Lexical Analysis
Programming Languages

Sujit Kumar Chakrabarti

IIITB

Manual Implementation of Lexical Analysis

Section Goals

1. How FSAs come together to build a complete lexical analyser
2. Functional programming
3. Software design: modularity, testability, scalability

Manual Implementation of Lexical Analysis

Functional Programming – Paradigm shift in Programming

1. Takes work to get the program to compile
2. Requires very little time to test and debug
3. Terse, expressive and maintenable
4. Type system does the heavylifting
5. Useful in system programming

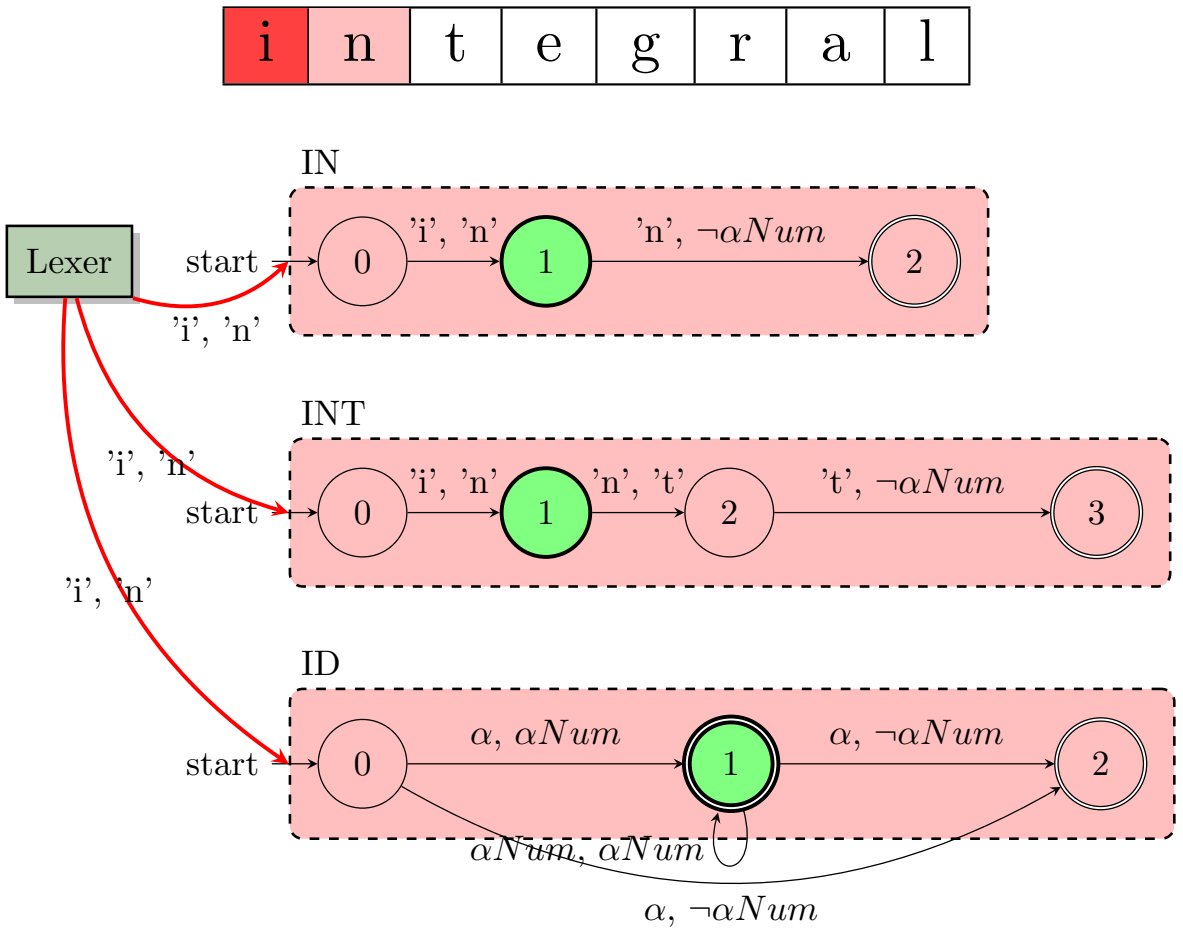
Manual Implementation of Lexical Analysis

The Lexer Algorithm

```
procedure LEXER(buffer)
    C ← all scanners
    S ← {}
    while there is input left do
        c, la ← NEXTCHAR(buffer)
        S ← all s ∈ C terminating in success
        C ← all s ∈ C making a move
        if C = {} then
            if S ≠ {} then
                choose randomly from S and return corresponding token
            else
                raise LexicalError
            end if
        end if
    end while
end procedure
```

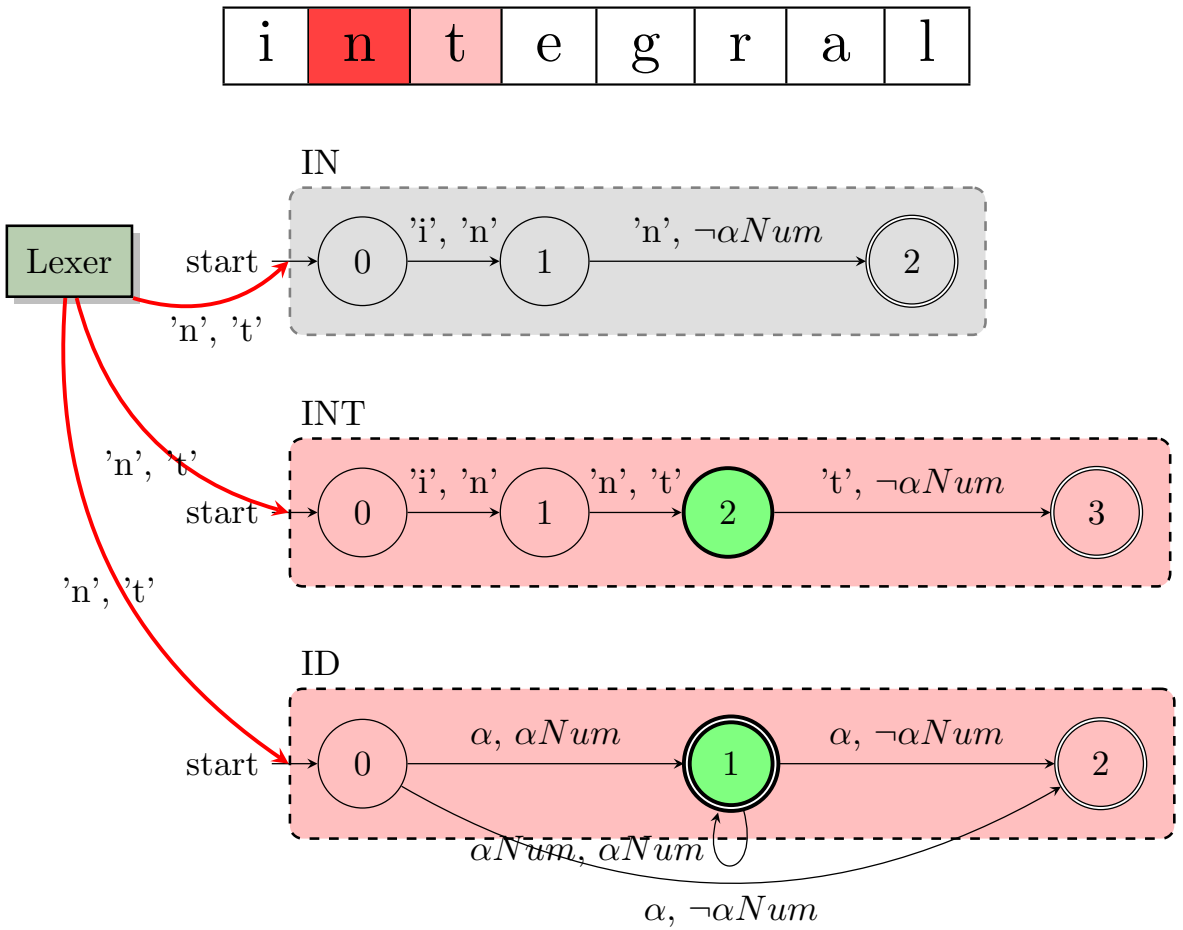
Implementation of Lexical Analysis

Manual Implementation of Lexical Analysers – Example



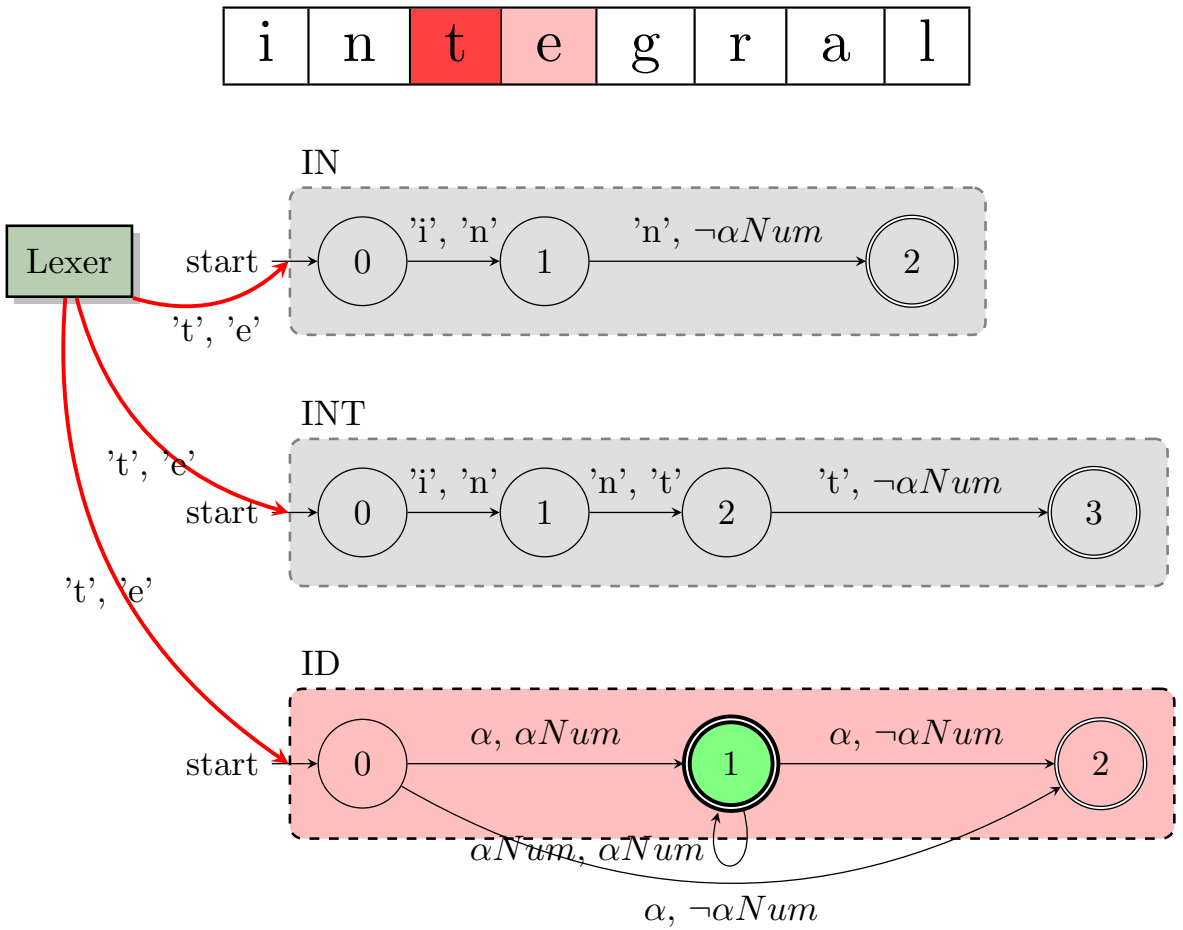
Implementation of Lexical Analysis

Manual Implementation of Lexical Analysers – Example



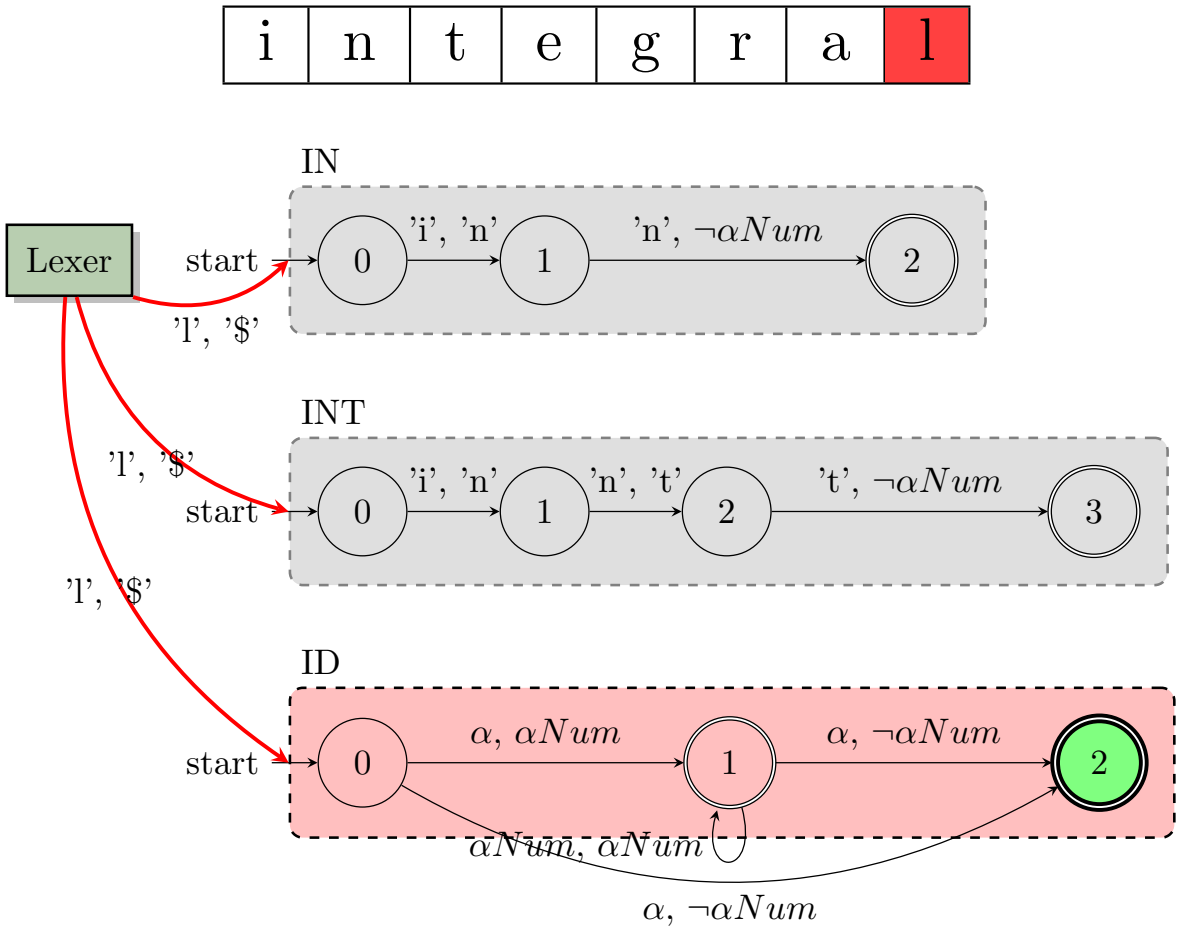
Implementation of Lexical Analysis

Manual Implementation of Lexical Analysers – Example



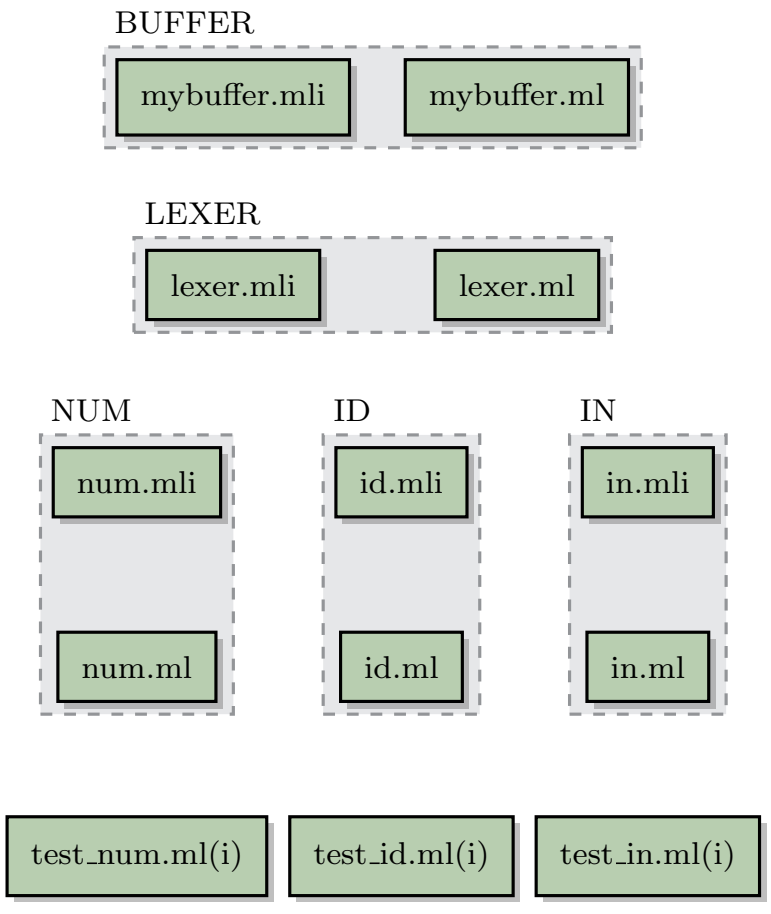
Implementation of Lexical Analysis

Manual Implementation of Lexical Analysers – Example



Implementation of Lexical Analysis

Manual Implementation of Lexical Analysers – Example



Manual Implementation of Lexical Analysis

Lazy implementation

1. FSA: Pair of *current state* and *list of accepting states*

```
let id () : State.state *
  ((char -> char option -> State.state) list) =
```

2. State $\mapsto$ Mutually recursive higher order function

```
let rec one (c : char) (lookahead : char option)
  : State.state =
  ...
  ...
and two  (c : char) (lookahead : char option)
  : State.state =
```

3. Takes two characters as input: *current character* and *lookahead* ($\Sigma' = \Sigma \times \Sigma$)

Manual Implementation of Lexical Analysis

State

state.mli

```
type state =
    Terminate of bool
  | State of (char -> char option -> state)
```

1. Models the return type of a state function
2. *Terminate in failure (`false`) or success (`true`)*
   OR
   Next state
3. Caller can decide if/when/how to explore the next state

Manual Implementation of Lexical Analysis

Buffer

mybuffer.mli

```
exception End_of_buffer
val from_string : bytes -> (unit -> (char * char option))
val from_file : bytes -> (unit -> char -> char option)
```

mybuffer.ml

```
let from_file fname =
  ...
  let buffer () : (char * char option) =
    ...

  in
  buffer
```

1. Represents the input source
2. Stateful
3. Caller can decide if/when/how to explore the next state

Manual Implementation of Lexical Analysis

Source code

1. Buffer
2. FSAs (NUM, ID, IN)
3. Lexer
4. Makefile