Compiladores

Análise Sintática: *Parsing*

(Implementação do Padrão de Projeto *Visitor* para Expressões)

Vinicius H. S. Durelli
✉ durelli@ufsj.edu.br

Organização

Utilizando o padrão de projeto *Visitor*

➤ **Motivação**: por meio do padrão de projeto *Visitor* (Gamma et al. 1994) é possível **definir novas operações** para um conjunto de tipos **em um só lugar** (Nystrom 2021).

> Tudo isso sem alterar os tipos!

➥ No contexto da nossa implementação, o padrão será "refinado" um pouco: as operações que atuam sobre expressões, normalmente, retornam valores.

➥ Visto que não é possível assumir que todas as operações produzem o mesmo tipo de retorno, iremos empregar *generics* para possibilitar que cada operação retorne um tipo de resultado.

➤ Inicialmente, precisamos editar `GenerateAst` para introduzir a interface *Visitor* (novamente, tudo é inserido dentro da classe principal a fim de manter todas as definições em um único arquivo):

```
writer.println("abstract class " + baseName + " {");
...
defineVisitor(writer, baseName, types);
...
for (String type :  types) {
```

Introduzindo a implementação do padrão ao *script*

Introdução de um método `visit` para cada subclasse

➤ O código a seguir iterativamente introduz um método `visit` para cada subclasse:

```
private static void defineVisitor(
    PrintWriter writer, String baseName,
    List<String> types) {

  writer.println("  interface Visitor<R> {");

  for (String type : types) {
    String typeName = type.split(":")[0].trim();
    writer.println("    R visit" + typeName +
        baseName + "(" + typeName + " " +
        baseName.toLowerCase() + ");");
  }

  writer.println("  }");
}
```

Introduzindo a implementação do padrão ao *script*

➤ O método abstrato `accept` é definido dentro da classe base (`Expr`).

Alterando o método `defineAst`:

```
    defineType(writer, baseName, className, fields);
}
...
 // definicao do accept()
 writer.println();
 writer.println("  abstract <R> R accept" +
                "(Visitor<R> visitor);");
...
writer.println("}");
```

Aqui `R` é uma declaração de tipo genérico, algo como: uma forma de indicar que `R` pode ser substituido por qualquer tipo.

- O primeiro `<R>` indica que o método vai usar um tipo genérico,
- O segundo `R` indica que o método deve retornar um objeto do tipo genérico do método; e
- O terceiro `Visitor<R>` sinaliza que o tipo genérico de `Visitor` deve ser do mesmo tipo genérico do método.

Introduzindo a implementação do padrão ao *script*

Cada subclasse deve invocar o método `visit` apropriado

- Por fim, cada subclasse deve invocar o método `visit` que corresponde ao seu tipo. Tais modificações são feitas no método `defineType`:

```
writer.println(" }");
...
writer.println();
writer.println("    @Override");
writer.println("    <R> R accept(Visitor<R> " +
        "visitor) {");
writer.println("      return visitor.visit" +
        className + baseName + "(this);");
writer.println("    }")
...
writer.println();
for (String field :  fields) {
```

Implementando um *pretty printer* para auxiliar na depuração do *parser*

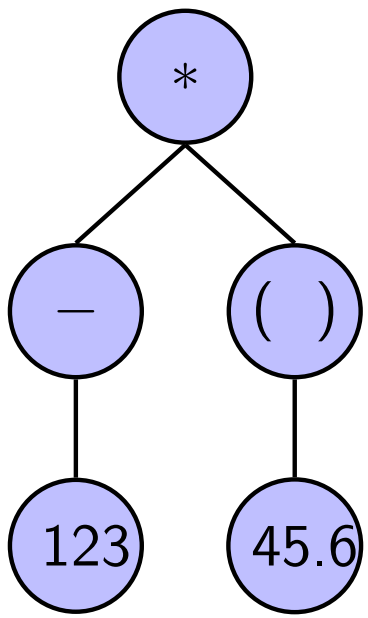
☛ **Para depurar o *parser* e o interpretador que serão implementados, é interessante visualizar a árvore sintática a fim de verificar se a mesma apresenta a estrutura esperada.**

➤ Dado uma árvore sintática conforme a apresentada abaixo.

O *pretty printer* deve exibir algo semelhante à linguagem Lisp:

- Expressões aparecem entre parênteses;
- Mais fácil para visualizar o aninhamento e *agrupamento* de subexpressões e tokens.

$$(* \ (- \ 123) \ (\mathit{group} \ 45.6))$$



Para tal, vamos utilizar o implementação do padrão de projeto *Visitor*...

➤ Começamos com uma (nova) classe (`AstPrinter`) que implementa a interface `Expr.Visitor`:

```
package edu.ufsj.lox;

class AstPrinter implements Expr.Visitor<String> {
  String print(Expr expr) {
    return expr.accept(this);
  }
}
```

Em seguida, é preciso implementar métodos *"visit"* para cada um dos tipos de expressões: *binary*, *grouping*, *literal* e *unary*.

```
@Override
public String visitBinaryExpr(Expr.Binary expr) {
  return parenthesize(expr.operator.lexeme,
                      expr.left, expr.right);
}

@Override
public String visitGroupingExpr(Expr.Grouping expr){
  return parenthesize("group", expr.expression);
}

@Override
public String visitLiteralExpr(Expr.Literal expr) {
  if (expr.value == null) return "nil";
  return expr.value.toString();
}

@Override
public String visitUnaryExpr(Expr.Unary expr) {
  return parenthesize(expr.operator.lexeme,
          expr.right);
}
```

O método `parenthesize...`

➤ Os métodos `visit` (com exceção de `visitLiteralExpr`) utilizam o método `parenthesize`:

```
private String parenthesize(String name,
        Expr... exprs) {
  StringBuilder builder = new StringBuilder();

  builder.append("(").append(name);
  for (Expr expr : exprs) {
    builder.append(" ");
    builder.append(expr.accept(this));
  }
  builder.append(")");

  return builder.toString();
}
```

É importante notar que `parenthesize` invoca `accept` para cada subexpressão, tal etapa possibilita que recursivamente uma árvore sintática seja exibida.

Exemplo

- Exemplo do funcionamento do *pretty printer*:

```
public static void main(String[] args) {
  Expr expression = new Expr.Binary(
      new Expr.Unary(
          new Token(TokenType.MINUS, "-", null, 1),
          new Expr.Literal(123)),
      new Token(TokenType.STAR, "*", null, 1),
      new Expr.Grouping(
          new Expr.Literal(45.67)));

  System.out.println(
  new AstPrinter().print(expression));
}
```

Saída:

$(* \ (- \ 123) \ (\textit{group} \ 45.6))$

Considerações finais...

 Nesta aula terminamos a implementação do *script* para geração das classes da árvore sintática por meio de metaprogramação.

- Implementamos uma forma de facilitar a introdução de operações (por meio do padrão de projeto *Visitor*).

Na próxima aula: leitura/revisão do código.

Referências

Gamma, Erich et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, p. 416.

Nystrom, Robert (2021). *Crafting Interpreters*. Genever Benning, p. 640.