Go Programming - Parsers in Go

Concepts of Programming Languages

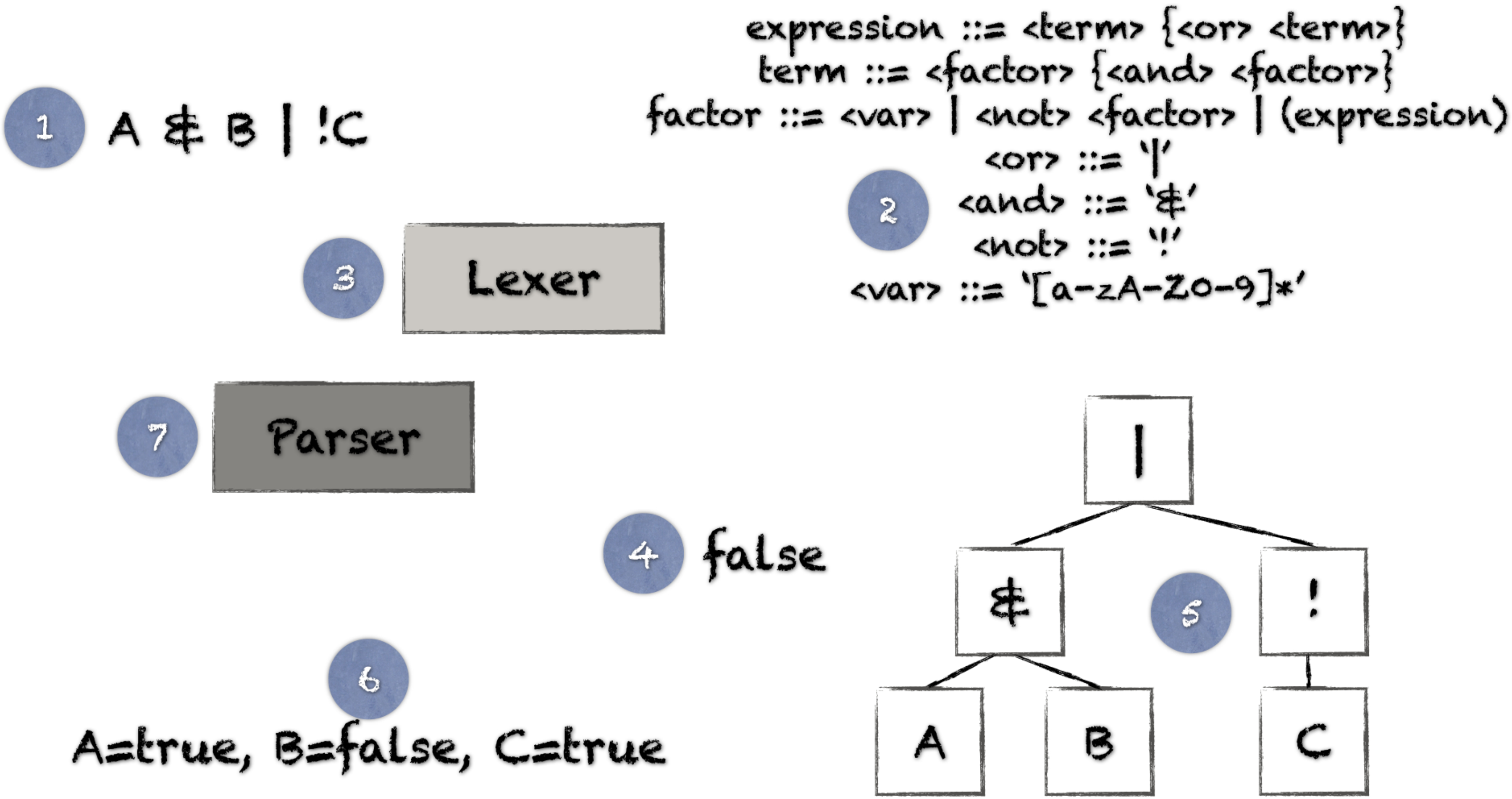
26 October 2020

Bernhard Saumweber
Rosenheim Technical University

Focus of today's lecture: Practice your GO-Skills

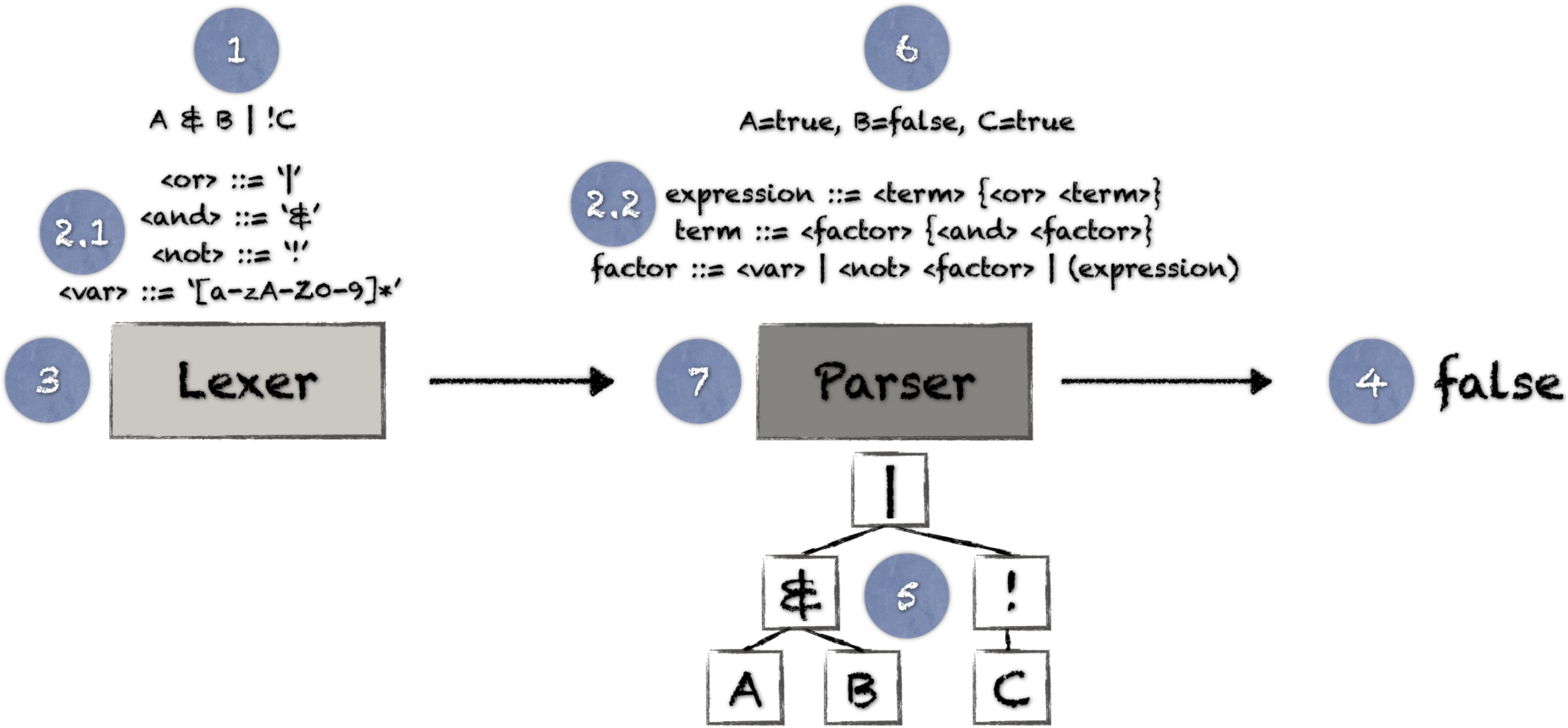
- on something new what is called *parsers*

New things you will learn today and which you should remember!



- Could you describe or name the things? How are these things connected?

What we want to achieve: Evaluating boolean expressions with GO



Three building blocks: Grammar

- Grammar: Defines *Lexer* and *Parser* rules (e.g. Backus-Naur)

```
expression ::= <term> {<or> <term>}
term ::= <factor> {<and> <factor>}
factor ::= <var> | <not> <factor> | (expression)
<or> ::= '|'
<and> ::= '&'
<not> ::= '!'
<var> ::= '[a-zA-Z0-9]+'
```

- Parser rules: Defines how the *Abstract Syntax Tree* or *Parse Tree* is build

```
expression ::= <term> {<or> <term>}
term ::= <factor> {<and> <factor>}
factor ::= <var> | <not> <factor> | (expression)
```

- Lexer rules: Defines how the *tokens* are determined

```
<or> ::= '|'
<and> ::= '&'
<not> ::= '!'
<var> ::= '[a-zA-Z0-9]+'
```

Three building blocks: Lexer

- Performs the lexical analysis and tokenize the input into **n** *tokens*

```
lexer.tokenize("A & B | !C") ->  ["A", "&", "B", "|", "!", "C" ]
```

- *Tokens* are based on the grammar and are consumed by the *parser*
- Simple lexer example:

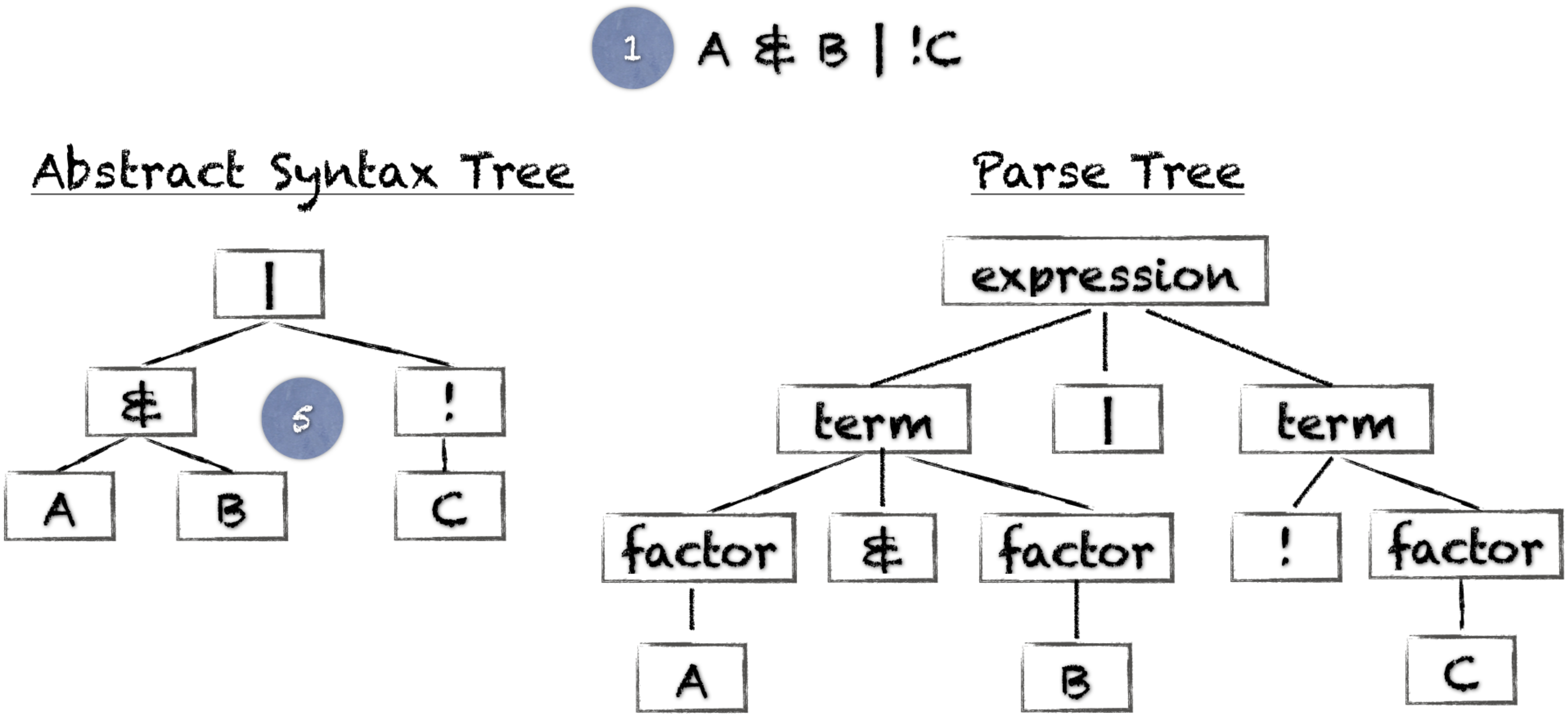
```
switch currentChar {
// check for terminal
case byte('&'), byte('|'), byte('!'), byte('('), byte(')'):
    if token != "" {
        result = append(result, token)
        token = ""
    }
    result = append(result, string(currentChar))
    break
// var assumed
default:
    token += string(currentChar) // concat var chars
}
```

Implement a lexer for boolean expressions

- [(see Exercise4.md)](https://github.com/0xqab/concepts-of-programming-languages/blob/master/docs/exercises/Exercise4.md)

Three building blocks: Parser

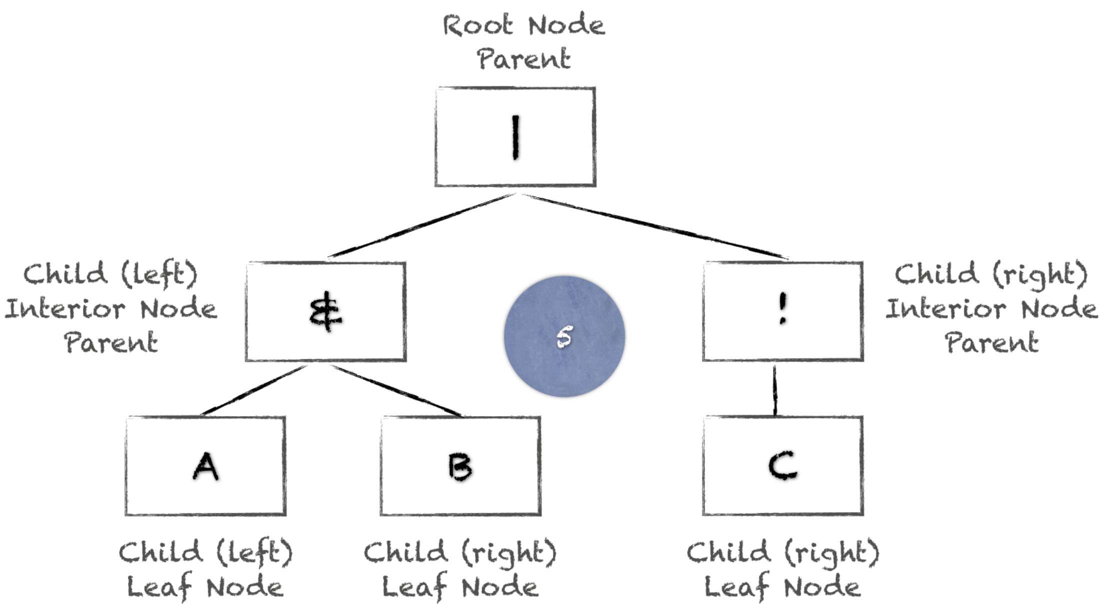
- Builds an *Abstract Syntax Tree* (AST) or *Parse Tree* from the tokens



- AST represents the abstract syntactic structure (does not contain all the details)

Abstract Syntax / Parse Tree terminology

- A *tree* consists of hierarchically organized *nodes* and has a *root* node
- Nodes below the root are *child* nodes
- All nodes except the root have a unique *parent*
- A Node that has children (1..N) and is not the root is called an *interior* node
- A node without children is a *leaf* node



We will implement a recursive descent parser to build the AST

- Top-Down: identity root first, move down the subtrees, until leaves are found
- LL-Parser: Left-to-right (input); Leftmost derivation (replace left most non-terminal (leaf))
- Usually one method for each rule of the grammar -> structure of the parser mirrors the structure of the grammar

Algorithm (high level):

- Implement methods, one for each rule
- Start from the main top rule of the grammar (*expression*)
- Call rules (*term*, *factor*) as defined in the grammar until we have a *var* token.
- Step back to 3. and analyze the next token

How to implement this in Go? First Step: AST

- Define a *Node* which can evaluate a *map[string]bool*

```
type node interface {
    Eval(vars map[string]bool) bool
}
```

- Implement interior nodes: *OR*, *AND*, *NOT*

```
type or struct {
    lhs node
    rhs node
}
func (o *or) Eval(vars map[string]bool) bool {
    return o.lhs.Eval(vars) || o.rhs.Eval(vars)
}
```

- Implement leaf nodes: *var*

```
func (v *val) Eval(vars map[string]bool) bool {
    return vars[v.name] // missing vars will be evaluated to false!
}
```

How to implement this in Go? Second Step: Parsing

- Implement all parser rules: *expression*, *term*, *factor*
- The rule *expression ::= <term> {<or> <term>}* as code:

```go
func (p *Parser) expression() {
    p.term() //an expression always has a term
    for p.token == "|" { //maybe the term is followed 'or' and another term
        lhs := p.rootNode
        p.term()
        rhs := p.rootNode
        p.rootNode = &or{lhs, rhs}
    }
}
```

- Put everything together

```go
p := NewParser(NewLexer("a & b | !c"))
vars := map[string]bool{"a": true,"b": true,"c": false,}
p.Eval(vars) // true
```

Implement a parser for boolean expressions

(see Exercise4.md)(https://github.com/0xqab/concepts-of-programming-languages/blob/master/docs/exercises/Exercise4.md)

Check

What is: "A & B | !C"?

- The first three letters of the alphabet
- A grammer to parse expressions
- A boolean expression with placeholders (A, B, C)

And now... Antlr

- 1: Get Antlr.

```
go get github.com/antlr/antlr4/runtime/Go/antlr
```

- 2: Define Grammar

```
grammar bool;
expression : term | OR expression
term : factor | AND factor ...
```

- 3: Generate Lexer and Parser

```
antlr4 -Dlanguage=Go MyGrammar.g4
```

- 4: Use the generated code

```
bool_parser.go, bool_lexer.go, ...
```

(Quickstart Antlr4 for GO)(https://github.com/antlr/antlr4/blob/master/doc/go-target.md)

Summary

- Parsing is fun
- Three building blocks: Grammar, Lexer, Parser
- Use parser generators - they are more stable than your code

Bibliography

- Books

Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (https://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)

Writing Compilers and Interpreters: A Software Engineering Approach (https://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=UCLGQTPIYSWYKRRM)

- Blogs / Links

ruslanspivak.com/lsbasi-part7/ (https://ruslanspivak.com/lsbasi-part7/)

github.com/antlr/antlr4/blob/master/doc/go-target.md (https://github.com/antlr/antlr4/blob/master/doc/go-target.md)

tomassetti.me/guide-parsing-algorithms-terminology/ (https://tomassetti.me/guide-parsing-algorithms-terminology/)

Thank you

Bernhard Saumweber
Rosenheim Technical University
bernhard.saumweber@qaware.de (mailto:bernhard.saumweber@qaware.de)
http://www.qaware.de (http://www.qaware.de)