Abstract Syntax Tree Generation

Principles of Programming Languages

MEng. Tran Ngoc Bao Duy
Department of Computer Science
Faculty of Computer Science and Engineering
Ho Chi Minh University of Technology, VNU-HCM

Overview

1. Understanding Abstract Syntax Trees (AST)
2. Traversing parse tree
3. Visitor pattern
4. Using Visitor in ANTLR4

What is an AST?

Definition

An **Abstract Syntax Tree (AST)** is a hierarchical tree representation of the abstract syntactic structure of source code. Unlike parse trees, ASTs omit unnecessary syntactic details and focus on the essential structure of the program.

Comparison of Parse Tree and AST

Aspect	Parse Tree	AST
Includes syntax	Yes	No
Structure	More verbose	Simplified
Purpose	Parsing and verification	Semantic analysis and code generation

TRAVERSING PARSE TREE GENERATED BY ANTLR4

MEng. Tran Ngoc Bao Duy

Generated parse tree

Suppose that in the BKOOL.g4 file we write as follows:

```
1  grammar BKOOL;
2
3  program: expr EOF; // starting rule
4  expr: expr ADD term | expr SUB term | term;
5  term: term MUL fact | term DIV fact | fact;
6  fact: LP expr RP | ID | INT;
7
8  ADD: '+'; SUB: '-'; MUL: '*'; DIV: '/';
9  ID: [a-zA-Z]+; INT: [0-9]+;
10 WS: [ \t\r\n]+ -> skip;
```


Generated parse tree

Suppose that in the BKOOL.g4 file we write as follows:

```
1  grammar BKOOL;
2
3  program: expr EOF; // starting rule
4  expr: expr ADD term | expr SUB term | term;
5  term: term MUL fact | term DIV fact | fact;
6  fact: LP expr RP | ID | INT;
7
8  ADD: '+'; SUB: '-'; MUL: '*'; DIV: '/';
9  ID: [a-zA-Z]+; INT: [0-9]+;
10 WS: [ \t\r\n]+ -> skip;
```


Generated parse tree

Then, when we run the command `python run.py gen`, we see that 7 files are generated in the `target` directory, with a focus on the following 2 files:

1. `BKOOLLexer.py`: containing the lexer (scanner), in `class BKOOLLexer`
2. `BKOOLParser.py`: containing the parser, in `class BKOOLParser`

Understanding ANTLR4 parse tree

To traverse the generated parse tree, we need to understand the data types of the nodes in the tree according to the following rules:

- `grammar BKOOL` generates `class BKOOLParser`. The parser class is responsible for constructing parse trees according to the grammar rules.
- The nested `*Context` classes inside `BKOOLParser` represent nodes in the parse tree and help manage rule-specific parsing:
  - `ProgramContext` from parser rule `program`.
  - `ExprContext` from parser rule `expr`.
  - `TermContext` from parser rule `term`.
  - `FactContext` from parser rule `fact`.

Common Ways to Access Children

Parse trees are represented by context classes, and you can access child nodes from the parent node using various methods provided by ANTLR's `ParserRuleContext class`. Assume that `ctx` is storing a node in the parse tree:

1. By Indexing: `ctx.getChild(index)`
2. By Type (Specific Rule Context): `ctx.term()`, in which `term` is a rule type in the right hand side of a parser rule. The return value could be in:
   - A single object – when the grammar rule allows only one occurrence of that child.
   - A list of objects – when the rule allows multiple occurrences of that child (includes rules in EBNF).
3. By Field Reference: `ctx.term(0)` or `ctx.term(1)`

Other supported methods in `ParserRuleContext class`:

- Use `getChildCount()` to get the number of children.
- Use `getText()` to get the string representation of a node.

Traverse the parse tree

ANTLR4 provides two ways to traverse the parse tree, categorized into:

1. **Implicit Traversal** (Listener Pattern)
2. **Explicit Traversal** (Visitor Pattern)

Traverse the parse tree

ANTLR4 provides two ways to traverse the parse tree, categorized into:

1. **Implicit Traversal** (Listener Pattern): automatically walks through the parse tree depth-first (post-order traversal). The listener methods are triggered automatically when entering and exiting parse tree nodes.
2. **Explicit Traversal** (Visitor Pattern)

Comparison of Visitor and Listener in ANTLR4

Feature	**Visitor**	**Listener**
Control	Manual control over traversal	Automatic DFS traversal
Method Names	`visitXXX(ctx)`	`enterXXX(ctx)`, `exitXXX(ctx)`
Flexibility	High	Moderate
Ease of Use	Complex	Simple
Use Case	Evaluation, transformation	Analysis, validation
Performance	Slightly slower	Faster

Traverse the parse tree

ANTLR4 provides two ways to traverse the parse tree, categorized into:

1. **Implicit Traversal** (Listener Pattern)
2. **Explicit Traversal** (Visitor Pattern) : provides manual control over how nodes are visited in the parse tree, allowing selective traversal of child nodes.

Comparison of Visitor and Listener in ANTLR4

Feature	**Visitor**	**Listener**
Control	Manual control over traversal	Automatic DFS traversal
Method Names	`visitXXX(ctx)`	`enterXXX(ctx)`, `exitXXX(ctx)`
Flexibility	High	Moderate
Ease of Use	Complex	Simple
Use Case	Evaluation, transformation	Analysis, validation
Performance	Slightly slower	Faster

For generating AST (Transformation), choosing **VISITOR**.

VISITOR PATTERN

Story: J98 – The Musical Wanderer

J98 was a traveler in the world of music, unbound by place or rules. He wandered through life, carrying his melodies, meeting different people, and leaving a lasting impact with each encounter.

1. Along his path, **J98** met **Bé La**, a child to whom he gave **3.5 million VND** without explanation.
2. **J98** crossed paths with **K-IBM**, a powerful figure in the industry. Without hesitation, he confronted **K-IBM**, accusing him of past oppression.
3. **J98** arrived at **Sâu Phát Sáng**, his loyal fan club. He sang **"Giông Bão"**, a song of resilience, as a tribute to those who had stood by him.

If in the world of object-oriented programming, how should we define objects (associated with behaviors)?

Story: J98 – The Musical Wanderer

J98 was a traveler in the world of music, unbound by place or rules. He wandered through life, carrying his melodies, meeting different people, and leaving a lasting impact with each encounter.

1. Along his path, **J98** met **Bé La**, a child to whom he gave **3.5 million VND** without explanation.
2. **J98** crossed paths with **K-IBM**, a powerful figure in the industry. Without hesitation, he confronted **K-IBM**, accusing him of past oppression.
3. **J98** arrived at **Sâu Phát Sáng**, his loyal fan club. He sang **"Giông Bão"**, a song of resilience, as a tribute to those who had stood by him.

If in the world of object-oriented programming, how should we define objects (associated with behaviors)?

Requirement: Implement these classes in Programming Code - Exercise 1 with the support of type() function.

Implementation in OOP (1)

```
1  class Person: pass
2  class BeLa(Person): pass
3  class KIBM(Person): pass
4  class SauPhatSang(Person): pass
5
6  class Visitor: pass
7  class J98(Visitor):
8    def visit(self, ctx: Person):
9      if type(ctx) is BeLa:
10       print("I will give you 3.5 million VND")
11     elif type(ctx) is KIBM:
12       print("You oppressed me in the past")
13     elif type(ctx) is SauPhatSang:
14       print("I will sing a song for you")
```


MEng. Tran Ngoc Bao Duy

Implementation in OOP (1)

```
1  class Person: pass
2  class BeLa(Person): pass
3  class KIBM(Person): pass
4  class SauPhatSang(Person): pass
5
6  class Visitor: pass
7  class J98(Visitor):
8    def visit(self, ctx: Person):
9      if type(ctx) is BeLa:
10       print("I will give you 3.5 million VND")
11     elif type(ctx) is KIBM:
12       print("You oppressed me in the past")
13     elif type(ctx) is SauPhatSang:
14       print("I will sing a song for you")
```

Some issues:

- If there are more than three concrete subclasses of `Person`, what will the body of the `visit` method look like?
- In reality, handling an object belonging to a concrete class is not that simple. Would implementing everything within a single method be considered professional in terms of implementation?

From simple implementation to visitor pattern (2)

Divide the `visit` method into smaller visit methods:

```
class Visitor: pass
class J98(Visitor):
  def visit(self, ctx: Person):
    if type(ctx) is BeLa:
      return self.visitBeLa(ctx)
    elif type(ctx) is KIBM:
      return self.visitKIBM(ctx)
    elif type(ctx) is SauPhatSang:
      return self.visitSauPhatSang(ctx)
  def visitBeLa(self, ctx: BeLa):
    print("I will give you 3.5 million VND")
  def visitKIBM(self, ctx: KIBM):
    print("You oppressed me in the past")
  def visitSauPhatSang(self, ctx: SauPhatSang):
    print("I will sing a song for you")
```


From simple implementation to visitor pattern (2)

Divide the `visit` method into smaller visit methods:

```
class Visitor: pass
class J98(Visitor):
  def visit(self, ctx: Person):
    if type(ctx) is BeLa:
      return self.visitBeLa(ctx)
    elif type(ctx) is KIBM:
      return self.visitKIBM(ctx)
    elif type(ctx) is SauPhatSang:
      return self.visitSauPhatSang(ctx)
  def visitBeLa(self, ctx: BeLa):
    print("I will give you 3.5 million VND")
  def visitKIBM(self, ctx: KIBM):
    print("You oppressed me in the past")
  def visitSauPhatSang(self, ctx: SauPhatSang):
    print("I will sing a song for you")
```

Issue: Even though it has been divided into smaller visit methods, the parent `visit` function is still not professional and generalized; each time a new object type is introduced, it must be explicitly defined.

From simple implementation to visitor pattern (3)

Divide the `visit` method into smaller visit methods:

```
class Visitor: pass
class J98(Visitor):
  def visit(self, ctx: Person):
    class_name = ctx.__class__.__name__
    visit = getattr(self, "visit" + class_name)
    return visit(ctx)
  def visitBeLa(self, ctx: BeLa):
    print("I will give you 3.5 million VND")
  def visitKIBM(self, ctx: KIBM):
    print("You oppressed me in the past")
  def visitSauPhatSang(self, ctx: SauPhatSang):
    print("I will sing a song for you")
```


From simple implementation to visitor pattern (3)

Divide the `visit` method into smaller visit methods:

```
class Visitor: pass
class J98(Visitor):
  def visit(self, ctx: Person):
    class_name = ctx.__class__.__name__
    visit = getattr(self, "visit" + class_name)
    return visit(ctx)
  def visitBeLa(self, ctx: BeLa):
    print("I will give you 3.5 million VND")
  def visitKIBM(self, ctx: KIBM):
    print("You oppressed me in the past")
  def visitSauPhatSang(self, ctx: SauPhatSang):
    print("I will sing a song for you")
```

Issue: The implementation of the parent `visit` method is extremely awkward and unprofessional, failing to demonstrate encapsulation.

From simple implementation to visitor pattern (4)

Implement Exercise 2 in Programming Code:

Follow the steps below to modify the existing J98's story implementation:

1. In each concrete class of `Person`, define an additional `accept` method that takes a `Visitor` object as a parameter, calls, and returns the corresponding `visit` method. Example: In the `BeLa` class, call and return the `visitBeLa` method.
2. In the `visit` method of `J98`, call and return the `accept` method.
3. Keep all previously defined `visit` methods in the `J98` class unchanged.

From simple implementation to visitor pattern (4)

```
1  class Person: pass
2
3  class BeLa(Person):
4    def accept(self, v: Visitor):
5      return v.visitBeLa(self)
6
7  class KIBM(Person):
8    def accept(self, v: Visitor):
9      return v.visitKIBM(self)
10
11 class SauPhatSang(Person):
12   def accept(self, v: Visitor):
13     return v.visitSauPhatSang(self)
```


From simple implementation to visitor pattern (4)

```
1  class Visitor:
2    def visit(self, ctx: Person):
3      return ctx.accept(self)
4
5  class J98(Visitor):
6    def visitBeLa(self, ctx: BeLa):
7      print("I will give you 3.5 million VND")
8
9    def visitKIBM(self, ctx: KIBM):
10     print("You oppressed me in the past")
11
12   def visitSauPhatSang(self, ctx: SauPhatSang):
13     print("I will sing a song for you")
```


Visitor Pattern

Definition

The **Visitor Pattern** allows new operations to be added to existing object structures without modifying their classes. It separates the **data structure** (Elements) from the **operations** (Visitors) performed on them.

Key Components:

1. Elements (Objects being visited):
   - These are existing classes that need new operations.
   - Each element has an `accept(visitor)` method that allows a visitor to interact with it.
2. Visitors (Operations applied to elements):
   - A visitor contains various `visit(element)` methods for handling different element types.
   - New operations can be added without modifying element classes.

USING VISITOR IN ANTLR4

MEng. Tran Ngoc Bao Duy

Visitor class generated by ANTLR4

In 7 files generated in the `target` directory (for example, grammar BKOOL), `class BKOOLVisitor` is generated in file **BKOOLVisitor.py**:

- Each parser rule (a parser rule) will have a corresponding method to process it, named using the form `visit<ParserRuleName>`, where the first letter of the rule name is capitalized.
  Example:
  - `program` will have the method `visitProgram`.
  - `for_stmt` will have the method `visitFor_stmt`.
- To generate the AST, we will inherit from this class and implement each method specifically.

Tips for Writing AST Generation

1. Implement methods for parser rules from the simplest (rules containing the most terminal symbols) to the most complex, typically starting with the lower-level rules among those already implemented.
2. A parser rule has as many cases as the `visit` method must handle.
3. Whenever a non-terminal signal is encountered, call visit, but the returned result must be understood for further processing. If unsure how to handle it, use the + operator to join the list or distribute it into a separate list.
4. Only the child nodes are considered, while the child nodes of those children are ignored.
5. During tree generation, it is necessary to continuously compare with AST classes. If enough data is available to create an object, it should be created and returned immediately.

These rules are almost entirely consistent with grammars written in BNF format.

THANK YOU.