# 1 Parsing trees

## 1.1 Introduction

In this exercise a concrete syntax tree is transformed into an abstract syntax tree of some language. Imagine a parser that recognizes a language according to a given BNF grammar. The result of the parser execution could be a model according to a generic concrete syntax tree metamodel. To create the abstract syntax tree of the language, a model-to-model transformation need to be defined. Thus, for each grammar recognizing a given language, the corresponding model transformation will be in charge of creating the specific abstract syntax tree of such a language.

The metamodel shown in 1(a) will be used to represent the concrete syntax tree of any language recognized by our imaginary parser. For instance, if we have the following language, called "struct", to represent simple data structures:

```
struct Person {
  fullname : String;
  address  : String;
  age      : Integer;
}
```

the corresponding concrete syntax model will be the one shown in 1(b). A `Leaf` object typically corresponds to a token recognized by the parser, while `Node` corresponds to a non-terminal rule application, which probably will have one or more children (either nodes or leaves).

You can assume that, for the "struct" language, the following constraints are satisfied for any correct program:

```
context CST::Node do
  inv 'struct' do
    (self.kind == 'struct').implies(
      self.children.select { |n| n.kind_of? CST::Leaf }.size == 2 &&
      self.children.select { |n| n.kind_of? CST::Leaf }.any? { |n| n.kind == 'name' }
    )
  end

  inv 'body' do
    (self.kind == 'body').implies(
      self.children.all? { |n| n.kind_of? CST::Node } &&
      self.children.all? { |n| n.kind == 'attribute' }
    )
  end

  inv 'attribute' do
    (self.kind == 'attribute').implies(
      self.children.size == 2                         &&
      self.children.all? { |n| n.kind_of? CST::Leaf } &&
      self.children.any? { |n| n.kind == 'name' }     &&
      self.children.any? { |n| n.kind == 'type' }
    )
  end
end
```

```
context CST::Leaf do
  inv 'ptype' do
    (self.kind == 'type').implies(
      ['String', 'Integer', 'Any'].include?(self.value)
    )
  end
end
```

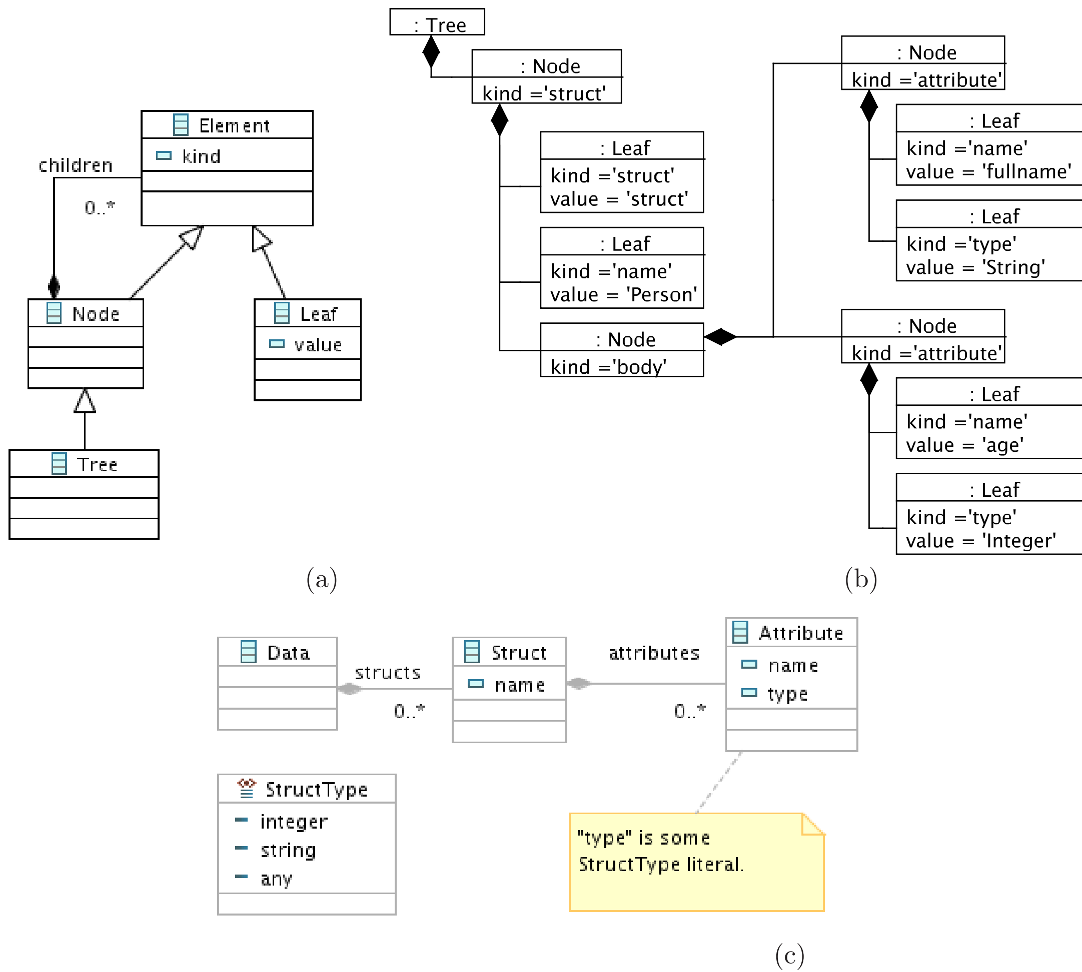


Figure 1: (a) Metamodel for generic concrete syntax trees. (b) Model of a "struct" language program. (c) Metamodel of the abstract syntax tree of the "struct" language.

Figure 1(c) shows the metamodel of the "struct" language. The `Data` metaclass aggregates one or more *struct definitions*. A `struct` is simply composed of attributes. An `attribute` has a name and a type, which is always one of these: string, integer or any (e.g. a "pointer" to another struct object).

## 1.2 Task

Your task is to write a model-to-model transformation that creates the abstract syntax tree of the “struct” language.

## 1.3 Purpose

The purpose of this example is to learn the following concepts:

- How bindings work.
- How to iterate over collections
- Learn how simple validation rules are written.

## 1.4 Hints

The following may help you to write the required transformation definition.

**Decorators**. Decorators allow you to add methods to some metaclass at runtime. This is useful to factorize code scattered through all the transformation definition. For instance, the following decorator adds a method to the `Node` metaclass to retrieve all children of type `Node` (i.e. reject objects of type `Leaf`).

```
decorator CST::Node do
  def nodes
    self.children.select { |n| n.kind_of? CST::Node }
  end
end
```

**Enumerated values**. To access an enumeration value you must use the `::` syntax: `Package::EnumType::EnumLiteralName`. For instance, in this example you will need to access the `PrimitiveType` enumeration. There are three enumeration literals, that are accessed in the following way:

```
StructAST::StructType::Integer
StructAST::StructType::String
StructAST::StructType::Any
```

Please, note that the literals `integer`, `string`, `any` are defined lowercase in the metamodel. The RubyTL engine makes them uppercase to allow them to be treated as Ruby constants.

**Collections**. Ruby provides functional-like methods to iterate over collections. These *iterators* usually receives a code block, which does something with each value of the collection. The following methods can be useful for this example (in Ruby a collection is defined with `[]`):

```
collection = [1, 2, 3, 4, 5]
collection.select { |i| i % 2  == 0 }   =>   [2, 4]
collection.reject { |i| i % 2  == 0 }   =>   [1, 3, 5]
collection.find   { |i| i == 3 }        =>   3
```

The `select` method returns a collection containing each element that satisfies the condition expressed in the code block. The `reject` method discard elements that satisfy the condition. Finally, `find` returns the first object that matchs the condition.