Regular expression examples

- **Identifiers:**
- **Integer literals**
- **The keyword `if`**
- **The left-shift operator**
- **Floating-point literals**

Ocamllex example 1

- **In this and following examples, the trailer is omitted; it will contain definitions like the ones on the previous slide.**
- **What does this do?**

```
rule main = parse
   ['0'-'9']+                    { "Int" }
 | ['0'-'9']+'.'['0'-'9']+       { "Float" }
 | ['a'-'z']+                    { "String" }
```

Ocamllex examples 2 and 3

```
let digit = ['0'-'9']
rule main = parse
   digit+                { "Int" }
 | digit+'.'digit+       { "Float" }
 | ['a'-'z']+            { "String" }
```

```
{   type token = Int | Float | Ident    }
rule main = parse
   ['0'-'9']+                    { Int }
 | ['0'-'9']+'.'['0'-'9']+       { Float }
 | ['a'-'z']+                    { Ident }
```

Ocamllex examples 4 and 5

```
{  type token = Int | Float | Ident  }
rule main = parse
   ['0'-'9']+                    { Int }
 | ['0'-'9']+'.'['0'-'9']+       { Float }
 | ['a'-'z']+                    { Ident }
 | _                             { main lexbuf }
```

```
{ type token = Int of int | Float of float | Ident of string }
rule main = parse
   ['0'-'9']+ as x                    { Int (int_of_string x) }
 | ['0'-'9']+'.'['0'-'9']+ as x       { Float (float_of_string x) }
 | ['a'-'z']+ as id                   { Ident id }
 | _                                  { main lexbuf }
```

Ocamllex example 6

```
{ type token = Int of int | Float of float | Ident of string | EOF }

rule main = parse
   ['0'-'9']+ as x                    { Int (int_of_string x) }
 | ['0'-'9']+'.'['0'-'9']+ as x       { Float (float_of_string x) }
 | ['a'-'z']+ as id                   { Ident id }
 | _                                  { main lexbuf }
 | eof                                { EOF }
```

Difficult cases...

- **Write regular expressions for various kinds of comments:**
  - **C++-style (`//...`)**
  - **C-style (`/*...  */`, no nesting)**
  - **OCaml-style (`(*...  *)`, with nesting)**

Handling C-style comments

```
let open_comment = "/*"
let close_comment = "*/"
rule main = parse
     digits '.' digits as f   { Float (float_of_string f) }
   | digits as n              { Int (int_of_string n) }
   | letters as s             { Ident s }
   | open_comment             { comment  lexbuf }
   | eof                      { EOF }
   | _                        { main lexbuf  }
and comment = parse
     close_comment            { main lexbuf }
   | _                        { comment lexbuf }
```

Handling OCaml-style comments

- **Every lexing function has an argument named** `lexbuf` **of type** `Lexing.lexbuf`**, but you can add additional arguments. This allows you to handle nested comments (even though they are not finite-state).**

```
rule main = parse
    | open_comment          { comment 1 lexbuf}
    | eof                   { [] }
    | _                     { main lexbuf }
and comment  depth = parse
      open_comment          { comment (depth+1) lexbuf }
    | close_comment         { if depth = 1
                              then main lexbuf
                              else comment (depth - 1) lexbuf }
    | _                     { comment depth lexbuf }
```