02332 Compiler Construction
Assignment 2, Syntax and Parsing

Hand-out: 29. October 2019
Due: 22. November 2019 23:55
Hand-in: on DTU Inside Course Page
Group hand-in is allowed, but maximum 4 people per group!
To hand in:

- All relevant source files (grammars and java).
- What your compiler produces on the test examples.
- A small (1-2 page) report in PDF format that documents what you did for each task (including answers to questions of the task), possibly with code snippets. Please state clearly the name of all group members (names and student number) with the task on the title page of the report.

The Project

The goal of this project is to write part of a "*Compiler Compiler*" (or "*coco*" for short), a tool for writing compilers and interpreters:

- Input:
  - The description of the abstract syntax of a language (that one wants to write a compiler or interpreter for) as a collection of algebraic datatypes (that may be mutually recursive)
  - For each datatype, one can provide a function (e.g. interpreter, type-checker, or translation) in Java.
- Output: a Java program that implements the algebraic datatype and the functions on it.

The input for *coco* is itself a small language, that is already provided in the archive `coco.v1.zip` that you find on DTU Inside. It contains:

- An ANTLR grammar for *coco*'s input language `coco.g4`,
- the abstract syntax tree for *coco*'s input language in Java `AST.java`,
- `main.java` that implements a visitor class producing data structures of `AST.java`.

You will *not* have to modify the ANTLR grammar, the visitors, or the class structure of the AST – all this is already done for you. The entire task you have to do is implementing a suitable method `String translate()` for all the classes in the AST.

Example 1. *We first explain the* coco *input language at hand of an example. The example is a language for arithmetic expressions similar to the first examples of the course:*

```
DATA expr                WITH { Double eval(Environment env) }
= Constant(Double v)          { return v; }
| Variable(String name)   { return env.getVariable(name); }
| Mult(expr e1, expr e2) { return e1.eval(env) * e2.eval(env); }
| Add (expr e1, expr e2) { return e1.eval(env) + e2.eval(env); }
;
```

Ignoring first all the expressions in braces, this defines an algebraic datatype `expr` *in the style of a functional language: an* `expr` *is either a* `Constant` *(containing a* `Double`*) or a* `Variable` *(containing a* `String`*) or a* `Mult` *of two* `expr` *or an* `Add` *of two* `expr`*. Thus this is standard abstract syntax.*

This should be translated into Java as a collection of class definitions similar to other examples:

```
abstract class AST{}
abstract class expr extends AST {};
class Constant extends expr{
  Double v;
  Constant(Double v){ this.v=v; }
}
class Variable extends expr{
  String name;
  Variable(String name){ this.name=name; }
}
class Mult extends expr{
  expr e1;
  expr e2;
  Mult(expr e1, expr e2){ this.e1=e1; this.e2=e2; }
}

class Add extends expr{
  expr e1;
  expr e2;
  Add(expr e1, expr e2){ this.e1=e1; this.e2=e2; }
}
```

Note that `class expr` *is* `abstract` *because we will have a function that can only be implemented for the concrete child classes. Also we have put an abstract class* `AST` *on top, so every data type is a descended of them.*

Next, the statement `WITH { Double eval(Environment env) }` *in the input text declares that we wish to define a method* `eval` *for every* `expr` *that gets as input an* `Environment` *and returns a* `Double`*. The datastructure* `Environment` *is our own definition, we use the same as in many examples before, namely a* `HashMap` *from* `String` *to* `Double`*, i.e., storing for each defined variable name a* `Double` *value.*

For each case of the `expr` *datatype, an implementation fo the* `eval` *function is given, e.g. for* `Add` *we have* `{return e1.eval(env) + e2.eval(env); }`*. The idea is that this becomes a method of the* `Add` *class:*

```
public  Double eval(Environment env)  { return e1.eval(env) + e2.eval(env); }
```

Of course, the class `expr` *has instead the abstract method:*

```
abstract public Double eval(Environment env);
```

The full example is found in the files `coco_input.txt` *and* `coco_output_expected.java`*. A larger example (the entire interpreter from the first assignment) is found in the example folder as* `interpreter.coco` *and* `expected-interpreter.java`*. Please read the README file of the archive to get more information on how to run the examples and testing your implementation.*

Task 1

The main part of *coco* is to generate Java code according to examples:

- Generating all the classes in Java to implement the given algebraic datatype of the input specification.
- Inserting all implementation of the described method for each class.

Test you implementation on the given examples, including `bigtest` (in the example folder).

Task 2

The `AST.java` of coco is itself an abstract syntax with a translation function, and this could itself be written in coco, i.e., so that coco can produce much of its own code. Implement a file `coco.coco` in coco syntax that generates as much as possible of coco, i.e., of the `AST.java` file, i.e., the AST classes and the translation function. Make necessary re-arrangements in the other source files of coco, so you get a working compiler out of this that uses the produced code from `coco.coco`.

Task 3 (voluntary/no points)

Consider the interpreter for a functional language developed in week 6. Could the AST datatype and its children together with the implementation of the `eval` function be described as coco-file?

Explain:

- What classes would actually be generated by coco? (And which ones would be given?)
- Would coco generate additional classes?
- Do we need to make any other changes to handle this example?