CS3300 Compiler Design: A tutorial on JavaCC/JTB

`Parsers, Syntax-Tree Builders, and Visitors`

Aman Nougrahiya

PACE Lab, Department of CS&E
Indian Institute of Technology Madras

August 27, 2019

Introduction

Goals of this tutorial

Given a grammar in LL(k) format, in this tutorial, we will learn :

A. how to *automatically* create :
   - a **parser** which takes any program as input and generates its AST, and
   - a set of default depth-first **visitors** that traverse over the AST,

   using **JavaCC/JTB**,

Introduction

Goals of this tutorial

Given a grammar in LL(k) format, in this tutorial, we will learn :

A. how to *automatically* create :
   - a **parser** which takes any program as input and generates its AST, and
   - a set of default depth-first **visitors** that traverse over the AST,

   using **JavaCC/JTB**,

B. how to **use** the generated parser, and default traversals (visitors), and

Introduction

Goals of this tutorial

Given a grammar in LL(k) format, in this tutorial, we will learn :

A. how to *automatically* create :
   - a **parser** which takes any program as input and generates its AST, and
   - a set of default depth-first **visitors** that traverse over the AST,

   using **JavaCC/JTB**,
B. how to **use** the generated parser, and default traversals (visitors), and
C. how to write custom visitors to **analyze** (and transform) the programs.

Why?!

This is my post-lunch nap time! Why shall I focus?

[1]

[1]Figures in this slide are taken from http://clipart-library.com

Why?!

This is my post-lunch nap time! Why shall I focus?

[1]

JavaCC/JTB are useful tools for working with *any* structured text (not just programs)!

[1] Figures in this slide are taken from http://clipart-library.com

Why?!

This is my post-lunch nap time! Why shall I focus?

[1]

JavaCC/JTB are useful tools for working with *any* structured text (not just programs)!

Also, ...

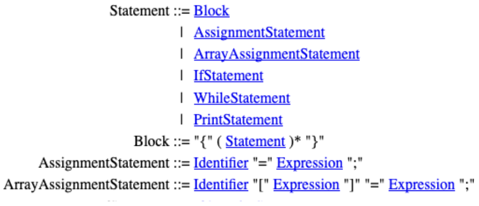
... rest of the ***assignments*** *use JavaCC/JTB :-)*

[1]Figures in this slide are taken from http://clipart-library.com

Outline

JavaCC grammar : An example

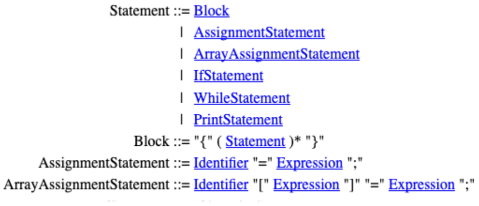
Note a sample grammar, and its JavaCC format below.



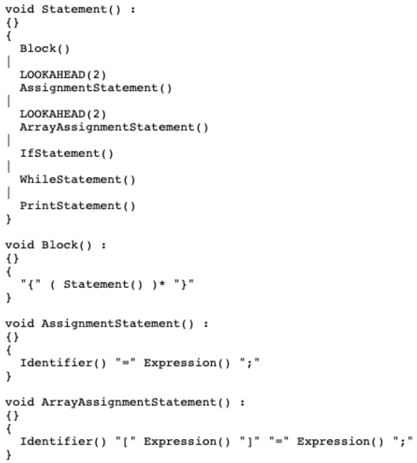
```
               Statement ::= Block
                           | AssignmentStatement
                           | ArrayAssignmentStatement
                           | IfStatement
                           | WhileStatement
                           | PrintStatement
                   Block ::= "{" ( Statement )* "}"
     AssignmentStatement ::= Identifier "=" Expression ";"
ArrayAssignmentStatement ::= Identifier "[" Expression "]" "=" Expression ";"
```

JavaCC grammar : An example

Note a sample grammar, and its JavaCC format below.



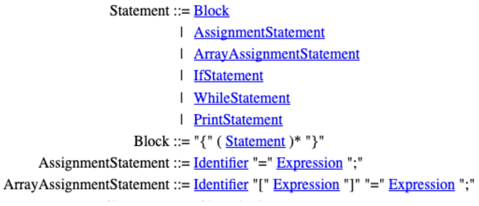
```
Statement ::= Block
            | AssignmentStatement
            | ArrayAssignmentStatement
            | IfStatement
            | WhileStatement
            | PrintStatement
    Block ::= "{" ( Statement )* "}"
AssignmentStatement ::= Identifier "=" Expression ";"
ArrayAssignmentStatement ::= Identifier "[" Expression "]" "=" Expression ";"
```



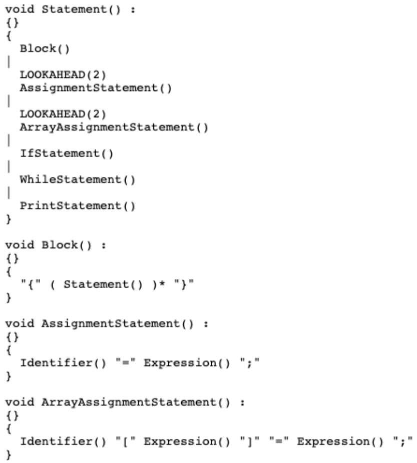
```
void Statement() :
{}
{
  Block()
|
  LOOKAHEAD(2)
  AssignmentStatement()
|
  LOOKAHEAD(2)
  ArrayAssignmentStatement()
|
  IfStatement()
|
  WhileStatement()
|
  PrintStatement()
}

void Block() :
{}
{
  "{" ( Statement() )* "}"
}

void AssignmentStatement() :
{}
{
  Identifier() "=" Expression() ";"
}

void ArrayAssignmentStatement() :
{}
{
  Identifier() "[" Expression() "]" "=" Expression() ";"
}
```

JavaCC grammar : An example

Note a sample grammar, and its JavaCC format below.



```
Statement ::= Block
            | AssignmentStatement
            | ArrayAssignmentStatement
            | IfStatement
            | WhileStatement
            | PrintStatement
Block ::= "{" ( Statement )* "}"
AssignmentStatement ::= Identifier "=" Expression ";"
ArrayAssignmentStatement ::= Identifier "[" Expression "]" "=" Expression ";"
```

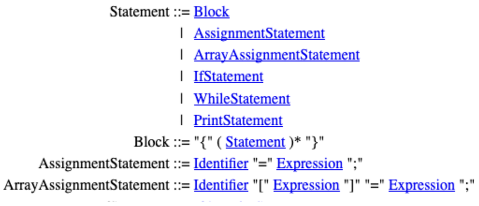


```
void Statement() :
{}
{
  Block()
|
  LOOKAHEAD(2)
  AssignmentStatement()
|
  LOOKAHEAD(2)
  ArrayAssignmentStatement()
|
  IfStatement()
|
  WhileStatement()
|
  PrintStatement()
}

void Block() :
{}
{
  "{" ( Statement() )* "}"
}

void AssignmentStatement() :
{}
{
  Identifier() "=" Expression() ";"
}

void ArrayAssignmentStatement() :
{}
{
  Identifier() "[" Expression() "]" "=" Expression() ";"
}
```

Note: You will get JavaCC grammars as input in your assignments.

Parsing

Note an example snippet, and its AST, as per the grammar.

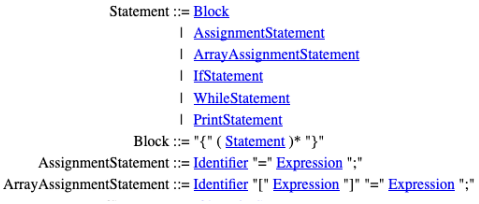


```
               Statement ::= Block
                           | AssignmentStatement
                           | ArrayAssignmentStatement
                           | IfStatement
                           | WhileStatement
                           | PrintStatement
                   Block ::= "{" ( Statement )* "}"
     AssignmentStatement ::= Identifier "=" Expression ";"
ArrayAssignmentStatement ::= Identifier "[" Expression "]" "=" Expression ";"
```

Figure: Example grammar.

Parsing

Note an example snippet, and its AST, as per the grammar.



```
              Statement ::= Block
                          | AssignmentStatement
                          | ArrayAssignmentStatement
                          | IfStatement
                          | WhileStatement
                          | PrintStatement
                  Block ::= "{" ( Statement )* "}"
    AssignmentStatement ::= Identifier "=" Expression ";"
ArrayAssignmentStatement ::= Identifier "[" Expression "]" "=" Expression ";"
```

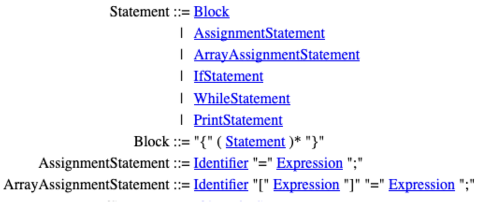
Figure: Example grammar.

```
{
    i = 5;
    arr[0] = 10;
}
```

Figure: Example program snippet.

Parsing

Note an example snippet, and its AST, as per the grammar.

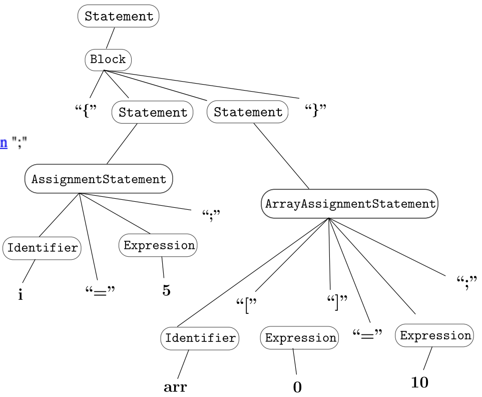


```
Statement ::= Block
            | AssignmentStatement
            | ArrayAssignmentStatement
            | IfStatement
            | WhileStatement
            | PrintStatement
Block ::= "{" ( Statement )* "}"
AssignmentStatement ::= Identifier "=" Expression ";"
ArrayAssignmentStatement ::= Identifier "[" Expression "]" "=" Expression ";"
```

Figure: Example grammar.

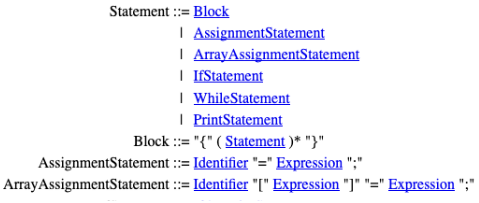
```
{
    i = 5;
    arr[0] = 10;
}
```

Figure: Example program snippet.



Parsing

Note an example snippet, and its AST, as per the grammar.

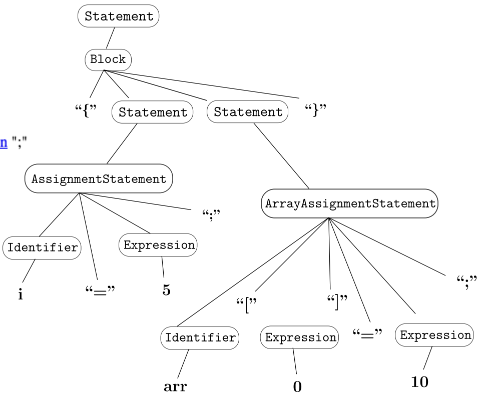


```
            Statement ::= Block
                        | AssignmentStatement
                        | ArrayAssignmentStatement
                        | IfStatement
                        | WhileStatement
                        | PrintStatement
                Block ::= "{" ( Statement )* "}"
  AssignmentStatement ::= Identifier "=" Expression ";"
ArrayAssignmentStatement ::= Identifier "[" Expression "]" "=" Expression ";"
```

Figure: Example grammar.

```
{
     i = 5;
     arr[0] = 10;
}
```

Figure: Example program snippet.



Note: Your parser will automatically generate the AST for any given input.

What?

What is JavaCC (Java Compiler Compiler)?

Input: Grammar specification in JavaCC format.

- A single grammar file.
- Also contains the lexical specifications.

Output: Parser (a Java program).

What?

What is JavaCC (Java Compiler Compiler)?

Input: Grammar specification in JavaCC format.

- A single grammar file.
- Also contains the lexical specifications.

Output: Parser (a Java program).

What is JTB?

Input: Plain JavaCC grammar.
Output:

- Syntax-tree classes (for non-terminals and terminals).
- Annotated JavaCC grammar, which builds the syntax-tree during parsing.
- Default visitors over the AST.

Outline

Setting up JavaCC/JTB

1. Install JavaCC/JTB *(instructions in backup slides)*.
2. Write/obtain the desired grammar in `.jj` format.

Setting up JavaCC/JTB

1. Install JavaCC/JTB *(instructions in backup slides)*.
2. Write/obtain the desired grammar in `.jj` format.
3. Use JTB to create syntax-tree classes, visitors, and annotated JavaCC grammar.

   ```
   $ java -jar jtb132.jar my-grammar.jj
   ```

   This generates `jtb.out.jj`, and other classes (in `syntaxtree` and `visitor` packages).

Setting up JavaCC/JTB

1. Install JavaCC/JTB *(instructions in backup slides)*.
2. Write/obtain the desired grammar in `.jj` format.
3. Use JTB to create syntax-tree classes, visitors, and annotated JavaCC grammar.

   ```
   $ java -jar jtb132.jar my-grammar.jj
   ```

   This generates `jtb.out.jj`, and other classes (in `syntaxtree` and `visitor` packages).
4. Use JavaCC to obtain the parser.

   ```
   $ javacc jtb.out.jj
   ```

   This generates the parser classes.

Setting up JavaCC/JTB

1. Install JavaCC/JTB *(instructions in backup slides)*.
2. Write/obtain the desired grammar in `.jj` format.
3. Use JTB to create syntax-tree classes, visitors, and annotated JavaCC grammar.

   ```
   $ java -jar jtb132.jar my-grammar.jj
   ```

   This generates `jtb.out.jj`, and other classes (in `syntaxtree` and `visitor` packages).
4. Use JavaCC to obtain the parser.

   ```
   $ javacc jtb.out.jj
   ```

   This generates the parser classes.
5. Invoke the generated Java parser in your program.

   e.g., `Node root = new MiniJavaParser(System.in).Goal();`

Setting up JavaCC/JTB

1. Install JavaCC/JTB *(instructions in backup slides)*.
2. Write/obtain the desired grammar in `.jj` format.
3. Use JTB to create syntax-tree classes, visitors, and annotated JavaCC grammar.

   ```
   $ java -jar jtb132.jar my-grammar.jj
   ```

   This generates `jtb.out.jj`, and other classes (in `syntaxtree` and `visitor` packages).
4. Use JavaCC to obtain the parser.

   ```
   $ javacc jtb.out.jj
   ```

   This generates the parser classes.
5. Invoke the generated Java parser in your program.

   e.g., `Node root = new MiniJavaParser(System.in).Goal();`

Suggestion : After these steps, open your project in Eclipse.

Demo

Demonstration of setting up a MiniJava parser.

Outline

Generated syntax-tree classes

The syntaxtree package

- All syntax-tree classes implement the Node interface.
- For each non-terminal, JTB creates one Java class with same name in syntaxtree.
- For each terminal, JTB uses a special class NodeToken, present in syntaxtree.

Generated syntax-tree classes

The `syntaxtree` package

- All syntax-tree classes implement the `Node` interface.
- For each non-terminal, JTB creates one Java class with same name in `syntaxtree`.
- For each terminal, JTB uses a special class `NodeToken`, present in `syntaxtree`.
- Each non-terminal class is connected via its *fields* to the RHS elements in its production rules.

Generated syntax-tree classes

The syntaxtree package

- All syntax-tree classes implement the `Node` interface.
- For each non-terminal, JTB creates one Java class with same name in `syntaxtree`.
- For each terminal, JTB uses a special class `NodeToken`, present in `syntaxtree`.
- Each non-terminal class is connected via its *fields* to the RHS elements in its production rules.

Note : In general, you do not need to modify any of the syntax-tree files.

Demo

Walkthrough of syntax-tree classes.

Outline

The visitor classes

Visitor Design Pattern

- In visitor design pattern, users can define *new operations* on objects, without having to update the *classes* of the objects [a].

The visitor classes

Visitor Design Pattern

- In visitor design pattern, users can define *new operations* on objects, without having to update the *classes* of the objects [a].
- On objects (AST nodes) we invoke **accept()** methods :
  - They take a visitor (representing a traversal) as an argument.
  - They internally invoke visit() method on the visitor, passing itself as an argument.

The visitor classes

Visitor Design Pattern

- In visitor design pattern, users can define *new operations* on objects, without having to update the *classes* of the objects [a].
- On objects (AST nodes) we invoke **accept()** methods :
  - They take a visitor (representing a traversal) as an argument.
  - They internally invoke visit() method on the visitor, passing itself as an argument.
- A **visit()** method may generally:
  - **process** the argument node being traversed, and
  - invoke the `accept()` method on any connected node(s), passing itself as the argument, to **traverse** the structure.

For each type of node, there may be an *overloaded* `visit()` method in each visitor.

The visitor classes

Visitor Design Pattern

- In visitor design pattern, users can define *new operations* on objects, without having to update the *classes* of the objects [a].
- On objects (AST nodes) we invoke **accept()** methods :
  - They take a visitor (representing a traversal) as an argument.
  - They internally invoke visit() method on the visitor, passing itself as an argument.
- A **visit()** method may generally:
  - **process** the argument node being traversed, and
  - invoke the `accept()` method on any connected node(s), passing itself as the argument, to **traverse** the structure.

  For each type of node, there may be an *overloaded* `visit()` method in each visitor.
- Where shall I write my code?

  *Users need to **write code in visit() methods**, not accept() methods.*

[a]Gamma, Helm, Johnson, Vlissides : **Design Patterns**, 1995.

The visitor classes

Visitor Design Pattern

- In visitor design pattern, users can define *new operations* on objects, without having to update the *classes* of the objects [a].
- On objects (AST nodes) we invoke **accept()** methods :
  - They take a visitor (representing a traversal) as an argument.
  - They internally invoke visit() method on the visitor, passing itself as an argument.
- A **visit()** method may generally:
  - **process** the argument node being traversed, and
  - invoke the `accept()` method on any connected node(s), passing itself as the argument, to **traverse** the structure.

  For each type of node, there may be an *overloaded* `visit()` method in each visitor.
- Where shall I write my code?

  *Users need to **write code in visit() methods**, not accept() methods.*

[a] Gamma, Helm, Johnson, Vlissides : **Design Patterns**, 1995.

Note : In each assignment, your key task would revolve around writing one or more visitors, and using them in your main program.

Default visitors by JTB

The `visitor` package

- Contains various default depth-first visitor classes for the AST.
- Categorized according to whether the visits
  - take any arguments (sent to the child node), and/or
  - return any values (back to the parent node).

Demo

Walkthrough of visitor classes.

Outline

Recap

- In your assignments, you will be given a `.jj` file.
- Setup your project as explained before.

[2] https://looneytunes.fandom.com/wiki/

Recap

- In your assignments, you will be given a `.jj` file.
- Setup your project as explained before.
- Design your solution as one or more visits (traversals) over the AST.

[2] https://looneytunes.fandom.com/wiki/

Recap

- In your assignments, you will be given a `.jj` file.
- Setup your project as explained before.
- Design your solution as one or more visits (traversals) over the AST.
- Select any default visitor(s); extend it to create your custom visitor(s).

[2] https://looneytunes.fandom.com/wiki/

Recap

- In your assignments, you will be given a `.jj` file.
- Setup your project as explained before.
- Design your solution as one or more visits (traversals) over the AST.
- Select any default visitor(s); extend it to create your custom visitor(s).
- In the custom visitor, write code for each *type* of node by overriding the corresponding overloaded `visit()` method.

[2] https://looneytunes.fandom.com/wiki/

Recap

- In your assignments, you will be given a `.jj` file.
- Setup your project as explained before.
- Design your solution as one or more visits (traversals) over the AST.
- Select any default visitor(s); extend it to create your custom visitor(s).
- In the custom visitor, write code for each *type* of node by overriding the corresponding overloaded `visit()` method.
- Invoke the parser and visitors from your `main()` method.

[2] https://looneytunes.fandom.com/wiki/

Recap

- In your assignments, you will be given a `.jj` file.
- Setup your project as explained before.
- Design your solution as one or more visits (traversals) over the AST.
- Select any default visitor(s); extend it to create your custom visitor(s).
- In the custom visitor, write code for each *type* of node by overriding the corresponding overloaded `visit()` method.
- Invoke the parser and visitors from your `main()` method.

[2]

[2] https://looneytunes.fandom.com/wiki/

Practice!

Example visitors

Let's start writing some visitors then!

- Write a visitor to print the name of all the classes.
- Count the number of explicit operators in the program.
- Print the fully-qualified name of all integer fields.
- TODO : Write a visitor to calculate the cost of each expression being printed, in terms of number of explicit operators present in the print expression. e.g., `System.out.println((2 + x) * y)` has cost of 2 (one for +, and one for ∗).
- TODO : Modify the previous visitor to calculate the cost of expressions, as per the following :
  - Cost of reading a constant : 0; that of reading a variable : 1
  - Cost of each arithmetic operator : 1; of array dereference : 2; other operators : 0
  - Cost of a method call : 4
- TODO : Write a pretty-printer (one which prints the program, taking care of newlines and indentations.)

Thank you!

Outline

Installation notes

Installing JavaCC

- Download and unzip `javacc-5.0.tar.gz`, say, at your home directory.

  https://java.net/projects/javacc/downloads/download/javacc-5.0.tar.gz

  ```
  $ cd ~
  $ tar xvzf javacc-5.0.tar.gz
  ```

- Set the path to `javacc` (present in `javacc-5.0/bin`) in your `PATH` environment variable.

  ```
  $ export PATH="~/javacc-5.0/bin:$PATH"
  ```

  Save this command in your `~\.bashrc` or `~\.bash_profile`.

Installing JTB

- *None required.* Simply download and use the `jar` file from

  http://compilers.cs.ucla.edu/jtb/Files/jtb132.jar

Special sub-classes of Node

There are special subclasses of Node, which are used to represent terminals, and meta-operators (+, ∗, ?, etc.) in the grammar.

- NodeToken is used to represent terminals in the AST.
  Terminal's string can be obtained using the field tokenImage.

Special sub-classes of `Node`

There are special subclasses of `Node`, which are used to represent terminals, and meta-operators (+, $*$, ?, etc.) in the grammar.

- `NodeToken` is used to represent terminals in the AST.
  Terminal's string can be obtained using the field `tokenImage`.
- `NodeList` represents + meta-operator, denoting one or more occurrences of its operand.
  e.g., In `A := B+`, the first field of A is of type `NodeList`, which contains a list of nodes of type `B` (in field `nodes:Vector<Node>`).

Special sub-classes of `Node`

There are special subclasses of `Node`, which are used to represent terminals, and meta-operators (+, ∗, ?, etc.) in the grammar.

- `NodeToken` is used to represent terminals in the AST.
  Terminal's string can be obtained using the field `tokenImage`.
- `NodeList` represents + meta-operator, denoting one or more occurrences of its operand.
  e.g., In `A := B+`, the first field of A is of type `NodeList`, which contains a list of nodes of type `B` (in field `nodes:Vector<Node>`).
- `NodeChoice` is used to denote a grammar choice, such as `A := B | C`.
  Field `choice` refers to the chosen node (of type `B` or `C`).

Special sub-classes of `Node`

There are special subclasses of `Node`, which are used to represent terminals, and meta-operators (+, ∗, ?, etc.) in the grammar.

- `NodeToken` is used to represent terminals in the AST.
  Terminal's string can be obtained using the field `tokenImage`.
- `NodeList` represents + meta-operator, denoting one or more occurrences of its operand.
  e.g., In `A := B+`, the first field of A is of type `NodeList`, which contains a list of nodes of type `B` (in field `nodes:Vector<Node>`).
- `NodeChoice` is used to denote a grammar choice, such as `A := B | C`.
  Field `choice` refers to the chosen node (of type `B` or `C`).
- *TODO :* Check what are `NodeSequence`, `NodeListOptional` and `NodeOptional` classes.

Advanced programming tips (1/2)

- Revise/learn Java Generics and Collections API.
- Decide whether you need to send information from parent to child, from child to parent, in both directions, or in neither. Accordingly pick an existing visitor.
- To create a custom visitor, start it as a copy of the selected visitor, make it extend the selected visitor, and then edit the copy.
- When passing information from child to parent, it might be easier to code the `visit()` methods in a bottom-up order.
  (e.g., expressions → statements → methods → classes).
- When passing information from parent to child, fill the `visit()` methods in top-down order.
  (e.g., classes → methods → statements → expressions).
- While writing `visit()` methods that call each other recursively, take each `visit()` one-by-one, and *assume* that the other `visit()` methods are already implemented while writing its code.

Advanced programming tips (2/2)

- Some situations may require more than one visitors.
- To keep your code clean, remove all those methods from your custom visitor that do not modify the inherited definition from selected visitor.
- Do not call `accept()` on those portions of AST which need not be processed.
- Java `Strings` can be passed as an argument to the parser, by wrapping it in a `ByteArrayInputStream`.
- The parser can be invoked on *any* non-terminal, not just the start symbol. Hence, AST for code *snippets* (e.g., a while loop) can be created with ease.