Parser Combinators in Go

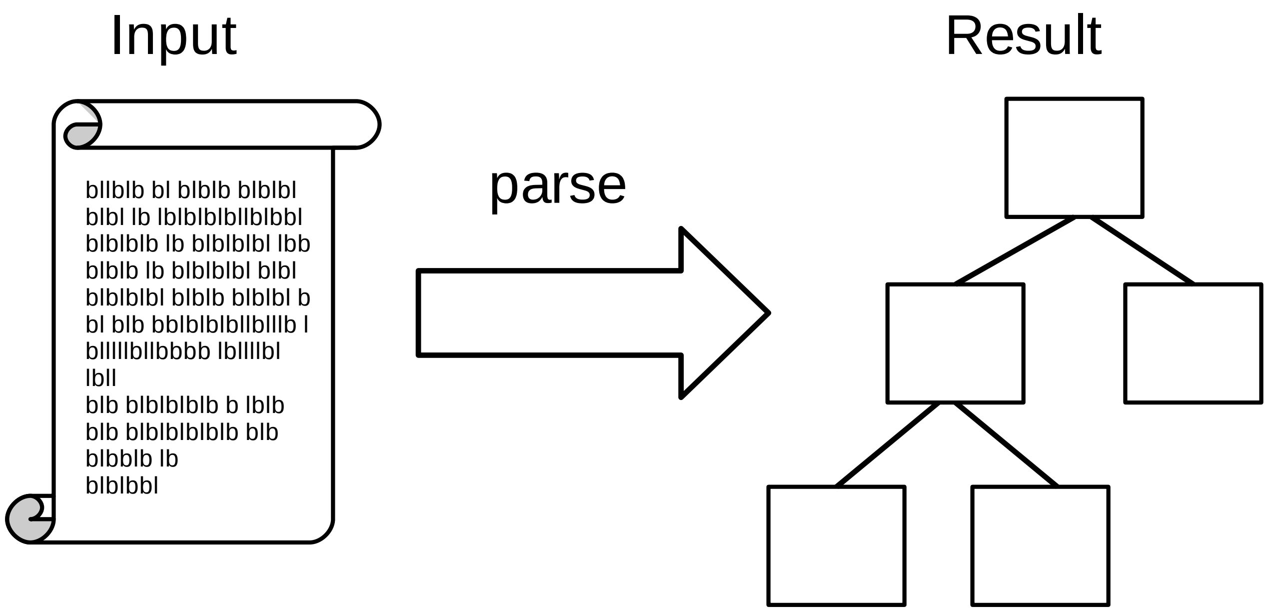
Concepts of Programming Languages
8 November 2018

Armin Heller on behalf of Johannes Weigend (QAware GmbH)
University of Applied Sciences Rosenheim

What's a parser?

- A parser is a function converting an **input** to a **result**.

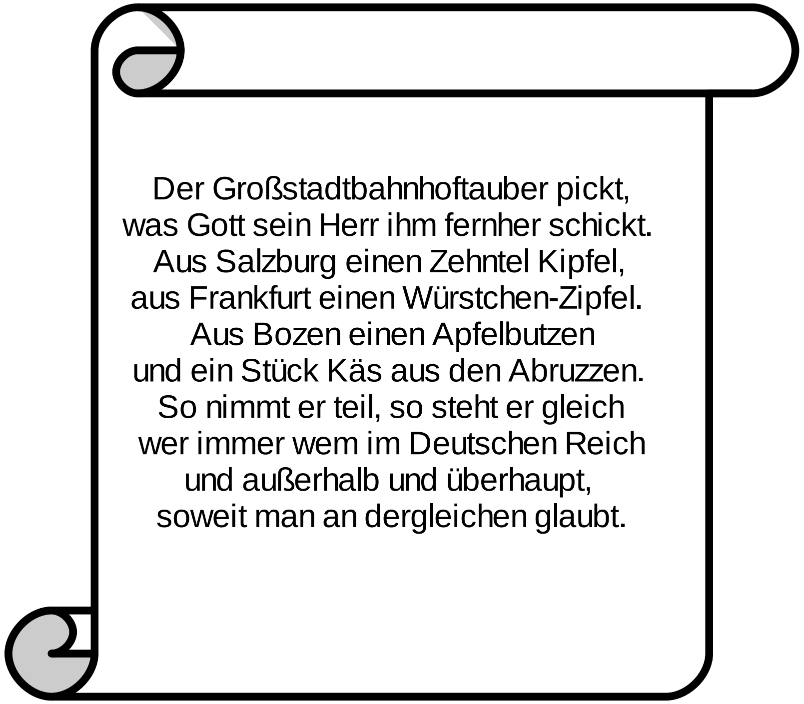
```
func parser (input ParserInput) ParserResult {
  ...
}
```



2

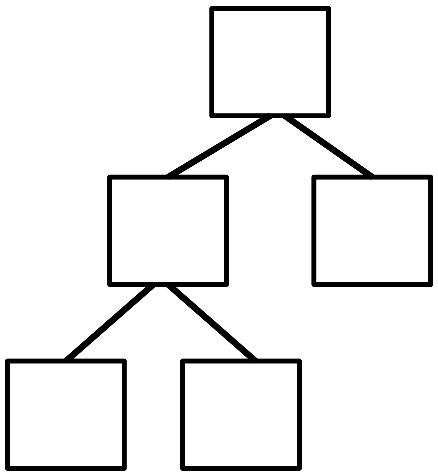
What's the input of a parser?

- The input of a parser is text.

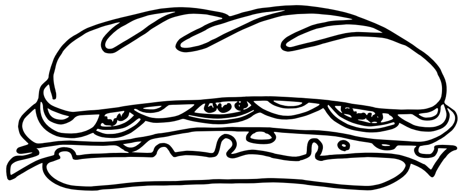


What's the output of a parser?

- Syntax trees
- Numbers
- Results of calculations
- etc.



or



or 42

4

A Parser ..

- .. is a function converting an **input** to a **result**.

```
func parser (input ParserInput) ParserResult {
  ...
}
```

- .. takes text as an argument.
- .. returns whatever we can and want to make it return, e. g. trees, numbers or boolean values.

5

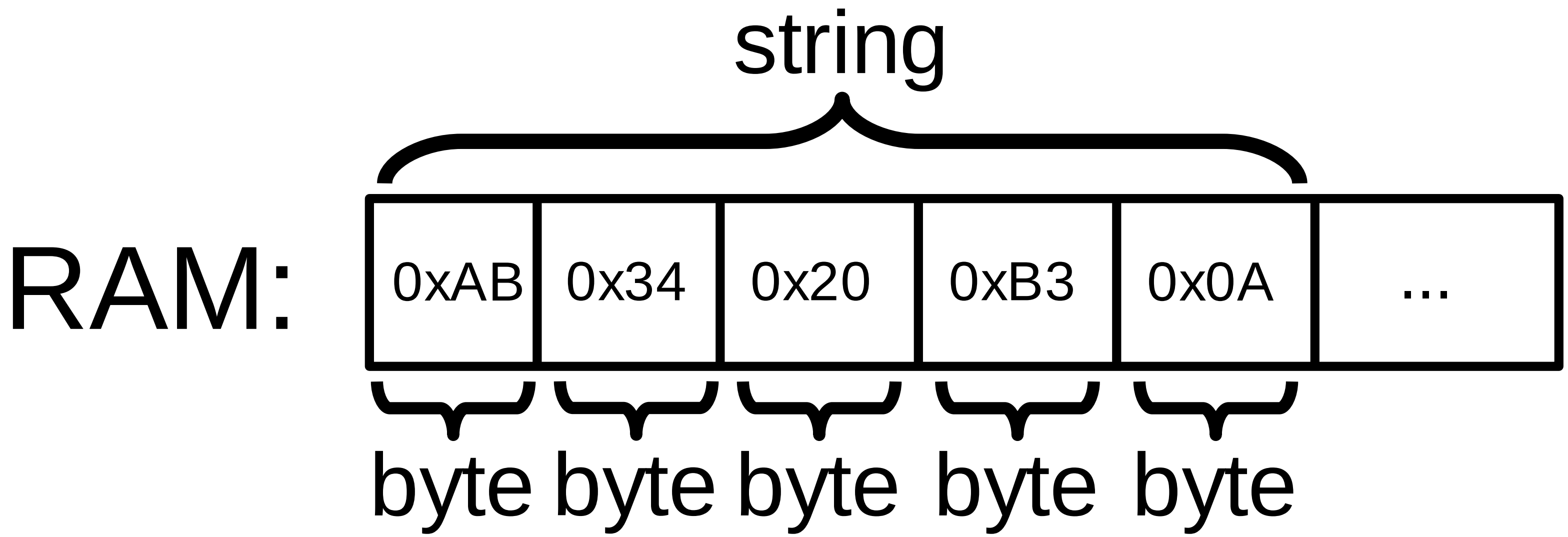
What is text in Go?

- We could use `string`
- We could use `[]rune`, (rune is just a unicode code point)
- We could use `os.File`
- We could use our own interface

6

Using string as input?

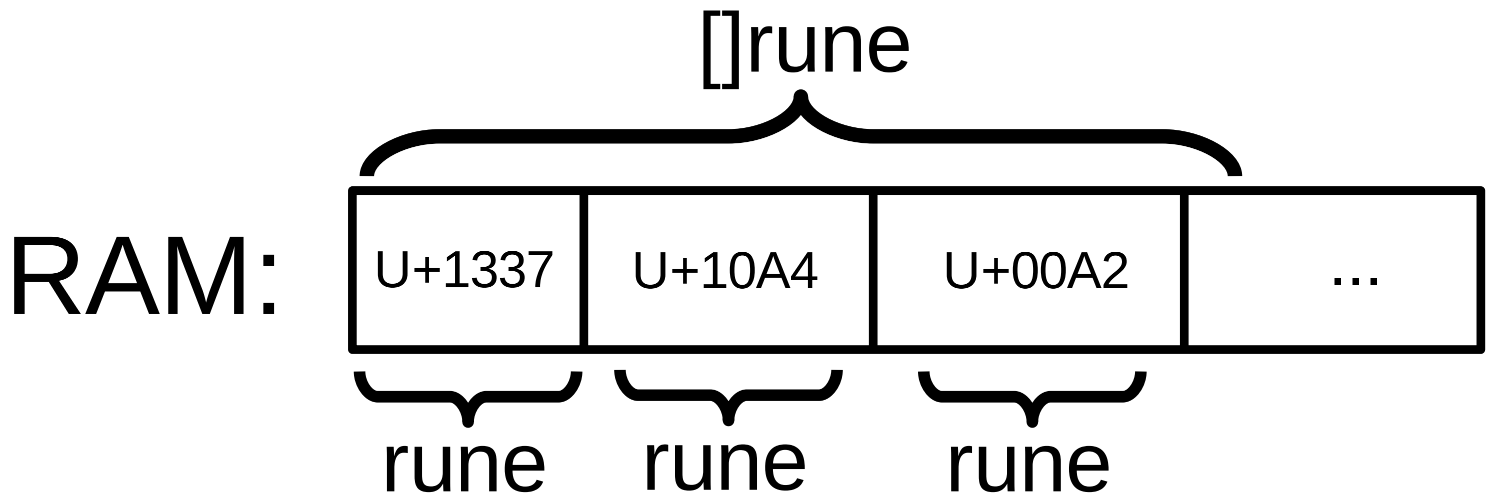
- Bad: `s[i]` will only give you bytes (except in range loops)
- Good: it's a built-in type with many library functions
- Possible Solution: `[]rune`



7

Using []rune as input? (runes are unicode code points)

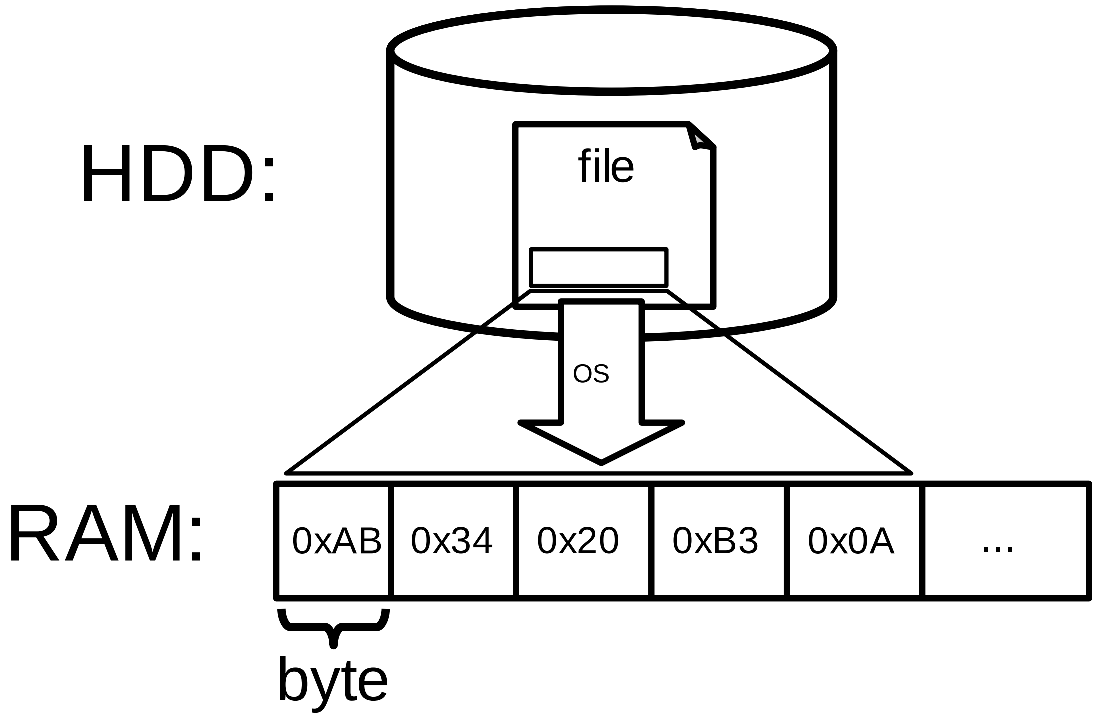
- Bad: The whole input needs to be in memory, just like with strings
- Good: Arrays are fast
- Bad: What if we want to read a file that doesn't fit in memory?
- Possible Solution: `os.File`



8

Using os.File as input?

- Good: Files don't have the RAM limitation
- Bad: Slow access
- What if we want to read from a socket? Can we still use `os.File`?



9

Problems of the built-in types

- `string`: in-memory, byte-level indexing
- `[]rune`: in-memory
- `os.File`: slow access
- `net.`TCPConn: slow access
- All of the above are inflexible, i. e. they're not covering all possible use-cases.
- We want to choose the correct input type depending on the use-case!
- Solution: We write our own interface

10

The parser input is of the following type

```
type ParserInput interface {

  CurrentCodePoint () rune

  RemainingInput () ParserInput

}
```

- We can use `string`, `[]rune`, `os.File` or `net.TCPConn` to implement this
- The implementation mustn't have side-effects!

11

Exercise

```
type ParserInput interface {

  CurrentCodePoint () rune

  RemainingInput () ParserInput

}
```

- Implement this interface using a `[]rune` and an `int` that marks the current position.
- If there's no more input just return `nil`.
- Implement a function `func stringToInput (s string) ParserInput` using your implementation.
- Help your classmates understand the solution if you can.

12

How about the following ParserResult

```
type ParserResult struct {

  Result interface{}

}
```

13

How about the following ParserResult

```go
type ParserResult struct {

  Result interface{}

}
```

- What if we can only parse half of the input?
- How do we communicate what we still have to parse?

14

The parser result is of the following type

```
type ParserResult struct {

  Result interface{}

  RemainingInput ParserInput

}
```

- We mustn't use side-effects on this struct!
- I. e. no field assignments after its construction!

15

Summary

```go
type Parser func (ParserInput) ParserResult

type ParserInput interface {
  CurrentCodePoint () rune
  RemainingInput () ParserInput
}

type ParserResult struct {
  Result interface{} // null iff parsing fails!
  RemainingInput ParserInput
}
```

- Write a parser that parses exactly one letter 'A' from the beginning of an input.

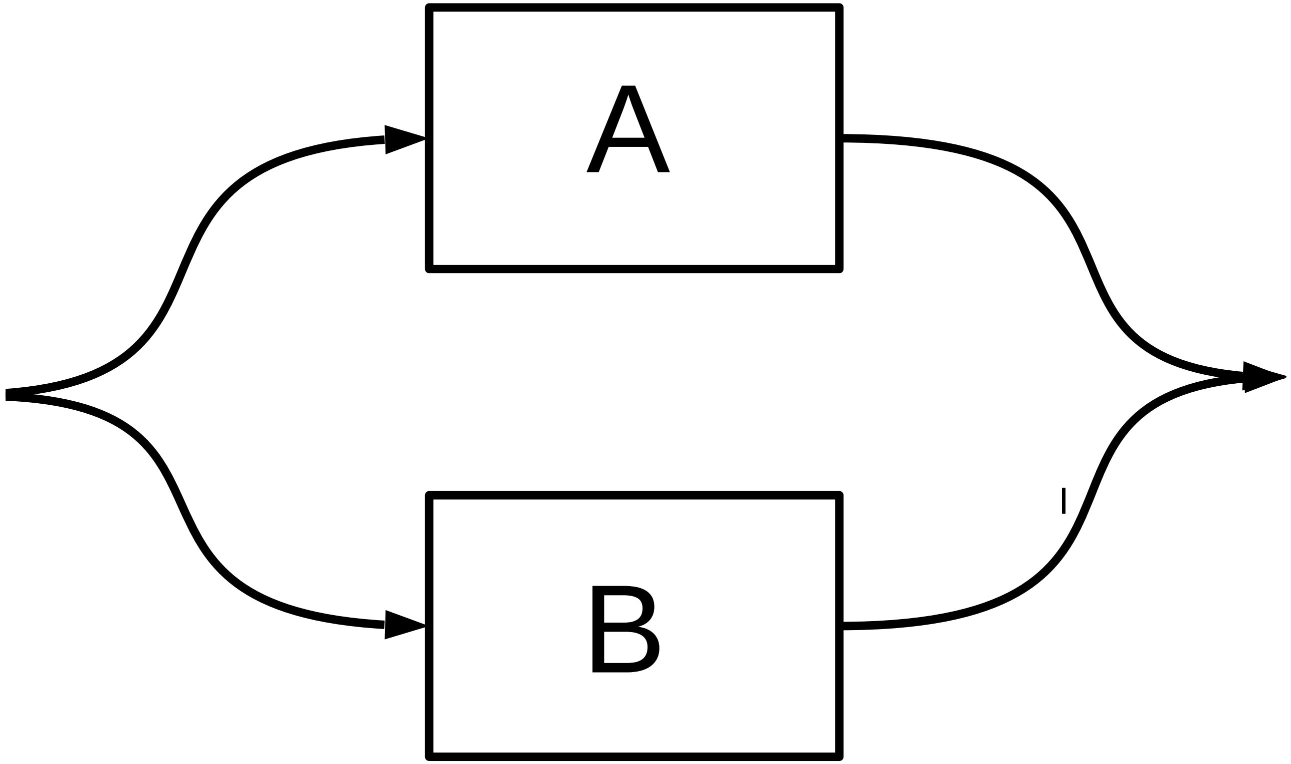
Context-free grammars

- Context-free grammars are recursive regular expressions

<table>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td>A | B</td><td>Parse an A or a B</td></tr>
<tr><td>A ^ B</td><td>Parse an A and parse the remaining input with B</td></tr>
<tr><td>A+</td><td>Repeatedly parse A, at least once</td></tr>
<tr><td>A*</td><td>Repeatedly parse A, zero or more times</td></tr>
<tr><td>A?</td><td>Parse an A or succeed parsing without taking anything from the input</td></tr>
<tr><td>"word"</td><td>Succeed if and only if the input starts exactly with the four letters "word"</td></tr>
</table>

A | B

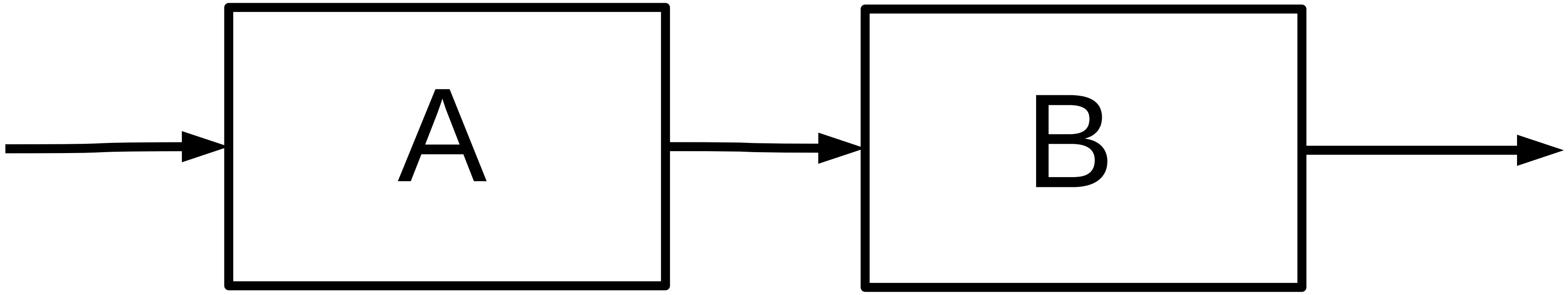
- Parse A or B



18

A ^ B

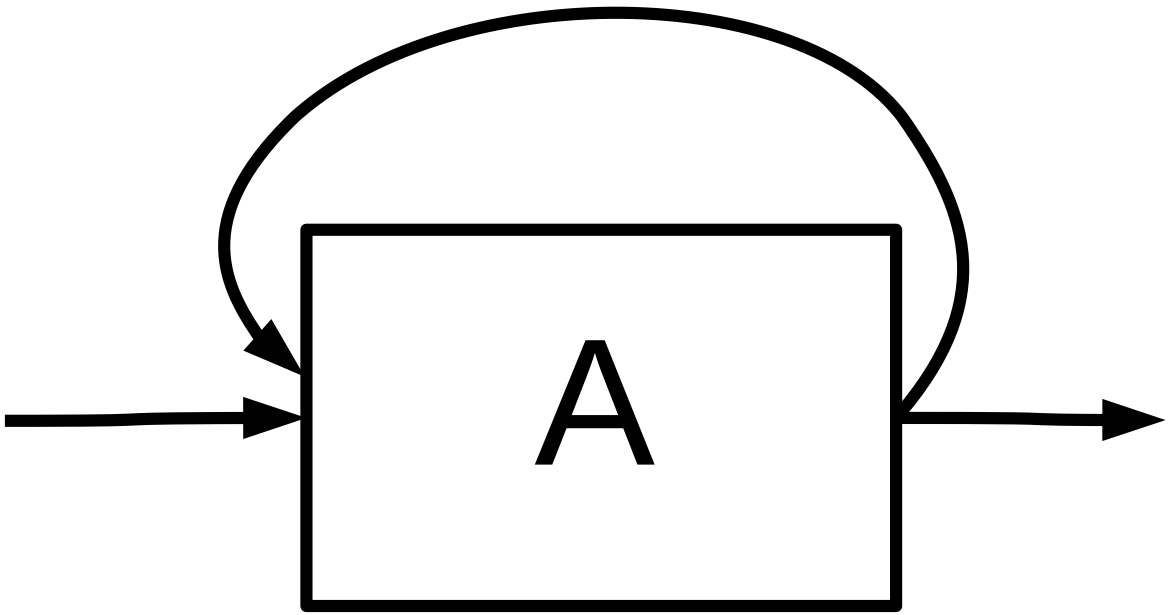
- Parse A and then B



19

A+

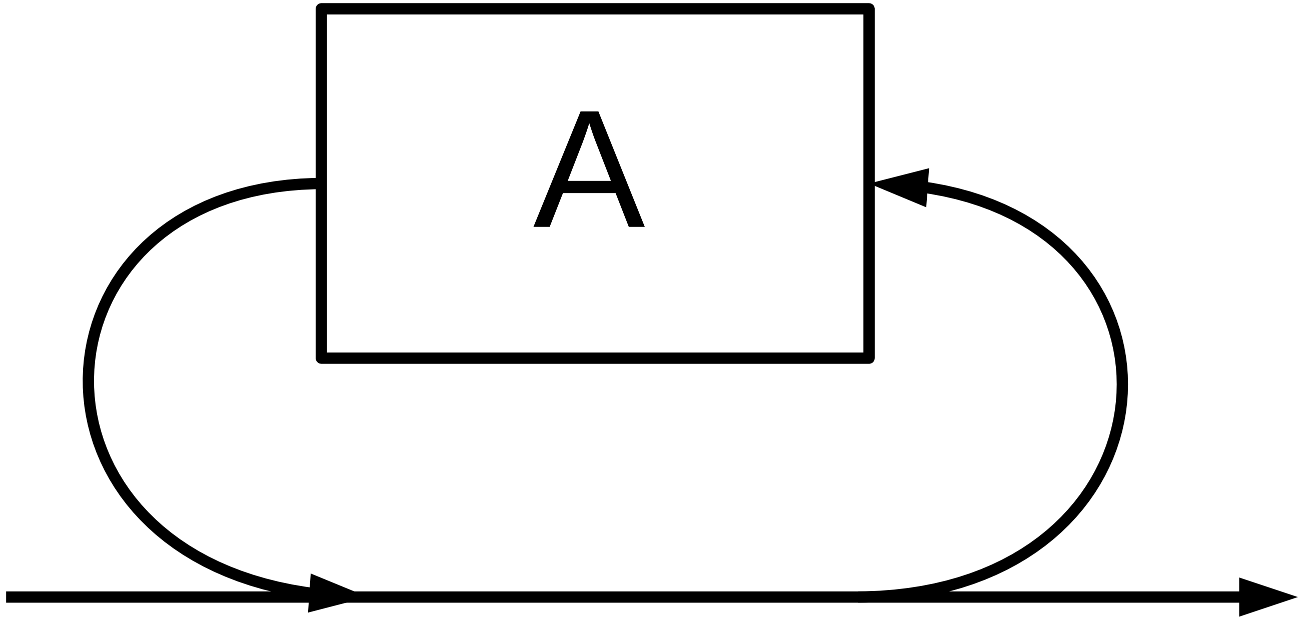
- Parse A once or more



20

A*

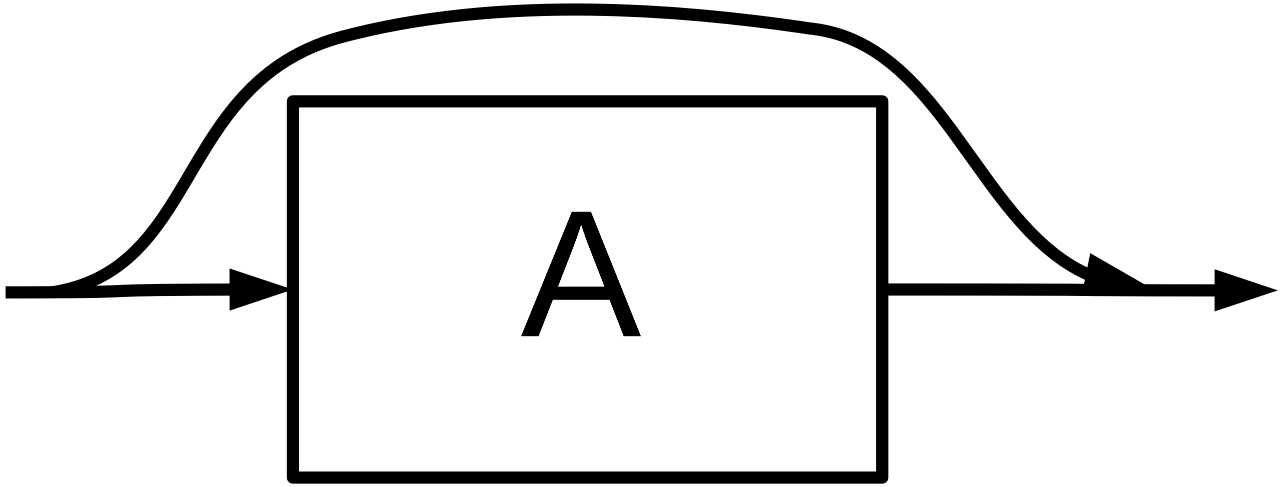
- Parse A zero or more times



21

A?

- Parse A zero or one times



Example Grammar

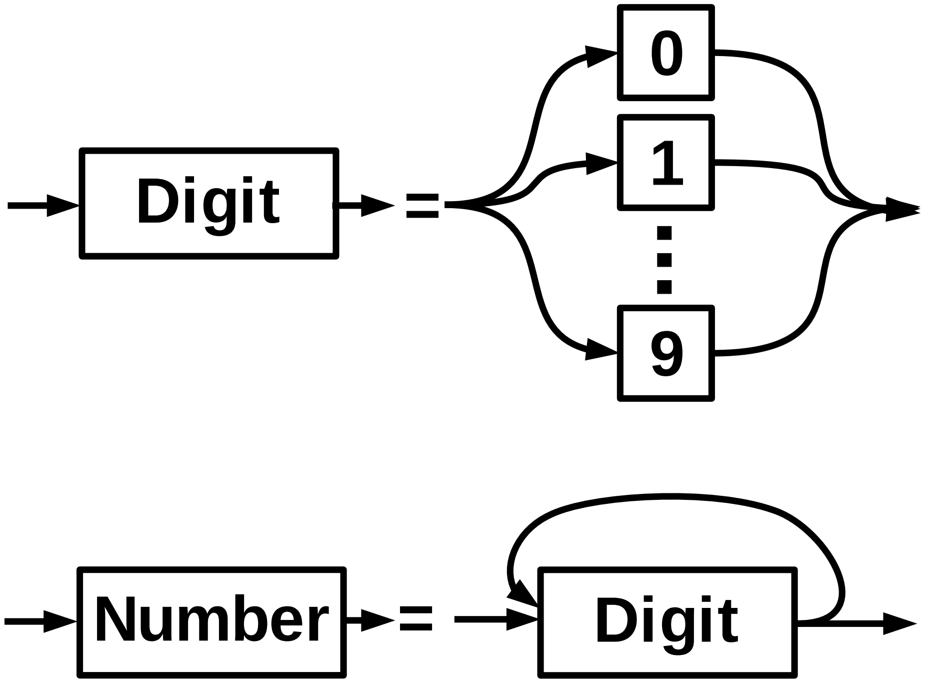
```
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
Number = Digit+

WordStartChar = "a" | "b" | "c" | ..
              | "A" | "B" | "C" | ..
WordChar = WordStartChar | Digit
Word = WordStartChar ^ WordChar*
```

- A Digit is a "0" or a "1" or a "2" or ..
- A Number is one or more Digits
- A WordStartChar is an "a" or a "b" or a "c" ..
- A WordChar is a WordStartChar or a Digit
- A Word is a WordStartChar followed by zero or more WordChars

23

Example Grammar

```
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
Number = Digit+
```



24

The point of this lecture

- We implement the parts of the grammar as functions on parsers.

Operator	Method call	Description
A	B	A.OrElse(B)	Try parsing with A. If parsing fails, try B.
A ^ B	A.AndThen(B)	Parse with A. Parse remaining input with B.
A+	A.OnceOrMore()	Repeat parsing with A.
A*	A.Repeated()	Repeat parsing with A, zero or more times.
A?	A.Optional()	Parse with p or succeed without parsing anything.
"F"	Expect('F')	Succeed iff the input starts with this character.

A | B

- Try to parse A. If that failed, try to parse B.

```
func (a Parser) OrElse (b Parser) Parser {

  return func (input ParserInput) ParserResult {

    var resultA = a (input)

    if resultA.Result == nil {
      return b (input)
    }

    return resultA
  }
}
```

- Limitation 1: If A is a prefix of B, then A will win.
- Limitation 2: When `A|A` fails, `OrElse` will try to parse A twice.

26

A ^ B

- Parse A and with the rest of the input parse B.

```
func (a Parser) AndThen (b Parser) Parser {
  return func (input ParserInput) ParserResult {

    var resultA = a (input)

    if resultA.Result == nil {
      return resultA
    }

    var resultB = b (resultA.RemainingInput)

    if resultB.Result == nil {
      return resultB
    }

    return ParserResult { Pair { resultA.Result, resultB.Result }, resultB.RemainingInput }
  }
}
```

27

a.Convert (f)

- Convert the result of a parser.

```
func (a Parser) Convert (f func (interface {}) interface {}) Parser {
  return func (input ParserInput) ParserResult {

    var result = a (input)

    if result.Result == nil {
      return result
    }

    result.Result = f (result.Result)
    return result
  }
}
```

28

An Example Parser

```
var ParseDigit Parser = Expect ('0') .OrElse (Expect ('1')).OrElse (Expect ('2')).
                OrElse (Expect ('3')).OrElse (Expect ('4')).OrElse (Expect ('5')).
                OrElse (Expect ('6')).OrElse (Expect ('7')).OrElse (Expect ('8')).
                OrElse (Expect ('9'))

var ParseNumber Parser = ParseDigit.OnceOrMore ()
```

29

A Recursive Example

We rewrite the number parser recursively.

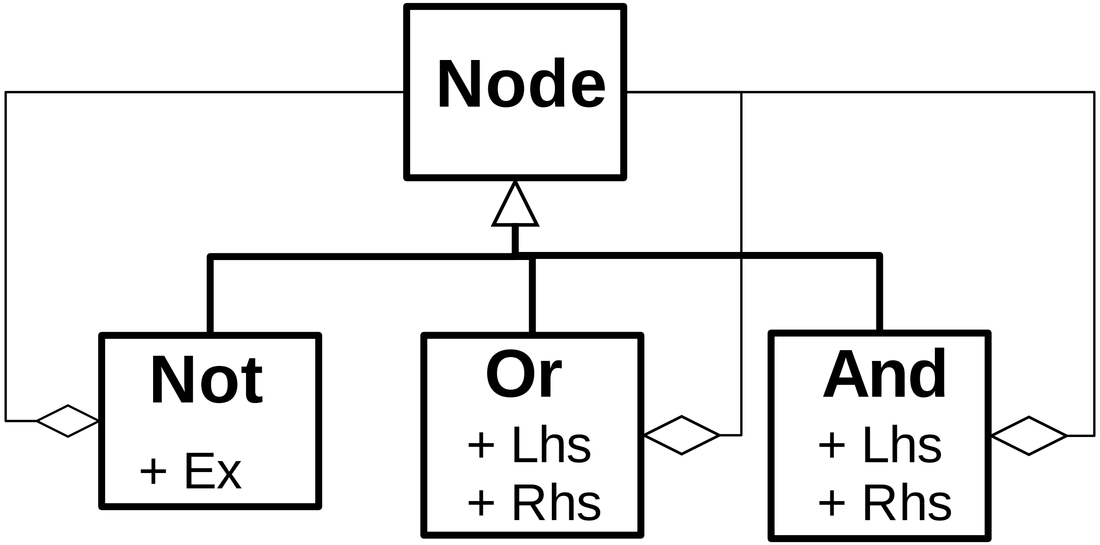
Number := Digit Number | Digit

```
func ParseNumber (input ParserInput) ParserResult {

  return ParseDigit.AndThen (ParseNumber).
         OrElse (ParseDigit)

}
```

30

Abstract Syntax Trees

- Abstract syntax trees are algebraic data types
- Example: Boolean Formulas (https://github.com/jweigend/concepts-of-programming-languages/blob/master/oop/ast/ast.go)



31

Exercise

- Implement the following grammar using parser combinators
- Use the type Node in "github.com/jweigend/concepts-of-programming-languages/oop/ast" as a syntax tree
- Use the method `Convert` to convert all the pairs and lists into values of type Node
- Print the trees to check your parser

```
Atom       = VariableName
           | "(" ^ Expression ^ ")"
Not        = "!"* ^ Atom
And        = Not ^ ("&" ^ Not)*
Or         = And ^ ("|" ^ And)*
Expression = Or
```

32

Thank you

Armin Heller on behalf of Johannes Weigend (QAware GmbH)
University of Applied Sciences Rosenheim
johannes.weigend@qaware.de, armin.heller@qaware.de (mailto:johannes.weigend@qaware.de,%20armin.heller@qaware.de)
http://www.qaware.de (http://www.qaware.de)
@johannesweigend (http://twitter.com/johannesweigend)