

        foreach ($expressao as $token) {
            if ($this->isOperacao($token)) {
                $operandoDireita = end($this->pilha);
                array_pop($this->pilha);
                $operandoEsquerda = end($this->pilha);
                array_pop($this->pilha);

                switch ($token) {
                    case '+':
                        $this->pilha[] = new Adicao($operandoEsquerda, $operandoDireita);
                        break;
                    case '-':
                        $this->pilha[] = new Subtracao($operandoEsquerda, $operandoDireita);
                        break;
                    case '*':
                        $this->pilha[] = new Multiplicacao($operandoEsquerda, $operandoDireita);
                        break;
                    case '/':
                        $this->pilha[] = new Divisao($operandoEsquerda, $operandoDireita);
                        break;
                }
            } elseif (is_numeric($token)) {
                $this->pilha[] = $this->parseNumero($token);
            } else {
                $this->pilha[] = new Variavel($token);
            }
        }
        return end($this->pilha);
    }
}
```

Com o parser o mesmo exemplo de antes ficaria da seguinte forma:

```
//O parser é criado recebendo a expressão em seu construtor.
$parser = new Parser('9 5 + x - 20 2 * 10 - y / -');

//A árvore sintática abstrata é gerada automaticamente pelo parser.
$arvore = $parser->parse();

//O contexto é um array que define quanto vale cada variável.
$contexto = ['x' => 2, 'y' => 3];

//Início da recursão de interpretação.
echo $arvore->interpretar($contexto);
```

Saída:

```
2
```

Aplicabilidade (Quando utilizar?)

O padrão *Interpreter* pode ser utilizado quando existe uma linguagem para interpretar e é possível representar instruções da linguagem como árvores sintáticas abstratas. O *Interpreter* funciona melhor quando:

- A gramática é simples. Para gramáticas complexas, a hierarquia de classes da gramática se torna grande e incontrolável.

- Eficiência não é uma preocupação crítica. Os *Interpreters* mais eficientes geralmente não são implementados pela interpretação direta de árvores de análise sintática abstrata, mas pela tradução primeiro para outra forma.

Componentes

ExpressaoAbstrata: Declara um método abstrato *interpretar()* comum a todos os nós na árvore sintática abstrata.

ExpressaoNaoTerminal:

- É necessária uma classe desse tipo para cada regra R ::= R_1 R_2 .. .R_n da gramática.

- Mantém variáveis de instância do tipo ExpressaoAbstrata para cada um dos símbolos R_1 a R_n.

- Implementa um método *interpretar()* associado aos símbolos não terminais da gramática. O método *interpretar()* chama a si próprio recursivamente nas variáveis que representam R_1 a R_n.

ExpressaoTerminal:

- Implementa um método *interpretar()* associado aos símbolos terminais da gramática, ele define o caso base da recursão iniciada por uma ExpressaoNaoTerminal.

- É necessária uma instância para cada símbolo terminal em uma sentença.

Contexto: Contém informação que é global para o interpretador.

Cliente: Constrói ou recebe uma árvore sintática abstrata que representa uma determinada sentença da linguagem definida pela gramática. A Árvore sintática abstrata é montada a partir de instâncias das classes ExpressaoNaoTerminal e ExpressaoTerminal.

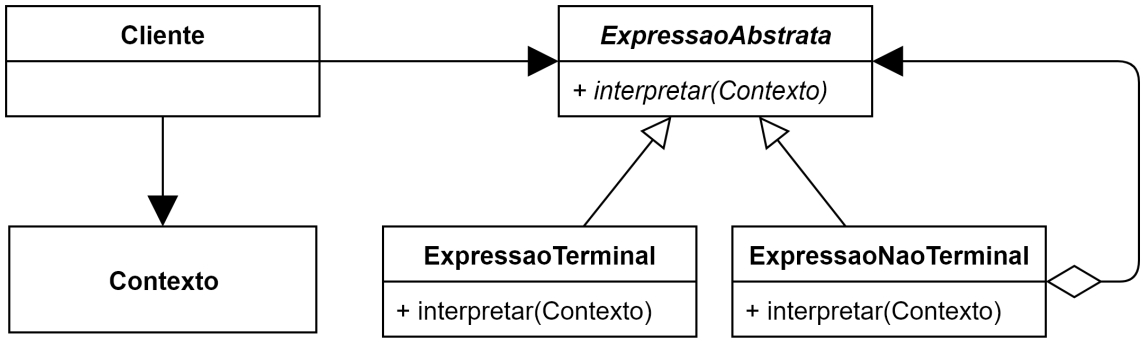


Diagrama de Classes

Consequências

- Se torna fácil alterar e estender a gramática. Como o padrão usa classes para representar regras gramaticais, é possível usar herança para alterar ou estender a gramática. As expressões existentes podem ser modificadas incrementalmente, e novas expressões podem ser definidas como variações das antigas.

- A implementação da gramática também fica fácil. As classes que definem os nós na árvore de sintaxe abstrata têm implementações semelhantes e são fáceis de escrever.

- Gramáticas complexas são difíceis de manter. O padrão *Interpreter* define pelo menos uma classe para cada regra da gramática. Portanto, gramáticas contendo muitas regras podem ser difíceis de gerenciar e manter.