2 Lexical Analysis

( **2. Lexical Analysis, p. 16**)

- **Lexical Analysis** is the first of three compiler phases that **analyze** the source program. (Once analysis is complete, the compiler starts to **synthesize** the target program.)

- Lexical analysis turns the **stream of characters** that comprise the input source program into a **streams of identifiers, keywords, constants, and punctuation marks**, the so-called **lexical tokens**.

2.1 Lexical Tokens

- A **lexical token** (short: **token**) is a sequence of characters that appears as a unit (**terminal**) in the source language grammar.

- Typical token types to be found in source language grammars:

Token type	Examples
ID	`foo  _bar  n14`
NUM	`73  042  1000`
STR	`"foo"  ""  "bar\n"`
REAL	`66.1  .5  10.  5.5e-10`
IF	`if`
FUNCTION	`function`
COMMA	`,`
NOTEQ	`!=`
LPAREN	`)`
RPAREN	`(`

  - Tokens `IF`, `FUNCTION` represent **reserved words** (**keywords**) of the source language. Since keywords and identifiers look similar, most source languages disallow identifiers named like reserved words.
  - **White space** (space, tab, newline, formfeed) and **comments** do not contribute to the meaning of the source program and are discarded by the lexcial analysis.

- Lexical analysis translates the input **character stream** into a **stream of tokens**.

  – **Example:**

    Input character stream (length 104 characters):

```
let function match0 (string s) : int =  /* find a zero */
   if (s = "0.0") then 0 else -1
```

    Output token stream after lexical analysis (length 21 tokens):

```
LET FUNCTION ID(match0) LPAREN STRING ID(s) RPAREN COLON INT
EQUALS IF LPRAREN ID(s) EQUALS STR(0.0) RPAREN THEN NUM(0)
ELSE MINUS NUM(1)
```

- **N.B.**: Some token types carry a **semantic value** (e.g., `NUM(0)`) which subsequent phases will need to actually compile the program.

  Token type `IF` does not need a semantic value ("*any* `if` *is like any other*").

- Sometimes, the valid character sequences associated with a token type are easily characterized, e.g., by a simple string of characters:

  Token type `IF` $\equiv$ `if`

  ("`i` immediately followed by `f`").

- Sometimes, this characterization is considerably more complex:

  Token type `ID` $\equiv$

  > "Valid identifiers consist of a **sequence** of **letters** (`a-z`, `A-Z`), **or digits** (`0-9`), **or underscores** (_). The **first** character must be a letter **or** an underscore. The **length** is not restricted but it must be **equal to or larger than 1**."

- We need a concise way to describe such characterizations of character sequences (we could always program a token recognizer by hand, but this will be complex and error-prone).

  (**N.B.**: the number of legal Tiger `ID` tokens is *infinite* as is the number of `STR`, `NUM`, ... tokens.)

2.2 Regular Expressions

- **Regular expressions** (REs) comprise a formalism in which we can concisely describe (even possibly inifinite) sets of characters sequences.

  [REs can describe sets of sequences of any symbol type, but we will exclusively work with characters in some standard character encoding, e.g., ASCII or Unicode. The set of all available characters is also called the **alphabet**.]

- Each RE M describes a certain set of character sequences. This set is also called the **language** $L(M)$ recognized by M.

- RE are *formal* description of character sequences which enables us to *automatically derive* token recognizer programs (the so-called **lexers**) from a given RE.

- Any RE is built using the following RE **constructors** (let M, N denote arbitrary REs):

  ① $\mathtt{a}$ (any single character)
  $L(\mathtt{a}) = \{\texttt{"a"}\}$.

  ② ε (epsilon)
  $L(\varepsilon) = \{\texttt{""}\}$.

  ③ $M|N$ (alternation)
  A character sequence is in $L(M|N)$ if it is in $L(M)$ or $L(N)$, thus $L(M|N) = L(M) \cup L(N)$. **Example:** $L(\mathtt{a}|\mathtt{b}) = \{\texttt{"a"}, \texttt{"b"}\}$.

  ④ $M \cdot N$ (concatenation, also written as MN)
  $L(M \cdot N)$ contains all possible combinations of character sequences in $L(M)$ followed by character sequences in $L(N)$. **Example:** $L((\mathtt{a}|\mathtt{b}) \cdot \mathtt{a}) = \{\texttt{"aa"}, \texttt{"ab"}\}$.

  ⑤ M^* (repetition, *Kleene closure*)
  $L(M^*)$ contains the concatenations of zero or more character sequences in $L(M)$, i.e., $L(M^*) = L(\varepsilon) \cup L(M) \cup L(MM) \cup L(MMM) \cup \cdots$.
  Example: $L((b \cdot a)^*) = \{\texttt{""}, \texttt{"ba"}, \texttt{"baba"}, \texttt{"bababa"}, \ldots\}$.

- Using these basic forms of REs we can construct more complex REs:
  - **Examples:**
    - Binary numbers that are multiples of two:

$$(0|1)^* \cdot 0$$

    - Decimal numbers, possibly with a leading sign:

$$(+|-|\varepsilon) \cdot (0|1|2|3|4|5|6|7|8|9) \cdot (0|1|2|3|4|5|6|7|8|9)^*$$

- A number of RE **abbreviations** are in common use (these are nothing but RE "macros" adding *no* new functionality):

RE abbreviation	**Expansion**	**Remark**
`[abcd]`	(a|b|c|d)	
`[a-d]`	(a|b|c|d)	symbol set must be ordered
$M?$	$(M \mid \varepsilon)$	optional M
M^+	$(M \cdot M^*)$	one or more M

– **Example:**

  Decimal numbers, possibly with a leading sign:

$$(+|-)? \cdot \texttt{[0-9]}^+$$

- Summary of RE forms:

RE	**Meaning**
a	the one character sequence a itself
ε	the empty character sequence
$M\|N$	alternation, choose between M or N
$M \cdot N$	concatenation (also MN), M followed by N
M^*	repetition, M zero or more times
M^+	repetition, M one or more times
$M?$	optional, zero or one M
`[a-zA-Z0-9]`	character set alternation
.	any single character except newline (`"\n"`)
`"a.+*"`	quotation, the literal character sequence itself

- We can now use these REs to specify how the lexical analysis phase shall behave.
  - Simply specify a list of REs $M_1, M_2, \ldots M_k$ and k associated **actions** to execute, when the lexer has seen a character sequence that belongs to the language of some RE M_i:
  - **Example:** lexer specification for a simple language with numeric constants, identifiers, and "`-- foo`" style comments:

Lexer specification

```
1   if                                      { return IF;          }
2   [a-z][a-z0-9]*                          { return ID;          }
3   [0-9]+                                  { return NUM;         }
4   ([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)     { return REAL;        }
5   ("--"[a-z]*"\n")|(" "|"\n"|"\t")+       { /* noop, resume */ }
6   .                                       { error ();           }
```

  - **Remarks:**
    - Empty actions (like `/* noop */`) instruct the lexical analyzer to throw the matched characters away and **resume** lexical analysis (do not return from lexer).
    - The last entry catches all characters that have not been caught by any previous entry (some illegal character spotted in the input).

Lexer specification

```
1   if                                      { return IF;          }
2   [a-z][a-z0-9]*                          { return ID;          }
3   [0-9]+                                  { return NUM;         }
4   ([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)     { return REAL;        }
5   ("--"[a-z]*"\n")|(" "|"\n"|"\t")+       { /* noop, resume */ }
6   .                                       { error ();           }
```

- Note that the above specification is ambiguous as it stands:

① Is `"foo42"` reported as two tokens `ID(foo) NUM(42)` or one token `ID(foo42)` ?

> **Rule:** The *longest prefix* of the input we can recognize "wins".
> $\Rightarrow$ `"foo42"` is reported as `ID(foo42)`.

② Is `"if"` reported as the keyword `IF` or as the identifier `ID(if)` ?

> **Rule:** Whenever two REs M_i, M_j with $i < j$ recognize the same input prefix, M_i "wins" (the order in which we write down the lexer specification matters).
> $\Rightarrow$ `"if"` is reported as keyword `IF`.

2.3 Finite Automata

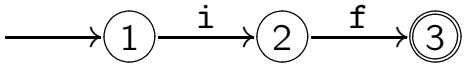
- REs are convenient to write down lexer specifications but we need to transform the REs before we can (automatically) generate a program that implements lexical analysis for us.

- $\Rightarrow$ Turn REs into **finite automata** (FA)[5].

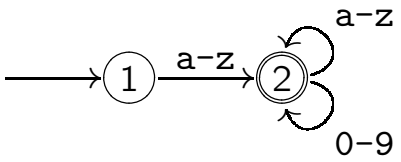
- A FA consists of
  - a **finite set of states**
    (one state is *the* **start state**, some—at least one—states may be **final states**),
  - directed **edges**, leading from state to (possibly the same) state
    (each edge is labelled by one character).

[5] Singular: **finite automaton**.

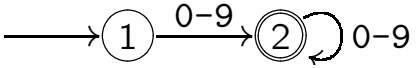
- The six FA corresponding to the six RE $M_1 \ldots M_6$ of the lexer specification shown earlier:



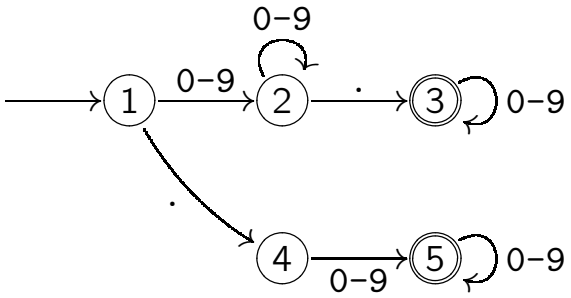
`IF`



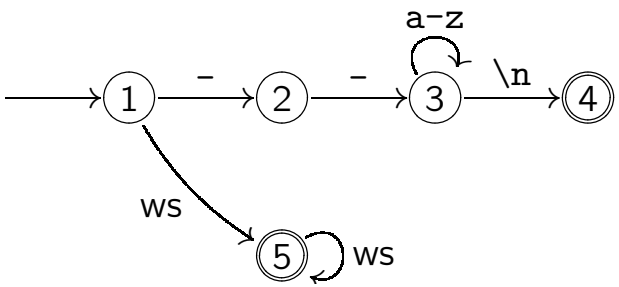
`ID`



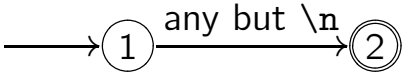
`NUM`



`REAL`



`white space/comment`



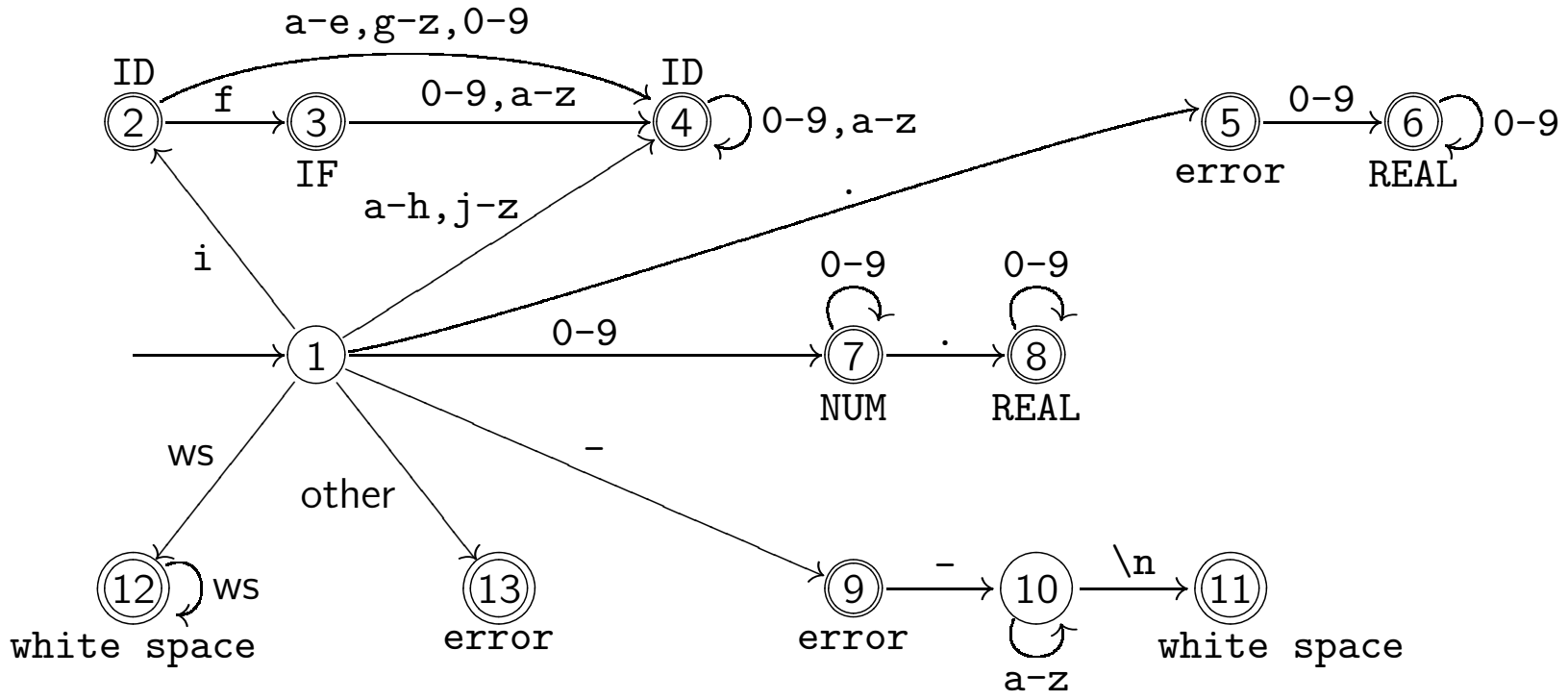
`error`

- **FA operation:**

① Start in start state.
② Read next character c, follow edge labelled c (if no such edge exists, halt the FA).
③ If we are in a final state, the read character sequence so far is **accepted** by the FA, otherwise the FA **rejects** the input.

- The set of character sequences accepted by a FA A is called the **language** $L(A)$ recognized by A.

- To build a single lexical analyzer from the six automata just shown, combine them into a single FA:



  - **NB:** each final state must be associated with the token type recognized in that state.

- How could we simulate the operation of such a FA with a computer program?
  - Rather simple, since the FA is **deterministic** (**deterministic FA**, **DFA**): *no* two edges leaving from one state are labelled with the *same* character.
  - Encode DFA as **transition matrix** *trans*[*s*][*c*]:

state *s* \ character *c*	...	0	1	2	...	-	...	e	f	g	h	i	j	...
1		7	7	7		9		4	4	4	4	2	4	
2		4	4	4		↯		4	3	4	4	4	4	
3		4	4	4		↯		4	4	4	4	4	4	
4		4	4	4		↯		4	4	4	4	4	4	
5		6	6	6		↯		↯	↯	↯	↯	↯	↯	
6		6	6	6		↯		↯	↯	↯	↯	↯	↯	
7		7	7	7		↯		↯	↯	↯	↯	↯	↯	
8		8	8	8		↯		↯	↯	↯	↯	↯	↯	
⋮														

  - ↯ indicates that the FA will halt.

- To implement the actions associated with the finite states, additionally maintain an array *action*[*s*] that maps final states to actions:

state s	action
1	□
2	`return ID`
3	`return IF`
4	`return ID`
5	`error ()`
6	`return REAL`
7	`return NUM`
8	`return REAL`
⋮	

Remarks:

– State 1 is not a final state and is not associated with any action.

– Final states 11 and 12 would be associated with a `resume ()` action.

- Now, how do we implemement the "longest input prefix sequence wins" rule?

DFA simulator

```
 1      typedef int state;
 2      state s;                                    /* current DFA state */
 3      int   pos = 0;                              /* current position in input[] stream */
 4      state Last_Final = ↯;                       /* remember last final state reached */
 5      int   Input_Position_at_Last_Final = 0;     /* remember pos we were in */
 6
 7  lex:
 8      s = 1;                                      /* start state is 1 */
 9
10      do {
11         c = input[pos++];
12         s = trans[s][c];                         /* DFA transition */
13
14         if (action[s] != □) {                    /* if we have reached a final state ... */
15            Last_Final = s;                       /* ... remember it ... */
16            Input_Position_at_Last_Final = pos;   /* ... and the current input pos */
17         }
18      } while (s != ↯);
19
20      execute action[Last_Final];                 /* invoke lexer action */
21
22      Last_Final = ↯;                             /* prepare to lex next token */
23      pos = Input_Position_At_Last_Final;
24
25      goto lex;
```

- **Example:** trace progess of the DFA while it consumes the input `"if foo42-"` ($\bot \equiv$ `pos`, $\top \equiv$ `Input_Position_At_Last_Final`):

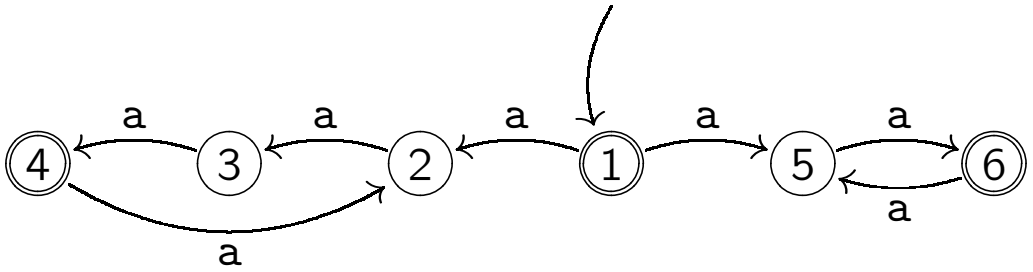
`Last_Final`	`s`	`input[]`	**Action**
↯	1	`⊤if foo42-⊥`	
2	2	`i⊤f⊥ foo42-`	
3	3	`if⊤ ⊥foo42-`	
3	↯	`if ⊤⊥foo42-`	`return IF`
↯	1	`if⊤ ⊥foo42-`	
12	12	`if ⊤f⊥oo42-`	
12	↯	`if ⊤f⊥oo42-`	`resume ()` (white space)
↯	1	`if ⊤f⊥oo42-`	
4	4	`if f⊤o⊥o42-`	
4	4	`if fo⊤o⊥42-`	
⋮	⋮	⋮	

2.4 Nondeterministic Finite Automata (NFA)

- A **nondeterminisitc finite automaton** (**NFA**) introduces two relaxations with respect to the DFA notion we have seen so far:

  ① Two or more edges leaving a state s may be labelled with the same character c (*choose* an edge: nondeterminism).

  – **Example** (this NFA accepts the empty sequence as well as a sequences whose length is a multiple of 2 *or* 3):

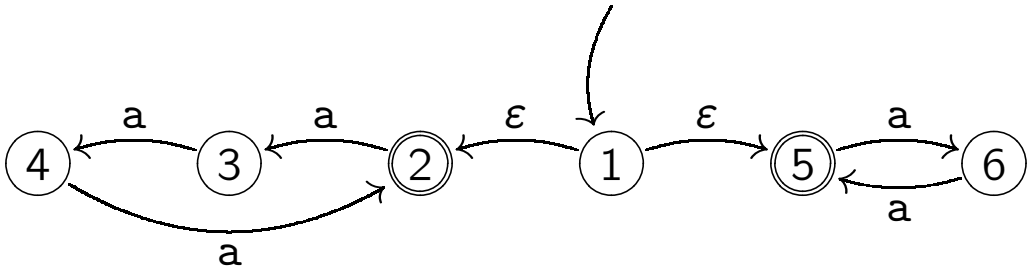


  – An NFA **accepts** a character sequence, if *any* choice of state transitions leads to some final state.

    (The NFA must "guess", and must always guess correctly.)

② Edges may be labelled with ε, we may traverse such an edge without consuming any input character at all.

– **Example** (this NFA accepts the same language like the on we've seen on the previous slide):



- The NFA idea may sound esoteric at first. NFAs are, however, most useful to automatically convert a given RE M into an **equivalent** NFA A.

- **Equivalent** NFA A?

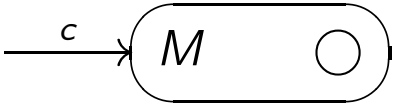
$$\text{NFA } A \text{ and RE } M \text{ equivalent} \quad \Leftrightarrow \quad L(A) = L(M) \ .$$

– **Question:** What would be the equivalent RE for the NFA shown above?

- Remember that REs are constructed from simple RE building blocks.

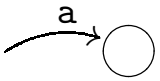
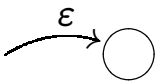
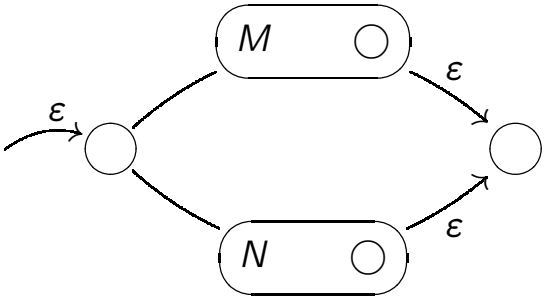
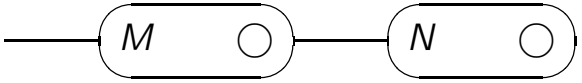
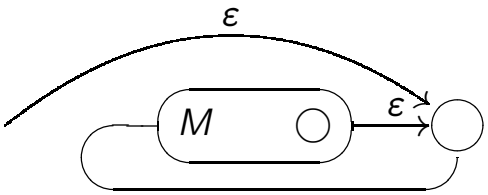
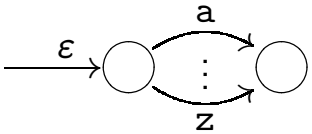
  We describe how to inductively build the NFA corresponding to a given RE by giving an equivalent NFA for each building block separately.

  – The NFA construction for each RE M (no matter how simple or complex) results in an NFA with
    - a single *tail* (start edge), and
    - a single *head* (ending state):



  – The NFA construction algorithm "pastes" the component NFAs using these tails and heads.

- **RE to NFA mapping algorithm:**

RE	$\mapsto$	NFA
a	$\mapsto$	
ε	$\mapsto$	
$M\|N$	$\mapsto$	
$M \cdot N$	$\mapsto$	
M^*	$\mapsto$	
[a–z]	$\mapsto$	

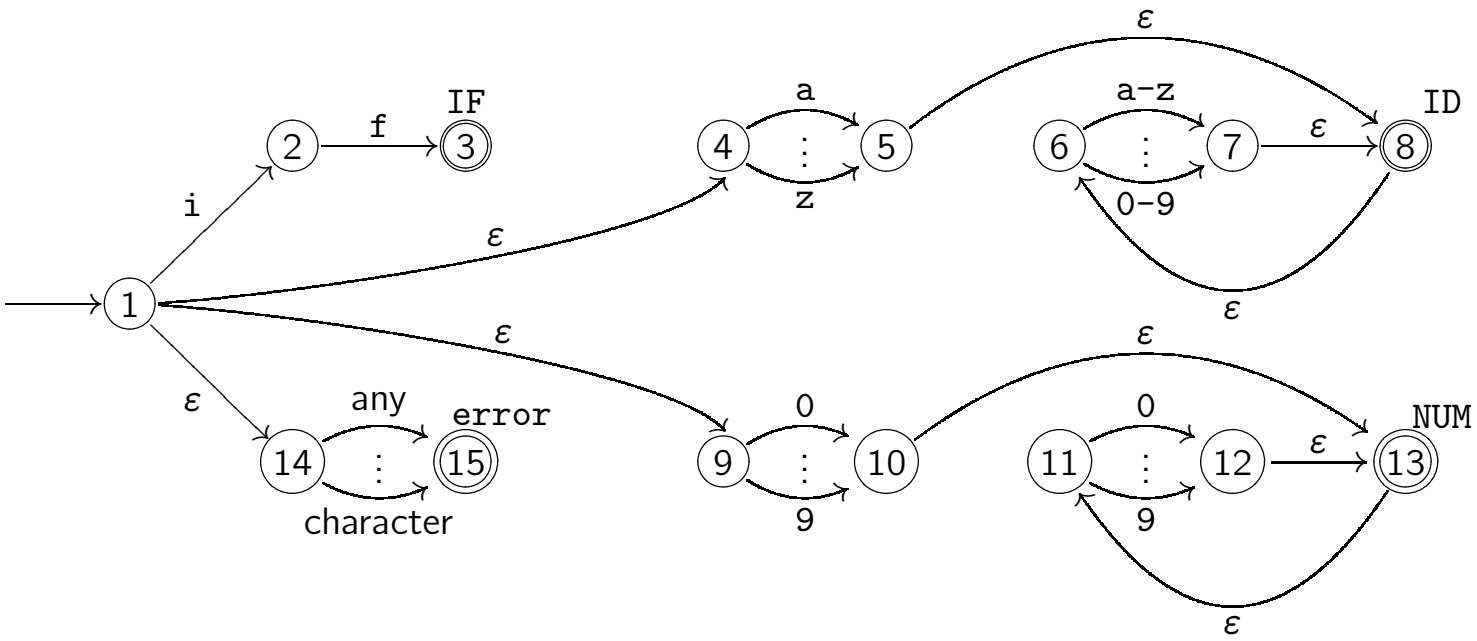
- To construct a single NFA from a given list of REs $M_1, M_2, \ldots M_k$,

  ① attach the tails (start edges) of $M_2, \ldots, M_k$ to the start state of M_1,

  ② make the start state of M_1 the overall start state,

  ③ mark the heads (ending states) of $M_1, \ldots, M_k$ to be final.

  – **Example:** NFA for the token types `ID`, `IF`, `NUM` (and the `error()` catch-all) discussed earlier:

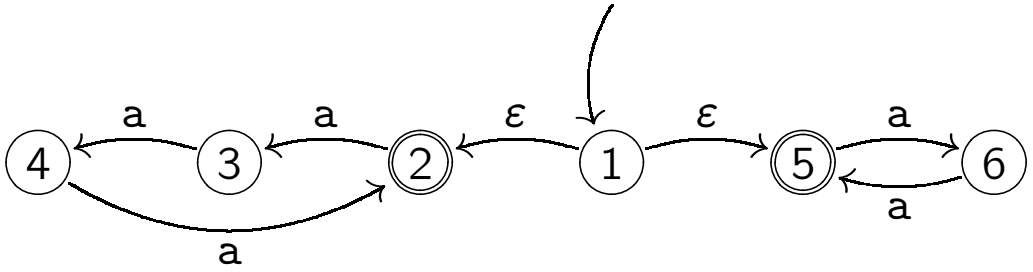


2.4.1 Converting an NFA to a DFA

- How to implement a DFA is quite obvious (see above). How shall we implement an NFA without good "guessing hardware"?

  > **Idea:** Avoid guessing, instead simply move to each possible following state "in parallel" (maintain a *set of current states* S).

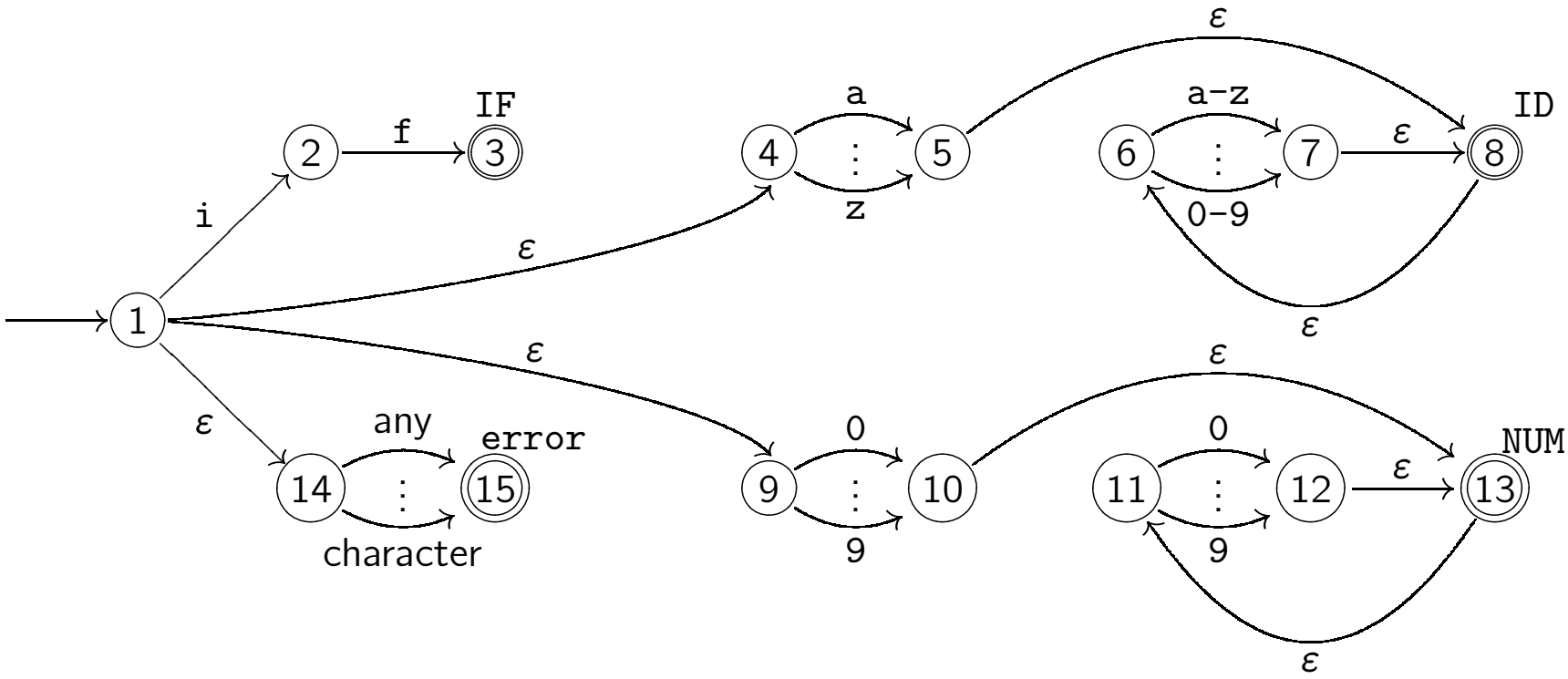
  – **Example:** simulate operation of NFA below on input `"aaa"`:



① $S = \{1\}$, follow ε-edges $\Rightarrow S = \{1, 2, 5\}$, next character: a $\Rightarrow S = \{3, 6\}$.
② $S = \{3, 6\}$, follow ε-edges $\Rightarrow S = \{3, 6\}$, next character: a $\Rightarrow S = \{4, 5\}$.
③ $S = \{4, 5\}$, follow ε-edges $\Rightarrow S = \{4, 5\}$, next character: a $\Rightarrow S = \{2, 6\}$.
④ $S = \{2, 6\}$, follow ε-edges $\Rightarrow S = \{2, 6\}$.
⑤ $2 \in S$ is final state $\Rightarrow$ **accept**.

NFA simulation: Follow ε-edges until state set S stable (ε-**closure**), then consume next character c and delete all states in S with no outgoing c-edge (eliminates "wrong guesses").

– **Example:** simulate the behaviour of the NFA below for input `"in"`:



① $S = \{1\}$, follow ε-edges ...

- More formally:
  - Let $edge(s, c)$ denote the set of all NFA states reachable from state s by using outgoing c-labelled edges.
    Example: $edge(1, \varepsilon) = \{4, 9, 14\}$ above.
  - We can compute the ε-closure of a set of states S by the following iteration:

$$
\begin{array}{l}
closure(S) \stackrel{\text{def}}{=} \\
T \leftarrow S; \\
\texttt{repeat} \\
\quad T' \leftarrow T; \\
\quad T \leftarrow T \cup \left(\bigcup_{s \in T} edge(s, \epsilon) \right); \\
\texttt{until } T = T'; \\
\texttt{return } T;
\end{array}
$$

    Example: $closure(\{1, 7\}) = \{1, 4, 6, 7, 8, 9, 14\}$.

- Now it is easy to specify the NFA behaviour:
  - Assume we are in the set of states d and we see character c. Travel from all states in d in parallel, then perform ε-closure to compute the new state set $DFAedge(d, c)$:

$$DFAedge(d, c) \;=\; closure\left(\bigcup_{s \in d} edge(s, c)\right) \quad .$$

  - To simulate an NFA with start state s_1 on the input character sequence $c_1 \cdots c_k$:

```
d ← closure({s_1});
foreach i ← 1 ... k do
    d ← DFAedge(d, c_i);
if d contains a final state then
    accept c_1 ··· c_k;
else
    reject c_1 ··· c_k;
```

  - **N.B.**: This NFA simulator operates just like a DFA (the DFA states correspond to NFA state sets), compare with slide 60, line 12.

- Computing *DFAedge*(d, c) anew for each input character read is rather costly.

- We can, however, simply compute the state sets (= the DFA states) beforehand. Effectively, this is a NFA $\rightarrow$ DFA conversion.

  N.B.: since each DFA state corresponds to a set of NFA states, the DFA size may be exponential in the NFA size.

- **NFA to DFA conversion algorithm:**
  - **Given:** NFA (start state s_1), and corresponding *DFAedge*() function.
  - **Output:** DFA transition matrix *trans*[*s*][*c*].

```
/* array states[]: remember NFA state sets we generated already */
states[0] ← ∅;
states[1] ← closure({s1});

p ← 1;
j ← 0;

while j ≤ p do
    foreach c in alphabet do
        e ← DFAedge(states[j], c);
        /* do we reach a state we generated already? */
        if e = states[i] for some i ≤ p then
            trans[j][c] ← i;
        else
            /* we have reached a new state */
            p ← p + 1;
            states[p] ← e;
            trans[j][c] ← p;
    j ← j + 1;
```

- **N.B.**: The algorithm potentially generates 2^n DFA states for a given NFA with n states (in practice, though, we find the DFA size to match the NFA size).

- Mark state d as final in DFA if at least one NFA state in $states[d]$ is final in the NFA:
  - Assign to $action[d]$ the token type associated with that final NFA state.
  - If more than one NFA state is final in $states[d]$, assign to $action[d]$ the token type that appeared *first* in the lexical analyzer specification. (This implements **rule priority**, see page 54.)

- **Example:** DFA generated from NFA for token types `ID`, `IF`, `NUM` (and `error()`):