02332 Compiler Construction
Mandatory Assignment: A Simple Interpreter

Hand-out: 14. September 2021
Due: 15. October 2021 23:59
Hand-in via Learn platform in groups of 3-4 people
To hand in:

- All relevant source files (grammars and java)
- Your 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.

Week 3

The initial task – if you have not yet done it already – is to download and install ANTLR for Java. When working in groups, we expect every group member to have a running installation on their own machine.

- On DTU Learn, we provide a starting point for your project, an archive called `impl` – a simple imperative language. It consists of the following files:
  - `Makefile` contains all necessary compilation commands, so you do not need to worry about the commands to run ANTLR and JAVA. You can to compile everything with `make` and `make test` to run a sample input to the calculator. Whenever you change anything, another `make test` causes the re-compilation of only those files that are affected by your changes.
  - `impl.g4` the ANTLR grammar. (When running ANTLR without the use of the `Makefile`, run ANTLR with the option `-visitor`.) This generates the files:
    - `implLexer.java` – the lexer for the calculator,
    - `implParser.java` – the parser for the calculator,
    - `implVisitor.java` – this file defines an interface `implVisitor` which is the key to actually do something useful with a parse tree that the parser returns.
  - `main.java` The actual main program that reads an input file given as command line argument, feeds it to the lexer and then to the parser. Finally, it implements the `implVisitor` interface in a class called `AstMaker`. This visits the parse tree and transforms into an Abstract Syntax Tree, i.e., giving a bit more high-level syntax.
  - `AST.java` constaints the definition of the Abstract Syntax Tree, i.e., Java classes for every syntactic construct. They all have an evaluation function (with different return types) which implement the corresponding piece of the interpreter.
  - `Environment.java` defines an environment, a mapping from variable names to Double values, i.e., describing the memory state of a program. An environment this is an argument to every `eval` method in the `AST` classes.
  - `impl_input.txt`. A sample input to the program.
  - `impl_additional.txt`. Another sample that showcases the additional features of the language you shall implement in the following tasks. So this input **does not yet run** with the given interpreter, but it should run once you have finished the project.

Please try to compile and run the interpreter, and play a bit with it, trying out different inputs. For all the above installation, there is nothing to hand in.

Task 1: The given language supports for arithmetics only addition and multiplication. Please extend it with subtraction and division by modifying the given example. You MUST NOT modify any of the ANTLR-generated files, but only the grammar and the main java file.

Test the calculator on several inputs – does it always compute the desired result, especially when we mix several operators without parentheses?

Task 2: This task is for extending the grammar of the language first "on paper", i.e., without trying to get it into ANTLR (we do that in another task below). Also we do not worry about operator precedence and associativity at this point. Please consider the file `impl_additional.txt` for this as it showcases the new features you are asked to implement in this project.

The new features to implement are the following:

- Conditional branching like `if (x==0 || y<z) then z=1;`
  where conditions can contain comparison of expressions (equal, smaller,...), and Boolean connectives (and, or, not). You can also implement an optional else-clause if you like.
- For-loops of the form `for(i=2..n)` where we limit the flexibility of the for loop as compared to languages like Java and C++ where where can make more complex conditions and updates: here we simply specify two expression that denote the minimum (in the example 2) and maximum (here n) value of the variable of the for loop. If you like you can instead implement a more Java/C++ style syntax.
- Arrays of the form `a[e]` where `a` is the name of the array and `e` is the index in the array to read or write. (We do not consider any complicated arrays like multi-dimensional arrays or arrays of arrays.)

Note that in this language, we do not make declarations of variables or arrays (or their sizes), but if a program tries to read a variable or array index that has not been assigned a value before, then this should give to a runtime error.

For instance

```
n=5;
for(i=1..n) a[i]=1;
for(i=1..n+1) a[i]=2*a[i];
```

would lead to a runtime error in the second loop when $i = n + 1$, because $a[n + 1]$ has not been assigned to yet.

You are welcome to design the concrete syntax to your personal taste, as long as all above concepts are supported (see `https://en.wikipedia.org/wiki/LOLCODE` for a rather bizarre example).

This task is to specify the syntax of the language as a context-free grammar. Moreover, please give for each construct an example of correctly used syntax.

Week 4

Task 3: Implement the extensions from task 2 in the actual project, i.e., extend the `impl.g4` file. Check that all examples of **Task 2** can indeed be accepted by the parser.

For each syntactic construct you have added in the previous week you now need to define a new method for the class `AstMaker`: a visitor for the new syntactic construct. It will result in a compile error if your impl.g4 defines new constructs and the AstMaker does not have a corresponding method. You can see what method the AstMaker must provide by looking at the ANTLR-generated file `implVisitor.java`. The type parameter `<T>` becomes `AST` and also each method needs to have the modifier `public`.

Before these methods can actually return something meaningful, one has to first define corresponding abstract syntax, which is part of next week's task. You can do that in one go, or for now just give a trivial implementation of the new methods by simply `return null;` — just so that it all compiles.

Extensively test the workings of your parser, in particular check that the operators in arithmetic expressions and Boolean conditions have the right precedence:

- multiplication and division bind stronger than addition and subtraction
- "not" binds stronger than "and", "and" binds stronger than "or".
- all the binary operators associate to the left.

For checking this, you should develop small test cases and document them in your report. It is easiest to check that with `grun`.

Week 5

Task 4 This week's task is to implement the Abstract Syntax Tree for the new constructs, including the corresponding `eval` methods.

To implement is thus a new class in `AST.java` to represent your new syntactic constructs, i.e., what the visitor from the previous week should return. It is best to first think of what data the construct should hold, i.e., what member variables should it have, and then how the `eval` method can be implemented.

For the implementation of `eval` for arrays, one can use the following simple trick: treat an expression like `a[5]` simply as a variable name, i.e., `"a[5]"` that can also be stored in `env`. However, if an expression contains variables like `a[i+3]`, you first need to evaluate `i+3`.

Task 5 is to connect the new AST with the visitors, i.e. to replace in the visitors the `return null;` from the previous week with generating elements of type `AST`. Note that due to limitations in Java's type system, the `visit` method will always return something of type `AST`, and one may need to do a type-cast like `(Expr)visit(ctx.e)` to tell Java that the result should actually be of the subclass `Expr`. (Java will test this at runtime, but by construction we will know this will always be correct.)

Week 6

Task 6 Implement a simple type checker similar to the lecture, namely a method `typecheck` for each class of `AST`. This should visit each part of the code once and check that each variable first occurs on the left-hand side of an assignment and whether it is used as an array or a double. If it gets used anywhere in the code later in a different way, it is a type error. For example

```
x=y;       // Error: y not defined
y=1;
z[1]=y[1]; // Error: y was defined as double, now used as array
z=1;       // Error: z was defined as array, now used as double
```

Week 7

Lab week to complete the implementation and report.