Spoon2AST

Description

A tool able to parse Java code to output the corresponding abstract-syntax tree (AST) in a JSON format. The tool can be downloaded here.

Usage

The tool requires as input a file/folder of the Java code, followed by optional files/folders that define the classpath. E.g.:

If classpath is not required:	`java -jar spoon2ast.jar MyClass.java`
If code requires a specific library:	`java -jar spoon2ast.jar MyClass.java MyLib.jar`
`Using a folder as input:`	`java -jar spoon2ast.jar src`

Nodes Description

This chapter describes some of the nodes and their relation to the child nodes. A node is represented as a JSON object that contains at least the following properties:

- **`nodetype:`** defines the type of the current node. This node is mandatory.
- **`location`**: location of the node in the source code.
- **`annotations:`** optional property that contains an array of annotations.
- **`comments:`** optional property containing an array of `Comment`s

There are other properties that may be defined in the node which are related to that type of node. The following sections define the several types of nodes and their relation.

Root

This is the top-most node. This node contains a property named **`compilation_units`** that contains an array of `CompilationUnit`s

CompilationUnit

A compilation unit a node of a source code file. The node contains the following properties:

- **`location`**: absolute path to the parsed file
- **`types`**: an array containing the `Type`s declared in the file

Class

A class has 6 properties:

- **`name:`** the name of the class
- **`package:`** the package of the class, may be empty if no package is defined in the class
- **`super:`** an optional TypeReference to the type that the class extends.
- **`interfaces:`** an optional array of TypeReferences of the types that the class implements
- **`formal_type_parameters:`** an optional array of the TypeParameter for the generic types
- **`members:`** an array of Members.

Interface

An interface has 5 properties:

- **`name:`** the name of the class
- **`package:`** the package of the class, may be empty if no package is defined in the class
- **`interfaces:`** an optional array of TypeReferences of the types that the class implements
- **`formal_type_parameters:`** an optional array of the TypeParameter for the generic types
- **`members:`** an array of Members.

Enum

A class has 7 properties:

- **`name:`** the name of the class
- **`package:`** the package of the class, may be empty if no package is defined in the class
- **`super:`** an optional TypeReference to the type that the class extends.
- **`interfaces:`** an optional array of TypeReferences of the types that the class implements
- **`formal_type_parameters:`** an optional array of the TypeParameter for the generic types
- **`members:`** an array of Members
- **`values:`** an array of EnumValues

TypeParameter

Defines a declaration of a type parameter (aka generics). For example, in class A<E> { ... }, the "E" is modeled as an instance of TypeParameter. This node has 2 properties:

- **`name:`** the name of the class
- **`super:`** an optional TypeReference to the type that the class extends.

NewArray

Contains 3 out of 4 properties:

- **`type:`** the TypeReference of the array base type
- **`type_casts`**: an optional array of TypeReference of the casts
- **`elements`**: an array of expressions that define the actual, initial, values of the array. If this property is defined than **`dimensions`** property is not present (e.g. int i = **{2,3,4}**)
- **`dimensions`**: an array of integer expressions that define the dimensions of the array. If this property is defined than **`elements`** is not present (e.g. int i = new int[**3**])

For

Contains 4 properties:

- **`init`**: an array of statements of the for initializations.
- **`condition`**: an expression node of the for condition.
- **`update`**: an array of statements of the for updates
- **`body:`** a statement if contains one executable statement, a CompoundStatement if the for body contains multiple statements**,** or a NullNode if it does not have a body, i.e., no statements.

CompoundStatement

Comment

A comment contains 3 properties:

- **`type:`** a string defining the type of the comment. Possible values: **`inline`**, **`block`**, **`javadoc`** or **`file`**
- **`content:`** a string with all the content of the comment
- **`position:`** a string that defines if the comment is **`before`** or **`after`** `the node owner of this comment`

Abstract Nodes Description

Expression

An expression can be: `ArrayRead, ArrayWrite, Assignment, BinaryOperator, Conditional, ConstructorCall, ExecutableReferenceExpression, FieldRead, FieldWrite, Invocation, Lambda, Literal, NewArray, OperatorAssignment, SuperAccess, ThisAccess, TypeAccess, UnaryOperator, VariableRead` or `VariableWrite`

Statement

A statement can be: `Assert`, `Assignment`, `OperatorAssignment`, `Block`, `Break`, `Case`, `Class`, `Comment`, `Continue`, `ConstructorCall`, `Do`, `Enum`, `For`, `ForEach`, `If`, `Invocation`, `LocalVariable`, `Loop`, `NewClass`, `Return`, `Switch`, `Synchonized`, `Throw`, `Try`, `TryWithResources` or `While`

Type

A type node can be a `Class`, `Interface` or `TypeParameter` (aka Generics). An `Enum` extends a class.

Reference

A reference node can be: `ArrayTypeReference`, `ExecutableReference`, `FieldReference`, `ParameterReference`, `TypeParameterReference` or `TypeReference`.

Member

`A member can be: AnonymousExecutable, Constructor, Field, Method, Type`

NullNode

A node that represents a child node that does not contain a value. This node is used, for instance, to represent an empty body of a `For` node.

List of Properties

This chapter defines a complete list of existing properties and a list of nodes that (may) use them.

- **`annotations:`** `all`
- **`anonymous_class:`** `NewClass`
- **`arguments:`** `ConstructorCall, Invocation, NewClass`
- **`body:`** `AnonymousExecutable, Catch, Constructor, Do, For, ForEach, Lambda, Method, Synchronized, Try, TryWithResources, While`
- **`bounds:`** `IntersectionTypeReference`
- **`cases:`** `Switch`
- **`catchers:`** `Try, TryWithResources`
- **`comments:`** `all`

- **compilation_units:** Root
- **condition:** Conditional, Do, For, Foreach, If, Switch, While
- **content:** Comment
- **declarator:** Reference
- **dimensions:** NewArray
- **elements:** NewArray
- **else:** Conditional, If
- **executable:** ConstructorCall, ExecutableReferenceExpression, Invocation, NewClass
- **expression:** Case, Lambda, Return, Synchronized Throw
- **expressions:** Assert
- **finalizer:** Try, TryWithResources
- **formal_type_parameters:** Type, Constructor, Method
- **index:** ArrayRead, ArrayWrite
- **init:** Field, For, LocalVariable
- **interfaces:** Type
- **label:** Statement
- **lhs:** Assignment, BinaryOperator, OperatorAssignment
- **location:** all
- **members:** Type
- **name:** Type, CatchVariable, CatchVariableReference, Constructor, Field, LocalVariable, Method, Parameter, TypeParameter, Reference, WildcardReference
- **nodetype:** all
- **operand:** UnaryOperator
- **operator:** BinaryOperator, OperatorAssignment, UnaryOperator
- **package:** Reference
- **parameter:** CtCatch
- **parameters:** Constructor, ExecutableReference, Lambda, Method
- **position:** Comment
- **resources:** TryWithResources
- **rhs:** Assignment, BinaryOperator, OperatorAssignment
- **statements:** Block, Case
- **super:** Class, Enum, TypeParameter
- **target:** ArrayRead, ArrayWrite, Break, ConstructorCall, Continue, ExecutableReferenceExpression, FieldRead, FieldWrite, Invocation, NewClass, SuperAccess, ThisAccess, TypeAccess
- **then:** Conditional, If
- **throws:** Constructor, Method
- **type:** Annotation, ArrayRead, ArrayTypeReference, ArrayWrite, Assignment, BinaryOperator, CatchVariable, Comment, EnumValue, ExecutableReferenceExpression, Field, Lambda, Literal, LocalVariable,

Method, NewArray, OperatorAssignment, Parameter, Reference, SuperAccess, ThisAccess, UnaryOperator, WildcardReference

- **type_arguments:** ArrayTypeReference, ExecutableReference, TypeReference
- **type_casts:** ArrayRead, ArrayWrite, Assignment, BinaryOperator, Conditional, ConstructorCall, ExecutableReferenceExpression, FieldRead, FieldWrite, Invocation, Lambda, Literal, NewArray, NewClass, OperatorAssignment, SuperAccess, ThisAccess, TypeAccess, UnaryOperator, VariableRead, VariableWrite
- **types:** CompilationUnit
- **update:** For
- **value:** EnumValue, Literal
- **values:** Annotation, Enum
- **var:** FieldRead, FieldWrite, ForEach, VariableRead, VariableWrite

Tools and APIs for JSON parsing

Here we list some tools that can be used to load the JSON file that is generated by Spoon2AST.

- org.json: http://mvnrepository.com/artifact/org.json/json/20160810
- Google GSON: https://mvnrepository.com/artifact/com.google.code.gson/gson/2.8.0

Observations

Using a library such as GSON the JSON file can be loaded and structured as a map. You can use this map as your tree and navigate this tree by using the keys “name”, “content” and “children”. You can also create your own tree with your own classes that represent each node, where each class points to a different “name” of a node and have a field List<Node> to include the children.
For instance, for the node named “VariableRead”, you can create a class such as:

```
class VariableRead extends Node {
    public Node getVariable()...
    public Node getType()...
}
```

Example of using Gson:

```
File json = new File("file.json");
String jsonString = readFile(json); //read the file as a string
Map<?, ?> jsonJavaRootObject = new Gson().fromJson(jsonString, Map.class);
System.out.println("JSON:\n" + jsonJavaRootObject);
```

Using GSON

Toda a conversão de JSON para classes java é bastante simples com o GSON depois de se habituarem a trabalhar com ele.

First Remarks:

1) vamos utilizar como exemplo a expressão "2 + a" cujo JSON correspondente será o seguinte:


```
{
  "nodetype": "BinaryOperator",
  "location": "/Users/tiago/Desktop/spoon2ast/src/Test.java:8",
  "type": {
       "nodetype": "TypeReference",
       "name": "int"
       },
  "operator": "+",
  "lhs": {
       "nodetype": "Literal",
       "location": "/Users/tiago/Desktop/spoon2ast/src/Test.java:8",
       "value": 2,
       "type": {
         "nodetype": "TypeReference",
         "name": "int"
         }
       },
  "rhs": {
       "nodetype": "VariableRead",
       "location": "/Users/tiago/Desktop/spoon2ast/src/Test.java:8",
       "var": {
         "nodetype": "LocalVariableReference",
         "name": "a",
         "type": {
              "nodetype": "TypeReference",
              "name": "int"
              }
         }
       }
}
```


2) Para este exemplo vamos precisar de pelo menos 5 classes concretas: BinaryOperator, Literal, VariableRead, TypeReference e LocalVariableReference

3) Cada uma dessas classes DEVE estender uma classe mãe, abstrata, que vou dar como nome BasicNode. Este BasicNode irá ter os fields comuns a todos os nos, ou seja (aqui só represento um pouco das classes):

```
public abstract class BasicNode{
	private String notetype;
	private String location;
    private List<Comment> comments; //Comment é uma classe para representar os comentários, que estenderá também BasicNode (Comment extends BasicNode)
}
```

4) Cada uma das classes a implementar irá ter fields associados aos seus atributos, p.e.:

```
//Adicionei uma classe abstrata chamada Expression
//Esta classe estende BasicNode e assim terá as propriedades comuns a todos os nós
public Expression extends BasicNode{
    protected TypeReference type;  //Como todas as expressões vão ter um tipo associado, deixei aqui o type, que pode ser acedido p.e. pela classe Literal
}

public BinaryOperator extends Expression { //aqui não estendi de BasicNode para esta classe poder ser usada como expression
	private String operator;
    private Expression lhs; //aqui defino lhs e rhs como Expression pois elas deverão poder conter qualquer expressão, e não obrigatóriamente uma VariableRead ou Literal como acima
	private Expression rhs;
}

public Literal extends Expression{ //Aqui não estendi de BasicNode mas sim de expression, para poder ser usado como uma expressão!
	private String value;
}

public TypeReference extends Reference {
    @SerializedName("package") //usem esta anotação quando a propriedade no JSON é uma palavra reservada. aqui para não dar erro no Java digo que a minha variável
	private String _package;  //chama-se "_package" MAS a propriedade que ele deve usar é mesmo "package"
}

public Reference extends BasicNode {
	private String name;
}

... //and so on...
```

Using GSON

Assumindo que as classes atrás mencionadas foram criadas:

1) Em primeiro lugar aquilo que vocês precisam para converter um determinado objeto JSON (que neste caso terá SEMPRE a propriedade "notetype") num objeto de uma classe definida por vocês é o que o GSON chama de "TypeAdapter" para a desserialização. Ou seja, vocês terão que criar uma classe que implementa a interface "com.google.gson.JsonDeserializer" como por exemplo:

```
public class MyNodeDeserializer implements JsonDeserializer<BasicNode> {
 @Override
 public BasicNode deserialize(JsonElement jsonElement, Type type,
JsonDeserializationContext jsonDeserializationContext) throws JsonParseException {
              //A conversão é feita aqui!
       }
}
```

2) agora terão de escrever o código que pega no objeto atual JSON e converte para a classe que vocês pretendem! Para isso usam então a propriedade que vem dentro de cada objeto JSON chamada "nodetype" e conforme esse nome escolhem a classe que pretendem. A classe que escolhem DEVE SER SEMPRE UMA CLASSE CONCRETA! No exemplo dado as classes concretas são: BinaryExpression, Literal, LocalVariableReference, TypeReference e VariableRead. As classes abstratas não entram aqui!

O corpo do método acima (deserialize) poderia ficar do género:

```
public BasicNode deserialize(JsonElement jsonElement, Type type,
JsonDeserializationContext jsonDeserializationContext) throws JsonParseException {
 try {
       JsonObject jsonObj = jsonElement.getAsJsonObject();  //vão buscar o objeto
atual como um JsonObject para poderem ir buscar a propriedade que querem
       JsonElement nodeTypeEl = jsonObj.get("nodetype");    // get the type of the
node so we can use the correct class
       if (nodeTypeEl == null) {
              throw new RuntimeException("nodetype property must be defined!"); // all
JSON objects must have the property nodetype
       }

       String nodeType = nodeTypeEl.getAsString(); //simply casting the object as
string
    Class<? extends BasicNode> classToUse = getClassToUse(nodeType); //somehow get
the Class to use based on the node type given
```

```
	return jsonDeserializationContext.deserialize(jsonElement, classToUse); //
automatic desearialization.
 } catch (Exception e) {
	throw new JsonParseException(e);
 }
}
```

3) o método getClassToUse(String nodeType) será onde vocês escolhem a classe que querem para cada nó. Podem ter um switch, um mapa que aponta um type a uma classe, podem usar reflection, podem usar um enum,... Ou seja aqui a abordagem é como quiserem, eu usei um enum em que cada valor do enum contem uma classe:

```
private Class<? extends BasicNode> getClassToUse(String nodeType) {

	return MyClass.valueOf(nodeType.toUpperCase()).myClass; // we use an enum to
map a node type to a specific class
}

private enum MyClass {
	// You Only need the concrete classes, NOT the abstract ones
	TYPEREFERENCE(TypeReference.class),
	LOCALVARIABLEREFERENCE(LocalVariableReference.class),
	BINARYEXPRESSION(BinaryExpression.class),
	LITERAL(Literal.class),
	VARIABLEREAD(VariableRead.class),
	// Define the remaining nodetype and corresponding classes
	;
	private Class<?> myClass;

	private MyClass(Class<? extends BasicNode> myClass) {
		this.myClass = myClass;
	}
}
```

4) Para dizer ao GSON para usar esta classe na desserialização devem registar o adaptador PARA TODAS AS CLASSES ABSTRATAS! (Pelo menos os testes que fiz se não puser as classes abstratas dá erro por o GSON tentar instanciar a classe abstrata) No exemplo dado as classes abstratas são: BasicNode, Reference e Expression. As classes concretas não entram aqui. A criação do gson deverá então ser da seguinte forma:

```
 NodeDeserializer typeAdapter = new NodeDeserializer();
 Gson gson = new GsonBuilder()
		.registerTypeAdapter(BasicNode.class, typeAdapter)
		.registerTypeAdapter(Reference.class, typeAdapter)
		.registerTypeAdapter(Expression.class, typeAdapter)
		.create();
```

5) O resto será responsabilidade do GSON, só terão agora que dar o input ao JSON e a classe do objeto mais exterior. Se fosse um JSON normal dado pelo spoon2ast o mais exterior seria o "Root", mas como estamos apenas a fazer um teste ao BinaryExpression este será o objeto mais exterior:

```
BinaryExpression fromJson = gson.fromJson(exampleString, BinaryExpression.class);
System.out.println(fromJson); //Deverá dar o toString que definirem para o
BinaryExpression!
```

6) Quanto à tua dúvida em relação aos arrays, não te precisas de preocupar com isso pois o GSON trata disso! Basta declarares as propriedades que são definidas como array como uma lista de uma determinada classe. Por exemplo no caso de um nó que tem a propriedade "arguments" (como o exemplo do Invocation), é esperado que seja um array de expressões, portanto na classe desse nó declaram um field do tipo: List<Expression> arguments. No caso do Invocation:

```
public class Invocation extends Expression{
       ... //outras propriedades
    public List<Expression> arguments; //O GSON irá tratar de criar primeiro cada um
dos argumentos e adiciona a esta lista!
}
```

Useful links

http://www.javacreed.com/simple-gson-example/
http://stackoverflow.com/questions/3629596/deserializing-an-abstract-class-in-gson
http://stackoverflow.com/questions/3763937/gson-and-deserializing-an-array-of-objects-with-arrays-in-it
http://stackoverflow.com/questions/6258796/gson-java-reserved-keyword