Compilers

INF-400

Burak Arslan
ext-inf400@burakarslan.com

Galatasaray Üniversitesi

Lecture II
2023-10-12

Course website

burakarslan.com/inf400

Lexing

Lexing;

1. is the first step towards the **definition** of a language
2. is a left-to-right scan operation with some **lookahead**
3. defines the set of token classes and the set of character sequences that belong to these classes
4. needs a set of characters to operate on called the **alphabet**

Lexing

1. Σ = English characters
   Language = English sentences
2. Not every string of English characters is an English sentence

- Σ = ASCII
  Language = C programs
- Note: ASCII character set is different from English character set

Lexing

```
import io

func main() -> int32 {
    io.print("Hello world!\n");
    return 0;
}
```

Lexing

```
KW_IMPORT IDENTIFIER(io) OP_NEWLINE
KW_FUNC IDENTIFIER(main) OP_LPAREN OP_RPAREN OP_RETURNS
                                    IDENTIFIER(int32) OP_LBRACE OP_NEWLINE
IDENTIFIER(io) OP_DOT IDENTIFIER(print) OP_LPAREN
               L_STRING("Hello world!\n") OP_RPAREN OP_SCOLON OP_NEWLINE
IDENTIFIER(return) L_INTEGER(0, 10) OP_SCOLON OP_NEWLINE
OP_RBRACE OP_NEWLINE
```

* Indented lines are continuations of the previous ones. There is actually only 5 lines in the above output

Lexing

Tokens

Language definitions start with tokens types:

Type	Examples
IDENTIFIERs	`foo bar main`
INTEGERs	`123 -456 42 0xFF`
FLOATs	`12.34 +56.78`
KEYWORDs	`if while import`
OPERATORs	`, . >= ==`

Lexing

Tokens

A token type specifies a **set of acceptable strings**

Lexing

Tokens

A token type specifies a **set of acceptable strings**

Type	**Examples**
INTEGERs	123 -456 42 0xFF

... but INTEGER set is infinite (?)

Lexing

Tokens

So we turn to regular languages. More specifically;

1. **Regular Expressions**: for defining the sets
2. **Deterministic Finite** (State) **Automata (DFA)**: for testing set membership

Lexing

Regular Expressions

Some examples:

Type	**Regex**
IF	
±INTEGER (dec)	
+INTEGER (hex)	
IDENTIFIER	
KEYWORD	

Lexing

Regular Expressions

Some examples:

Type	**Regex**
IF	`if`
±INTEGER (dec)	
+INTEGER (hex)	
IDENTIFIER	
KEYWORD	

Lexing

Regular Expressions

Some examples:

Type	**Regex**
IF	`if`
±INTEGER (dec)	[0-9]+
+INTEGER (hex)	
IDENTIFIER	
KEYWORD	

Lexing

Regular Expressions

Some examples:

Type	**Regex**
IF	`if`
±INTEGER (dec)	`-?[0-9]+`
+INTEGER (hex)	
IDENTIFIER	
KEYWORD	

Lexing

Regular Expressions

Some examples:

Type	**Regex**
IF	`if`
±INTEGER (dec)	`(+|-)?[0-9]+`
+INTEGER (hex)	
IDENTIFIER	
KEYWORD	

Lexing

Regular Expressions

Some examples:

Type	**Regex**
IF	`if`
±INTEGER (dec)	`(+|-)?[0-9]+`
+INTEGER (hex)	`0x[0-9a-fA-F]+`
IDENTIFIER	
KEYWORD	

Lexing

Regular Expressions

Some examples:

Type	**Regex**
IF	`if`
±INTEGER (dec)	`(+|-)?[0-9]+`
+INTEGER (hex)	`0x[0-9a-fA-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	

Lexing

Regular Expressions

Some examples:

Type	**Regex**
IF	`if`
±INTEGER (dec)	`(+|-)?[0-9]+`
+INTEGER (hex)	`0x[0-9a-fA-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

Let's classify the following token: **if**

Type	**Regex**
IF	`if`
±INTEGER (dec)	`(+|-)?[0-9]+`
+INTEGER (hex)	`0x[0-9a-f-A-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

Let's classify the following token: **if** $\Rightarrow$ `IF`

Type	**Regex**
IF	`if`
$\pm$INTEGER (dec)	`(+|-)?[0-9]+`
$+$INTEGER (hex)	`0x[0-9a-f-A-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

Let's classify the following token: **1234**

Type	**Regex**
IF	`if`
±INTEGER (dec)	`(+|-)?[0-9]+`
+INTEGER (hex)	`0x[0-9a-f-A-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

Let's classify the following token: **1234** $\Rightarrow$ `INTEGER`

Type	**Regex**
IF	`if`
$\pm$INTEGER (dec)	`(+|-)?[0-9]+`
$+$INTEGER (hex)	`0x[0-9a-f-A-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

Let's classify the following token: **1234A**

Type	**Regex**
IF	`if`
±INTEGER (dec)	`(+|-)?[0-9]+`
+INTEGER (hex)	`0x[0-9a-f-A-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

Let's classify the following token: **1234A** $\Rightarrow$ `REJECT`

Type	**Regex**
IF	`if`
$\pm$INTEGER (dec)	`(+|-)?[0-9]+`
$+$INTEGER (hex)	`0x[0-9a-f-A-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

Let's classify the following token: **import**

Type	**Regex**
IF	`if`
±INTEGER (dec)	`(+|-)?[0-9]+`
+INTEGER (hex)	`0x[0-9a-f-A-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

Let's classify the following token: **import** $\Rightarrow$ KEYWORD

Type	**Regex**
IF	`if`
$\pm$INTEGER (dec)	`(+|-)?[0-9]+`
$+$INTEGER (hex)	`0x[0-9a-f-A-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

Let's classify the following token: **importer**

Type	**Regex**
IF	`if`
±INTEGER (dec)	`(+|-)?[0-9]+`
+INTEGER (hex)	`0x[0-9a-f-A-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

Let's classify the following token: **importer** $\Rightarrow$ IDENTIFIER

Type	**Regex**
IF	`if`
$\pm$INTEGER (dec)	`(+|-)?[0-9]+`
$+$INTEGER (hex)	`0x[0-9a-f-A-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

Let's classify the following token: **importer** $\Rightarrow$ IDENTIFIER

???

Type	Regex
IF	`if`
$\pm$INTEGER (dec)	`(+|-)?[0-9]+`
$+$INTEGER (hex)	`0x[0-9a-f-A-F]+`
IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
KEYWORD	`(if|while|import|func)`

Lexing

Regular Expressions

The *lookahead* problem

1. `import` ⇒`KW_IMPORT`
2. `importer` ⇒`IDEN(importer)`

Lexing

Regular Expressions

Ambiguities in grammars are practically impossible to avoid.
C++ Examples:

- `A < B > c;`
  - if A and B are types, this is a variable definition;
  - if A and B are variables, this is a (pointless) comparison;
- `cin >> var;`
  - `operator>>`
- `vector<unique_ptr<string>> ;`
  - Right angle bracket

In the face of ambiguities;

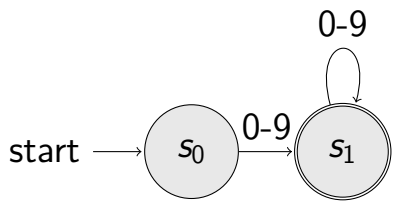
Longest match is preferred

Order of rules matter

Lexing

Deterministic Finite State Automata

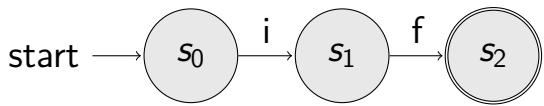
DFA that decides whether an input matches regexp: `[0-9]+`



Lexing

Deterministic Finite State Automata

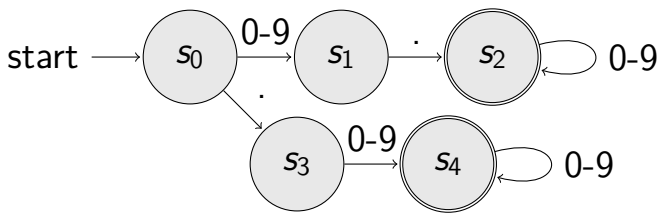
DFA that decides whether an input matches regexp: `if`



Lexing

Deterministic Finite State Automata

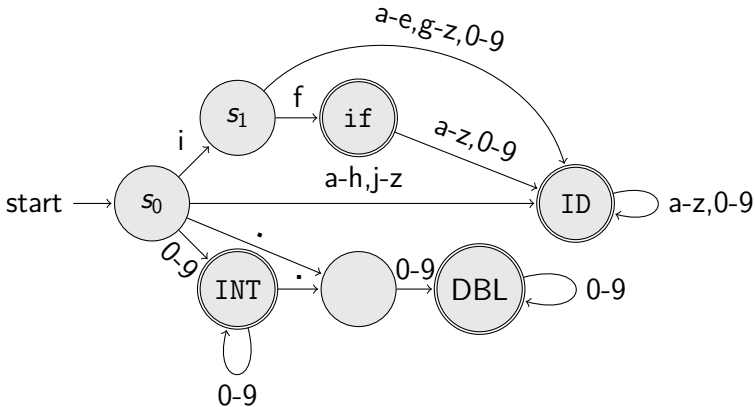
DFA ... regexp: . . . ?



Lexing

Deterministic Finite State Automata

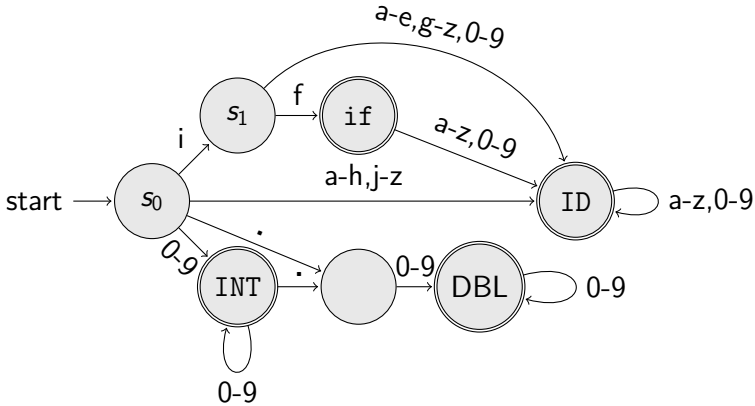
The combined DFA becomes the **lexical analyzer**



Lexing

Deterministic Finite State Automata

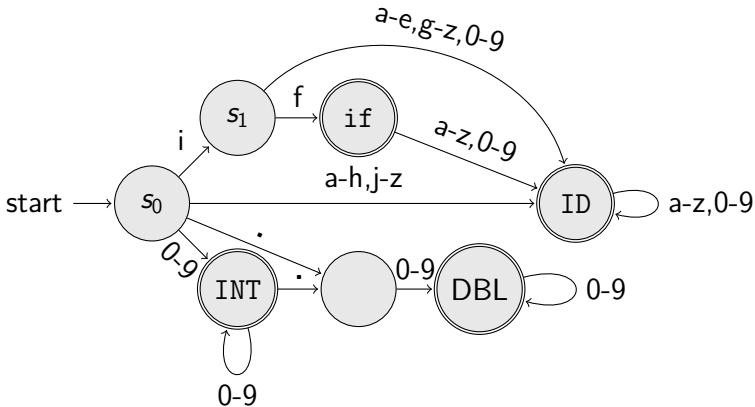
The **lexer generator**'s job is to combine all regexp to a coherent **DFA**



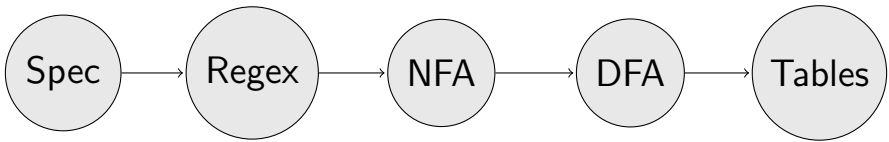
Lexing

Deterministic Finite State Automata

The lexgen is used to assign **callbacks** to accepting states



Implementing Lexers



Implementing Lexers

Regular Expressions

Remember: Regular expressions specify sets of strings.

$\forall A, B \in \text{Ⓢ}$, sets of strings over alphabet Σ;

- Neutral: $\{\texttt{""}\} \Longrightarrow \varepsilon$[1] $\neq \varnothing$
- Union: $A \cup B \Longrightarrow (A|B)$
- Concatenation: $\{s_1 s_2 \mid s_1 \in A \ \wedge \ s_2 \in B\} \Longrightarrow AB$
- Range: $\{\texttt{"a"}, \texttt{"b"}, \ldots, \texttt{"z"}\} \Longrightarrow [\texttt{a} - \texttt{z}]$
- Range Excl.: $\text{Ⓢ} \setminus \{\texttt{"a"}, \texttt{"b"}, \ldots, \texttt{"z"}\} \Longrightarrow [\texttt{\^{}a-z}]$

[1] singleton with an empty string

Implementing Lexers

Regular Expressions

Repetitions:
Let $A \in \mathbb{S}$, sets of strings over alphabet Σ, $A^n = \underbrace{AA \ldots A}_{n}$

- Optional: $A + \varepsilon \Longrightarrow A?$
- Zero or more: $\bigcup_{i \geq 0} A^i \Longrightarrow A*$
- One or more: $\bigcup_{i > 0} A^i \Longrightarrow A+$
- Explicit:
  - $\bigcup_{i \geq n} A^i \Longrightarrow A\{n\}$
  - $\bigcup_{i \geq n, i \leq m} A^i \Longrightarrow A\{n, m\}$ where $n \leq m$

Implementing Lexers

Regular expressions
are implemented using
Finite State Automata

- A finite automaton consists of
- An input alphabet Σ
- A set of states S
- A start state n
- A set of accepting states $F \subseteq S$
- A set of transitions state $\rightarrow$input state

Implementing Lexers

Finite State Automata

Two types:

- **DFA**: Deterministic Finite Automata
- **NFA**: Nondeterministic Finite Automata

Implementing Lexers

Finite State Automata

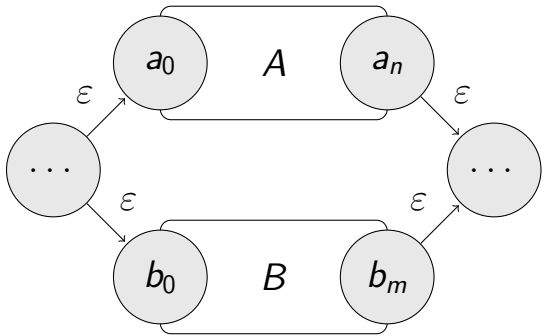
Two types:

- **DFA**:
  - No more than one move per input
  - ε moves are forbidden
- **NFA**:
  - Zero or more moves per input
  - ε moves are allowed

Implementing Lexers

Finite State Automata

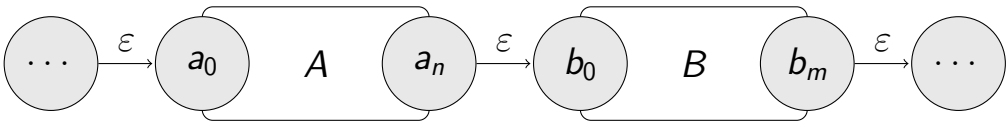
Regular expressions have direct NFA representations.
Eg. `(A|B)`:



Implementing Lexers

Finite State Automata

Regular expressions have direct NFA representations.
Eg. `AB`:



Implementing Lexers

Finite State Automata

NFA and DFA are equivalent and recognize both **regular languages**.

DFAs are faster to execute

Implementing Lexers

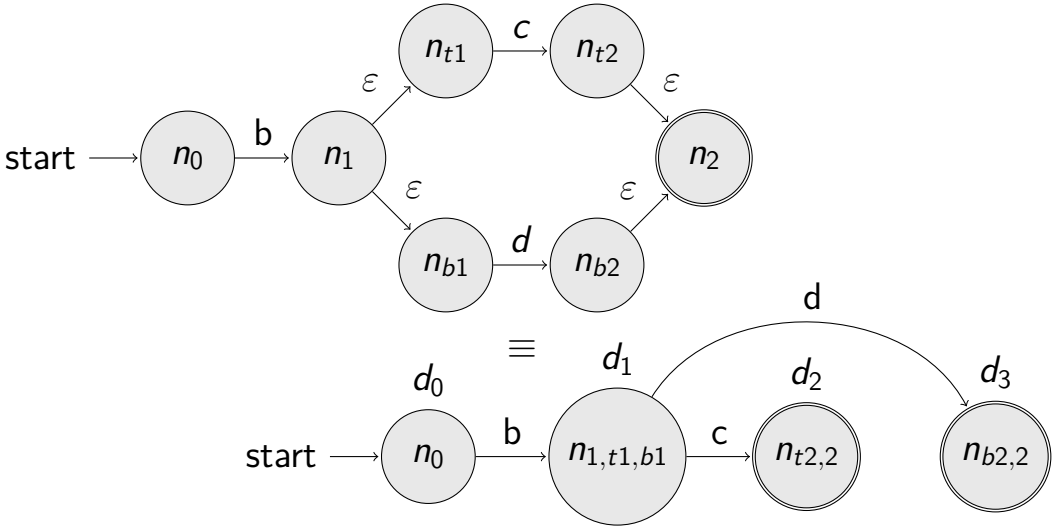
Finite State Automata

Conversion algorithm: **Simulation** / **Tracing** (recording execution)

- Start state of the DFA = States reachable from the start state of the NFA through ε input
- Let $n, n', n'', \ldots$ states from NFA,
- Let $d, d', d'', \ldots$ states from DFA,
- Add a new state $d \xrightarrow[a]{} d'$ if and only if n' is reachable from n, including ε input

Implementing Lexers

Finite State Automata



Implementing Lexers

Implementation

We use tables / grids / 2D arrays:

	input		
	b	c	d
d_0	d_1		
d_1		d_2	d_3
d_2			
d_3			

Implementing Lexers

Implementation

All in all, lexer generators' job boil down to:

- Unify all regular expressions into a single NFA
- Perform NFA $\Rightarrow$ DFA conversion
- Create DFA grid
- Execute DFA