Exercise 02: Grammars and ASTs

Compiler Construction (Fall '25)

Discussion: Sep, 23

1. (Theory) Grammar for Boolean Expressions

Create a grammar for Boolean expressions. It is composed of three main types of terminals

- operands: it should allow varying *identifiers*, and the literals `true` and `false`.
- operators: `AND`, `OR`, `NOT`, where `NOT` has highest precedence followed by `AND`. Thus, `OR` has the lowest precedence.
- brackets: it should be possible to change the precedence by using brackets (`(` and `)`).

Explicitly define the sets N, T, and P, and the grammar tuple G.

Create a parse tree for the following expression to check your grammar:

```
(small OR big) OR medium AND big AND NOT small
```

2. (Practice) AST for SPL programs

Now, that we have implemented the lexical analysis for SPL programs (c.f., Exercise 01), we can prepare the parsing.
Before we implement a parser, we need the data structures to store the result of the parsing, the AST. An AST should

- be concise (focus on operators and operands and their relationships)
- be easy to iterate
- avoid storing redundant and irrelevant information
- store knowledge which we may need for the semantic analysis (e.g., for type-checking or scoping)
- be robust against small changes in the grammar

In this task, you should design the required Java classes for an AST representing SPL programs.

(a) Take a piece of paper and create the parse tree for some of the SPL's statements and expressions. A grammar for SPL is available in Listing 1.

Consider which parts of the tree are redundant and represent irrelevant information.

(b) Now, implement the resulting datastructure and build the corresponding AST through your implementation. AST nodes can be implemented as immutable data objects (i.e., having *final public fields*). Later, the parser creates the AST but it is a good exercise to test our understanding of ASTs.

Notes:

- The SPL grammar lays the grounds for this task but there is some freedom in designing the AST structure. Remember the discussion of (concrete) parse tree vs. (abstract) syntax tree and think of which information might be relevant during later semantic analysis.
- It is reasonable to distinguish statements from expressions, each of which should form abstract top-level classes within the inheritance hierarchy that classifies the various kinds of AST nodes.
- Many kinds of expressions may be summarized as binary expressions.
- While it is useful to distinguish logical expressions, we do not have to distinguish logical `or` from `and`.

3. Visitor Pattern and AST printer

To understand and debug our implementation, implement an visitor that can print the AST. Your task is to implement a concrete visitor `AstPrinter` that converts an AST into a String. The String shall be written to the console.

As the AST nodes shall be immutable data objects, we need some other facility to operate on them. The *Visitor Pattern* (cf. Exercise 00) can serve as basis to separate the concerns of storing data and performing operations on the data. In this task, it helps leaving the AST nodes immutable. We encounter more kinds of operations to be performed on the AST when we arrive at the semantic analysis.

```
program        := declaration* EOF ;

declaration    := varDecl | statement ;

varDecl        := "var" IDENTIFIER ( "=" expression )? ";" ;

statement      := exprStmt
                | whileStmt
                | ifStmt
                | printStmt
                | block ;

exprStmt       := expression ";" ;

ifStmt         := "if" "(" expression ")" statement ( "else" statement )? ;

printStmt      := "print" expression ";" ;

whileStmt      := "while" "(" expression ")" statement ;

block          := "{" declaration* "}" ;

expression     := assignment ;

assignment     := IDENTIFIER "=" assignment | logic_or ;

logic_or       := logic_and ( "or" logic_and )* ;

logic_and      := equality ( "and" equality )* ;

equality       := comparison ( ( "!=" | "==" ) comparison )* ;

comparison     := term ( ( ">" | ">=" | "<" | "<=" ) term )* ;

term           := factor ( ( "-" | "+" ) factor )* ;

factor         := unary ( ( "/" | "*" ) unary )* ;

unary          := ( "!" | "-" ) unary | primary;

primary        := "true" | "false" | NUMBER | STRING | IDENTIFIER
                | "(" expression ")";
```

Listing 1: “A grammar for SPL”