COMPILER DESIGN

Lexical analysis

Lexical analysis

- Lexical analysis is the process of converting a sequence of **characters** into a sequence of **tokens**.

- A program or function which performs lexical analysis is called a *lexical analyzer*, *lexer* or *scanner*.
- A scanner often exists as a single function which is called by the parser, whose functionality is to extract the next token from the source code.
- The lexical specification of a programming language is defined by a set of rules which defines the scanner, which are understood by a lexical analyzer generator such as *lex* or *flex*. These are most often expressed as **regular expressions**.
- The lexical analyzer (either generated automatically by a tool like *lex*, or hand-crafted) reads the source code as a stream of characters, identifies the lexemes in the stream, categorizes them into tokens, and outputs a token stream.
- This is called "tokenizing."
- If the scanner finds an invalid token, it will report a lexical error.

Roles of the scanner

- Removal of comments
  - Comments are not part of the program’s meaning
    - Multiple-line comments?
    - Nested comments?
- Case conversion
  - Is the lexical definition case sensitive?
    - For identifiers
    - For keywords

Roles of the scanner

- Removal of white spaces
  - Blanks, tabulars, carriage returns
  - Is it possible to identify tokens in a program without spaces?
- Interpretation of compiler directives
  - **`#include`**, **`#ifdef`**, **`#ifndef`** and **`#define`** are directives to "redirect the input" of the compiler
  - May be done by a pre-compiler
- Initial creation of the symbol table
  - A symbol table entry is created when an **identifier** is encountered
  - The lexical analyzer cannot create the whole entries
  - Can convert literals to their value and assign a type
- Convert the input file to a token stream
  - Input file is a character stream
  - **Lexical specifications**: literals, operators, keywords, punctuation

Lexical specifications: tokens and lexemes

- Token: An element of the lexical definition of the language.
- Lexeme: A sequence of characters identified as a token.

Token	Lexeme
`id`	`distance,rate,time,a,x`
`relop`	`>=,<,==`
`openpar`	`(`
`if`	`if`
`then`	`then`
`assignop`	`=`
`semi`	`;`

Design of a lexical analyzer

Design of a lexical analyser

- Procedure
  1. Construct a set of regular expressions (REs) that define the form of any valid token
  2. Derive an NDFA from the REs
  3. Derive a DFA from the NDFA
  4. Translate the NDFA to a state transition table
  5. Implement the table
  6. Implement the algorithm to interpret the table

- This is exactly the procedure that a **scanner generator** is implementing.
- Scanner generators include:
  - Lex, flex
  - Jlex
  - Alex
  - Lexgen
  - re2c

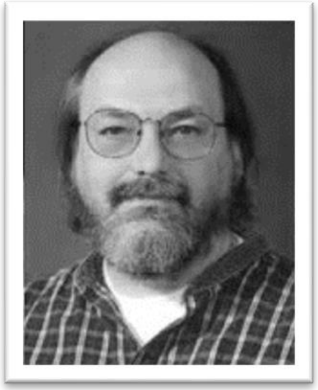
Regular expressions

```
ε         : { }
s         : {s | s in s^}
a         : {a}
r | s     : {r | r in r^} or {s | s in s^}
s*        : {s^n | s in s^ and n>=0}
s+        : {s^n | s in s^ and n>=1}
```

```
id ::= letter(letter|digit)*
```

Deriving DFA from REs

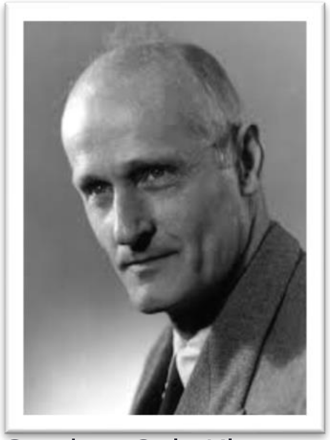
- ***Thompson's construction*** is an algorithm invented by Ken Thompson in 1968 to translate regular expressions into an NFA.
- ***Rabin-Scott powerset construction*** is an algorithm invented by Michael O. Rabin and Dana Scott in 1959 to transform an NFA to a DFA.
- ***Kleene's algorithm***, is an algorithm invented by Stephen Cole Kleene in 1956 to transform a DFA into a regular expression.
- These algorithms are the basis of the implementation of all scanner generators.



Ken Thompson

Michael O. Rabin

Dana Scott

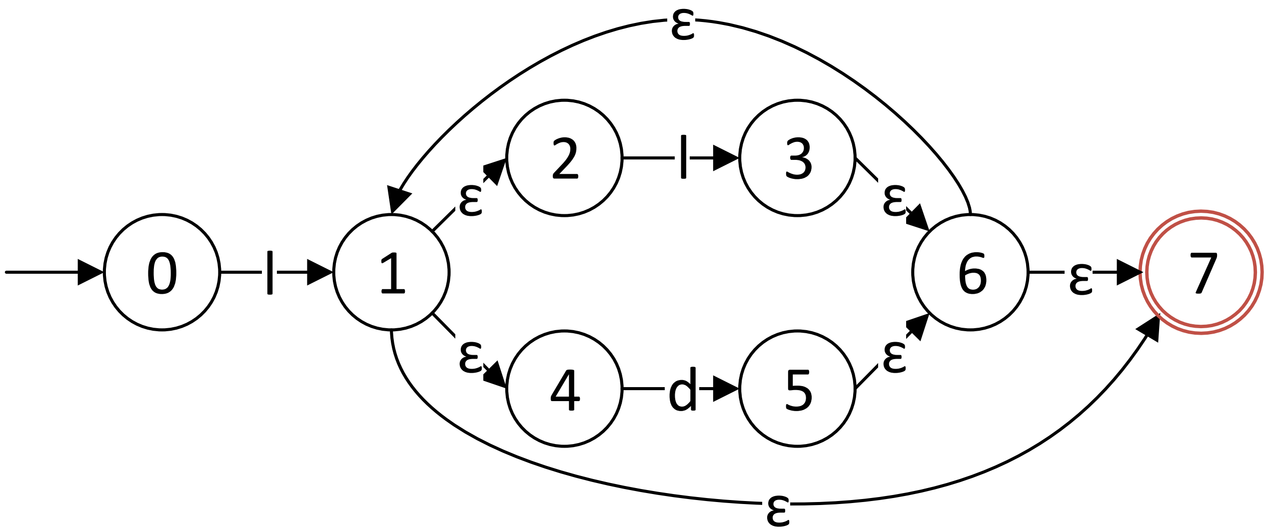


Stephen Cole Kleene

Thompson’s construction

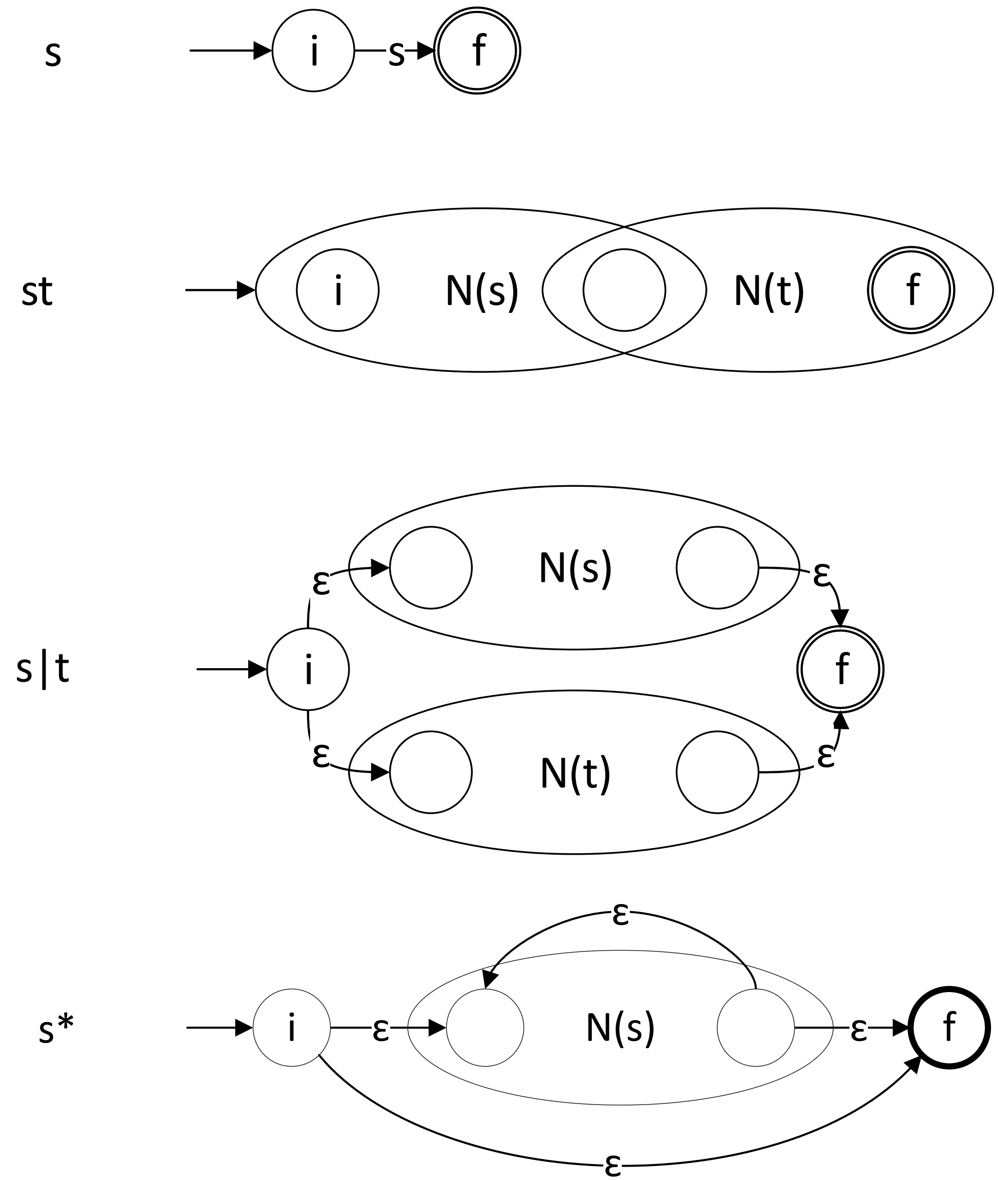
REs to NDFA: Thompson's construction

```
id ::= letter(letter|digit)*
```



Thompson's construction

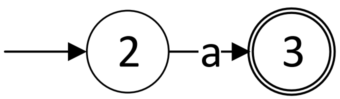
- Thompson's construction works recursively by splitting an expression into its constituent subexpressions.
- Each subexpression corresponds to a subgraph.
- Each subgraph is then grafted with other subgraphs depending on the nature of the composed subexpression, i.e.
  - An atomic lexical symbol
  - A concatenation expression
  - A union expression
  - A Kleene star expression



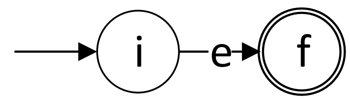
Thompson's construction: example

(a|b)*abb

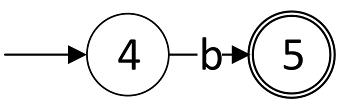
a



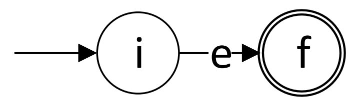
e



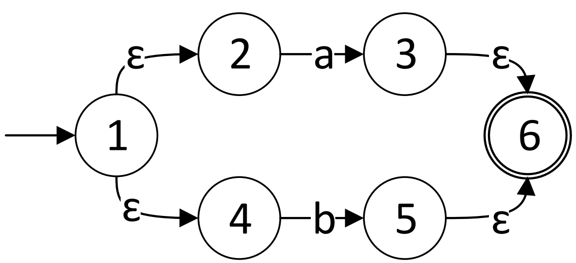
b



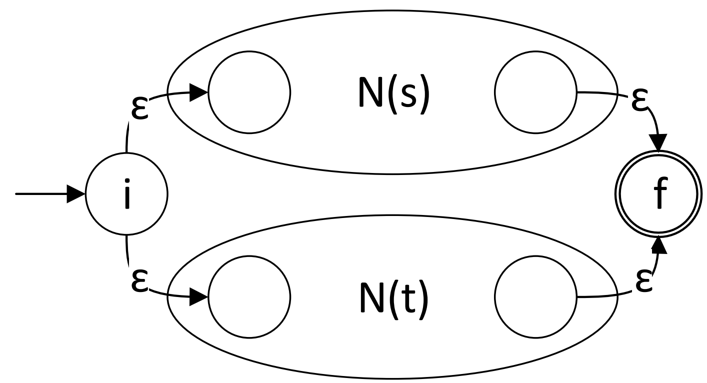
e



a|b



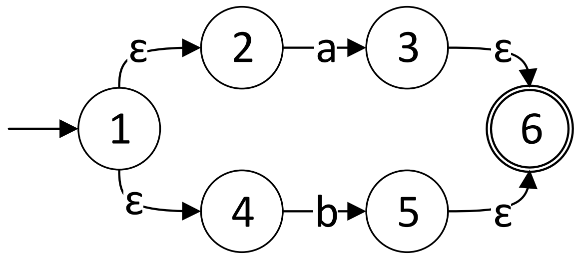
s|t



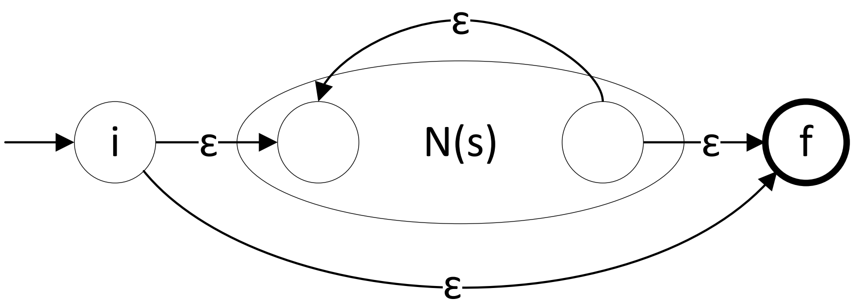
Thompson's construction: example

(a|b)*abb

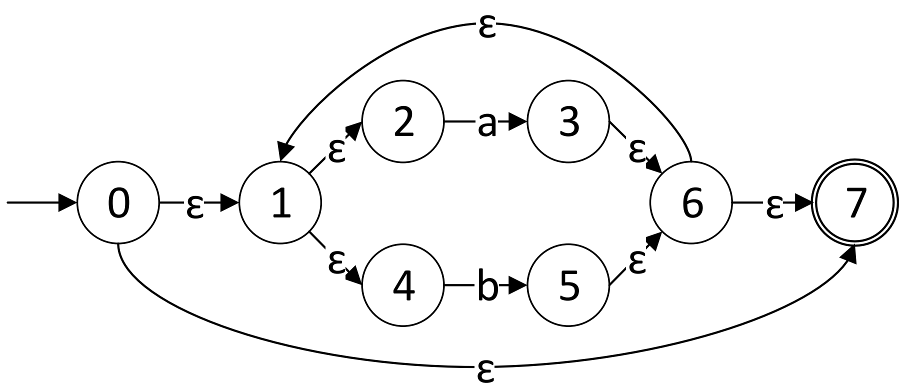
a|b



s*



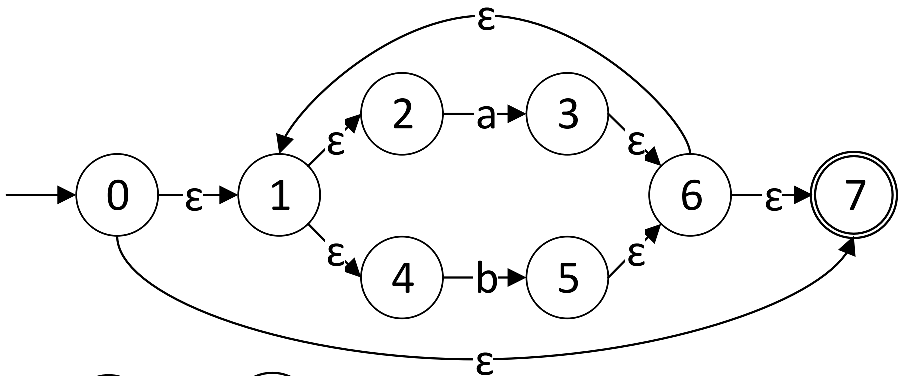
(a|b)*



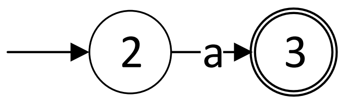
Thompson’s construction: example

(a|b)*abb

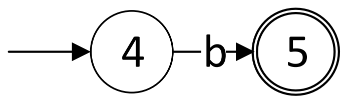
(a|b)*



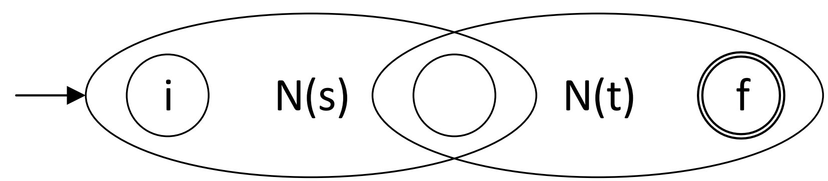
a



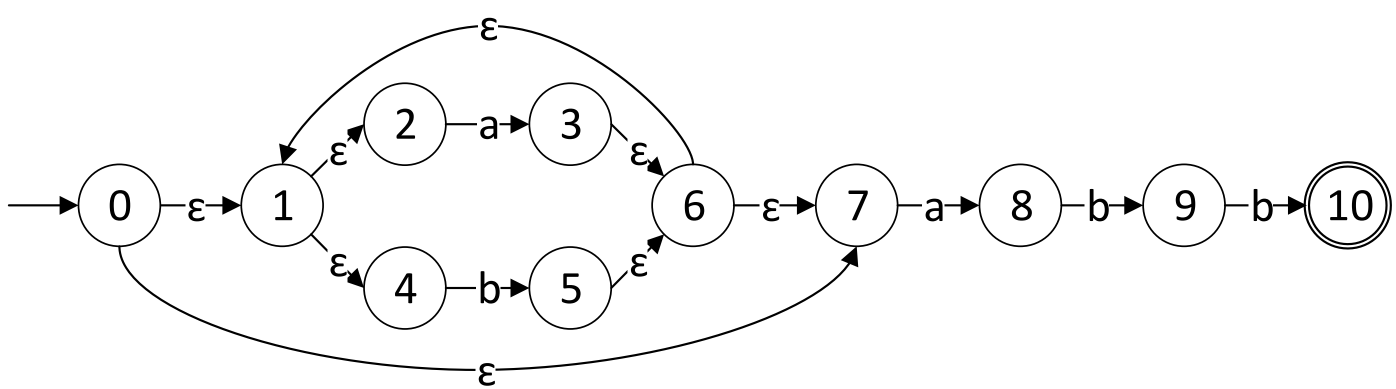
b



st



(a|b)*abb



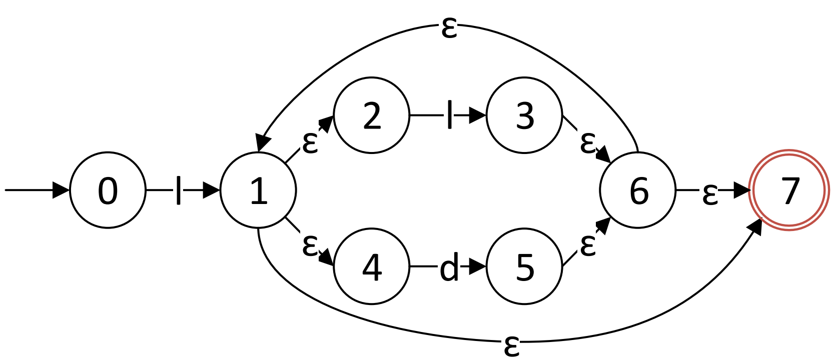
Rabin-Scott powerset construction

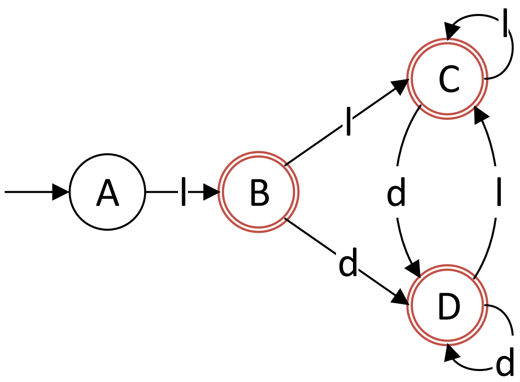
Rabin-Scott powerset construction: concepts

- S_{DFA}: set of states in the DFA
- S_{NFA}: set of states in the NFA
- Σ: set of all symbols in the lexical specification.

- `ε-closure(S)`: set of states in the NDFA that can be reached with ε transitions from any element of the set of states S, including the state itself.
- $Move_{NFA}(T,a)$: state in S_{NFA} to which there is a transition from one of the states in states set T, having encountered symbol a.
- $Move_{DFA}(T,a)$: state in S_{DFA} to which there is a transition from one of the states in states set T, having encountered symbol a.

```
id ::= letter(letter|digit)*
```

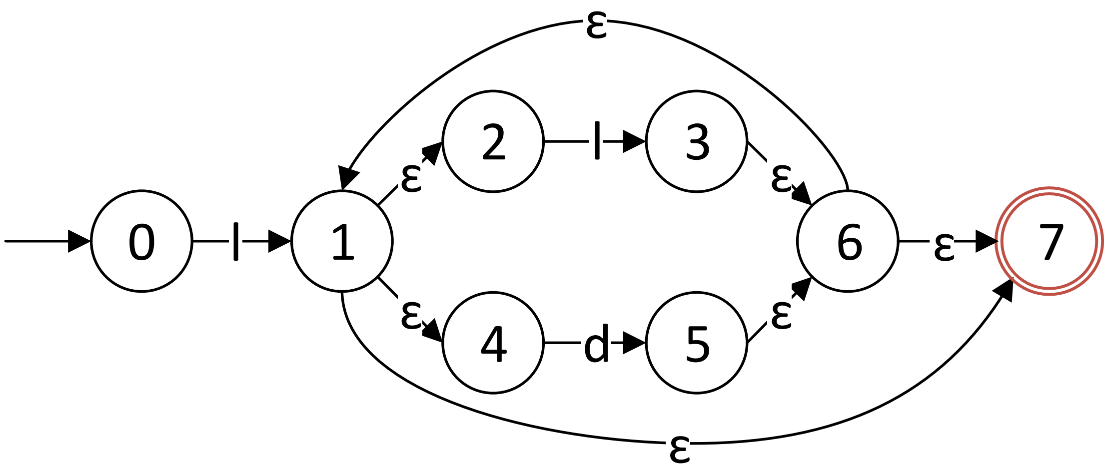


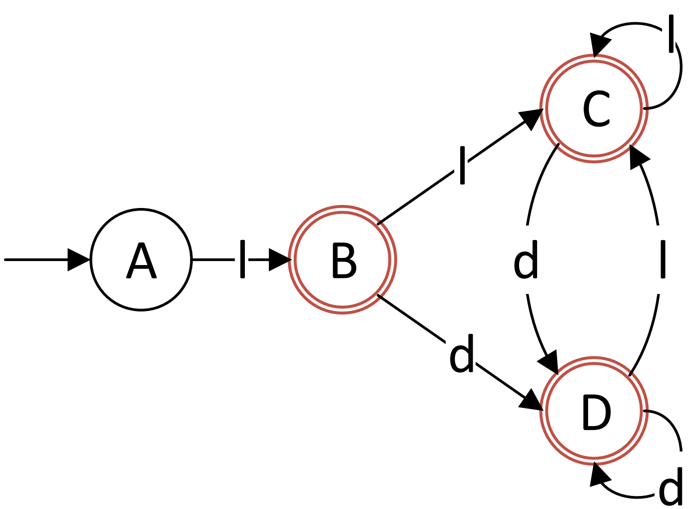


Rabin-Scott powerset construction: algorithm

```
S_DFA = {}
add ε-closure(S_0) to S_DFA as the start state
set this state as unmarked
while (S_DFA contains unmarked states)
   let T be an unmarked state in S_DFA and mark T
   for (each a in Σ)
      S = ε-closure(Move_NFA(T,a))
      if S is not in S_DFA
         add S to S_DFA as unmarked
      set Move_DFA(T,a) to S
for (each S in S_DFA)
   if any s∈S is a final state in the NFA
      mark s as a final state in the DFA
```

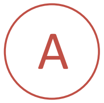
Rabin-Scott powerset construction: example





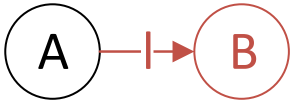
Rabin-Scott powerset construction: example

Starting state A = ε-closure(0) = {0}



State A : {0}

$move_{DFA}$(A,l)
= ε-closure($move_{NFA}$(A,l))
= ε-closure({1})
= {1,2,4,7}
= B

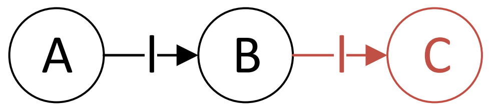


$move_{DFA}$(A,d)
= ε-closure($move_{NFA}$(A,d))
= ε-closure({})
= {}

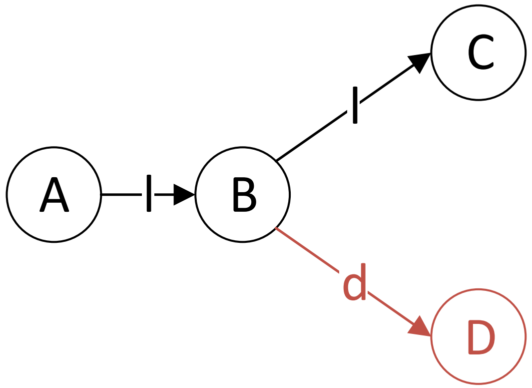
Rabin-Scott powerset construction: example

```
State B : {1,2,4,7}
```

$move_{DFA}$(B,l)
= ε-closure($move_{NFA}$(B,l))
= ε-closure({3})
= {1,2,3,4,6,7}
= C



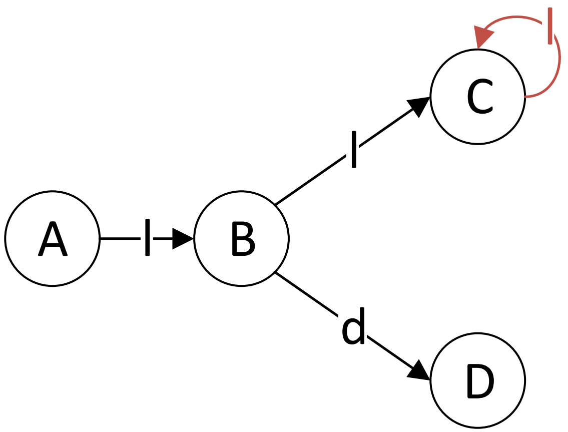
$move_{DFA}$(B,d)
= ε-closure($move_{NFA}$(B,d))
= ε-closure({5})
= {1,2,4,5,6,7}
= D



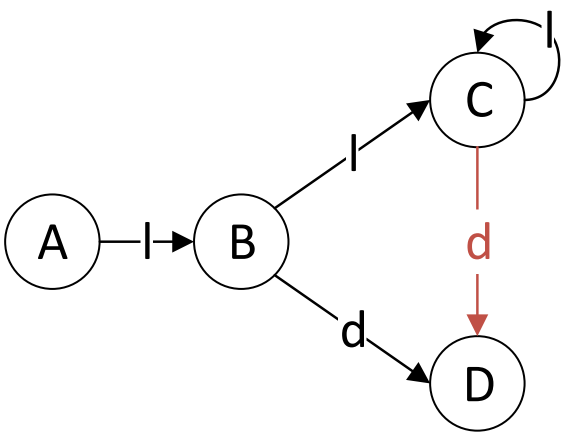
Rabin-Scott powerset construction: example

State C : {1,2,3,4,6,7}

$move_{DFA}$(C,l)
= ε-closure($move_{NFA}$(C,l))
= ε-closure({3})
= {1,2,3,4,6,7}
= C



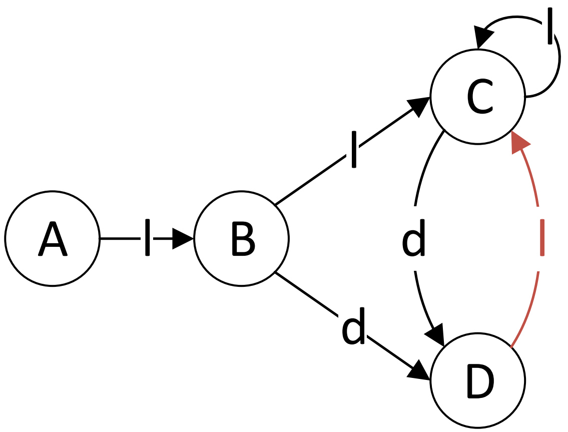
$move_{DFA}$(C,d)
= ε-closure($move_{NFA}$(C,d))
= ε-closure({5})
= {1,2,4,5,6,7}
= D



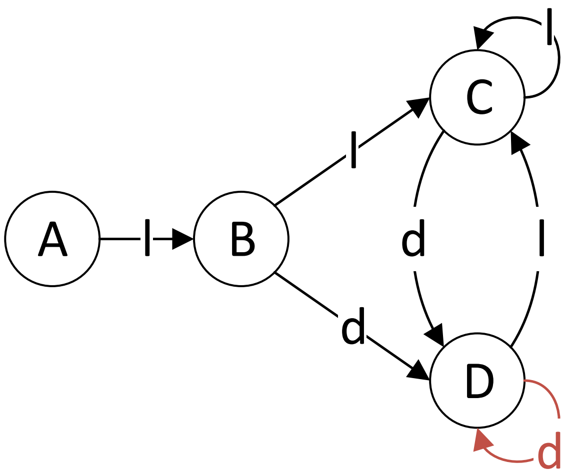
Rabin-Scott powerset construction: example

State D : {1,2,4,5,6,7}

$move_{DFA}(D,l)$
$= \varepsilon\text{-closure}(move_{NFA}(D,l))$
$= \varepsilon\text{-closure}(\{3\})$
$= \{1,2,3,4,6,7\}$
$= C$

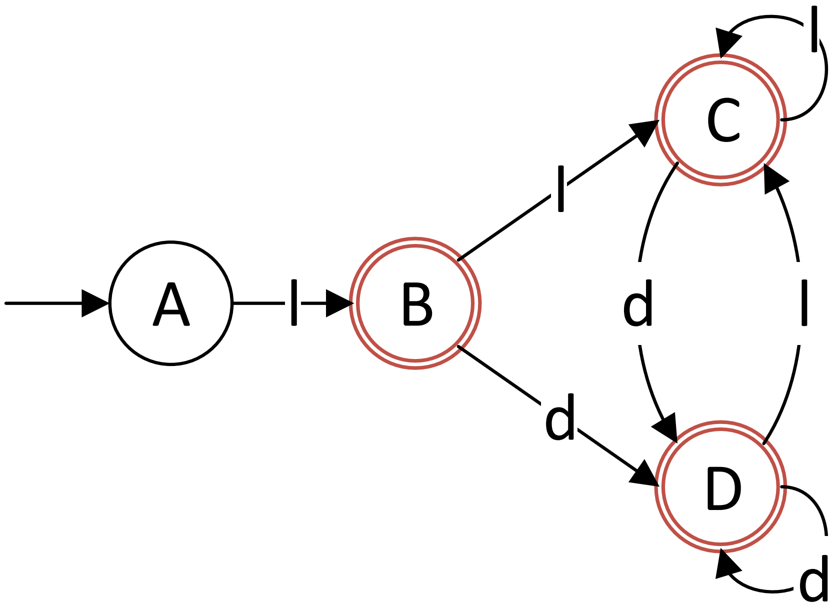


$move_{DFA}(D,d)$
$= \varepsilon\text{-closure}(move_{NFA}(D,d))$
$= \varepsilon\text{-closure}(\{5\})$
$= \{1,2,4,5,6,7\}$
$= D$

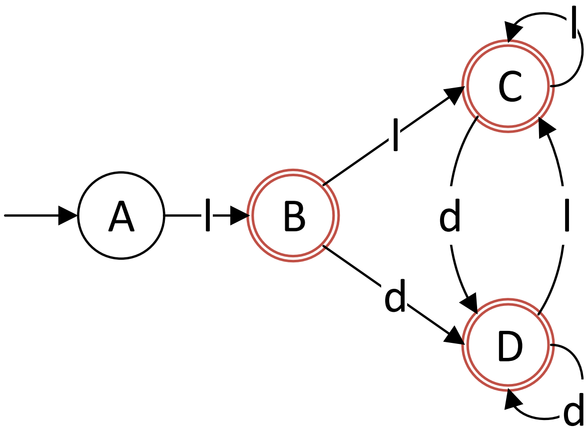


Rabin-Scott powerset construction: example

```
Final states:
```



Generate state transition table



state	letter	digit	final
A	B		N
B	C	D	Y
C	C	D	Y
D	C	D	Y

Implementation

Implementation concerns

- Backtracking
  - Principle : A token is normally recognized only when the next character is read.
  - Problem : Maybe this character is part of the next token.
  - Example : **`x<1`** "**`<`**" is recognized only when "**`1`**" is read. In this case, we have to backtrack one character to continue token recognition without skipping the first character of the next token.
  - Solution : include the occurrence of these cases in the state transition table.

- Ambiguity
  - Problem : Some tokens' lexemes are subsets of other tokens.
  - Example :
    - **`n-1`**. Is it `<n><-><1>` or `<n><-1>`?
  - Solutions :
    - Postpone the decision to the syntactic analyzer
    - Do not allow sign prefix to numbers in the lexical specification
    - Interact with the syntactic analyzer to find a solution. (Induces coupling)

Example

- Alphabet :
  - `{:, *, =, (, ), <, >, {, }, [a..z], [0..9]}`
- Simple tokens :
  - `{(, ), :, <, >}`
- Composite tokens :
  - `{:=, >=, <=, <>, (*, *)}`
- Words :
  - `id ::= letter(letter | digit)*`
  - `num ::= digit*`
  - `{…}` or `(*…*)` represent comments

Example

- Ambiguity problems

character	possible tokens
:	:, :=
>	>, >=
<	<, <=, <>
(	(, (*
*	*, *)

- Solution: Backtracking
  - Must back up a character when we read a character that is part of the next token.
  - Each case is encoded in the table

Example - DFA

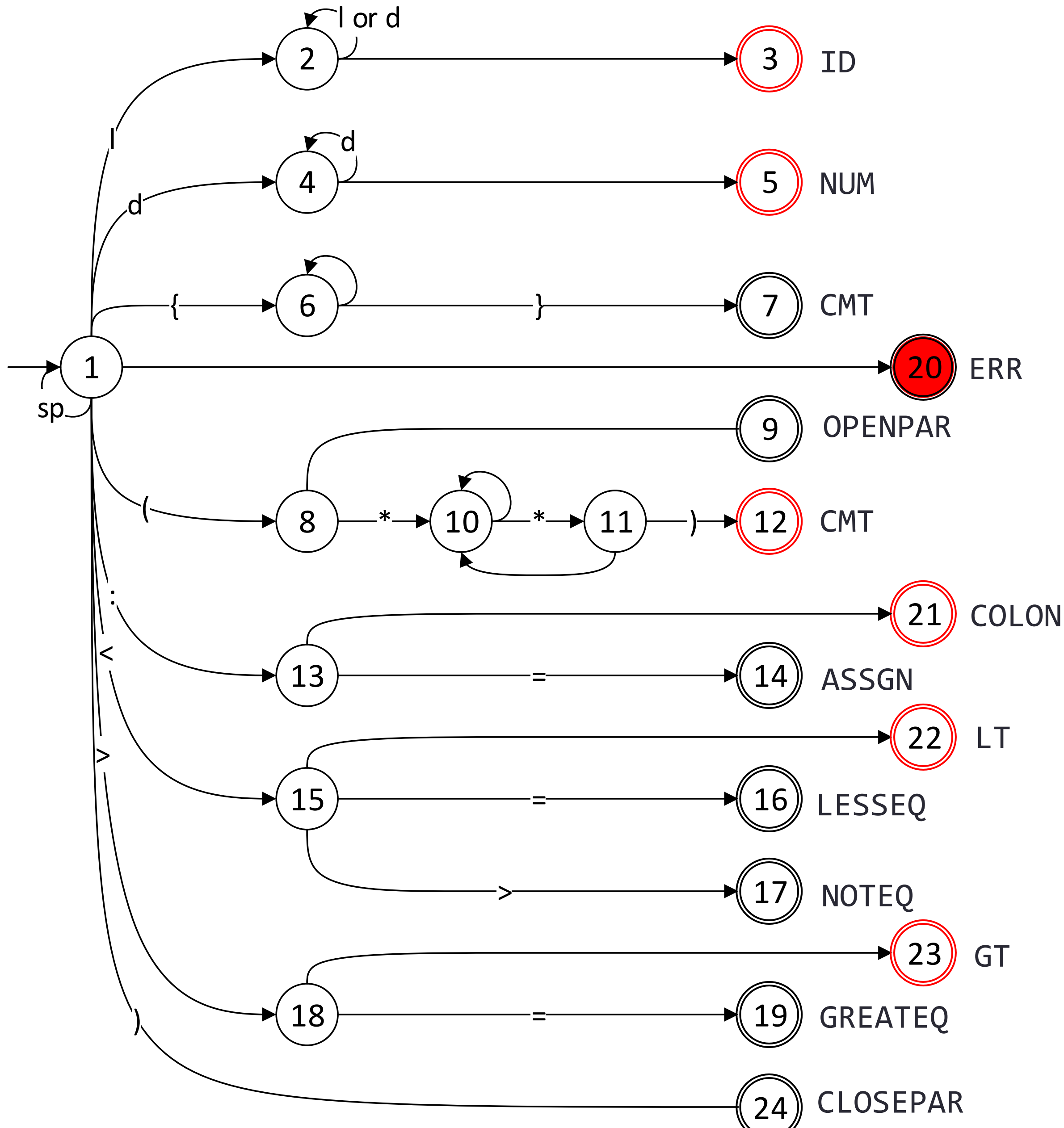

l or d
2
3
ID
l
d
4
5
NUM
d
{
6
}
7
CMT
1
sp
20
ERR
9
OPENPAR
(
8
*
10
*
11
)
12
CMT
:
21
COLON
13
=
14
ASSGN
<
22
LT
15
=
16
LESSEQ
>
17
NOTEQ
>
23
GT
18
=
19
GREATEQ
)
24
CLOSEPAR

Table-driven scanner – state transition table

	l	d	{	}	(	*	)	:	=	<	>	sp	final [token]	Backtrack
1	2	4	6	20	8	20	20	13	20	15	18	1		
2	2	2	3	3	3	3	3	3	3	3	3	3		
3	1	1	1	1	1	1	1	1	1	1	1	1	yes [id]	yes
4	5	4	5	5	5	5	5	5	5	5	5	5		
5	1	1	1	1	1	1	1	1	1	1	1	1	yes [num]	yes
6	6	6	6	7	6	6	6	6	6	6	6	6		
7	1	1	1	1	1	1	1	1	1	1	1	1	yes [cmt]	no
8	9	9	9	9	9	10	9	9	9	9	9	9		
9	1	1	1	1	1	1	1	1	1	1	1	1	yes [openpar]	no
10	10	10	10	10	10	11	10	10	10	10	10	10		
11	10	10	10	10	10	10	12	10	10	10	10	10		
12	1	1	1	1	1	1	1	1	1	1	1	1	yes [cmt]	yes
13	21	21	21	21	21	21	21	21	14	21	21	21		
14	1	1	1	1	1	1	1	1	1	1	1	1	yes [assgn]	no
15	22	22	22	22	22	22	22	22	16	22	17	22		
16	1	1	1	1	1	1	1	1	1	1	1	1	yes [lesseq]	no
17	1	1	1	1	1	1	1	1	1	1	1	1	yes [noteq]	no
18	23	23	23	23	23	23	23	23	19	23	23	23		
19	1	1	1	1	1	1	1	1	1	1	1	1	yes [gt]	no
20	1	1	1	1	1	1	1	1	1	1	1	1	yes [err]	no
21	1	1	1	1	1	1	1	1	1	1	1	1	yes [colon]	yes
22	1	1	1	1	1	1	1	1	1	1	1	1	yes [lt]	yes
23	1	1	1	1	1	1	1	1	1	1	1	1	yes [gt]	yes
24	1	1	1	1	1	1	1	1	1	1	1	1	yes [closepar]	no

Table-driven scanner - algorithm

```
nextToken()
  state = 1
  token = null
  do
    lookup = nextChar()
    state = table(state, lookup)
    if (isFinalState(state))
      token = createToken(state)
      if (table(state, “backup”) == yes)
        backupChar()
  until (token != null)
  return (token)
```

Table-driven scanner – functions

- `nextToken()`
  - Extract the next token in the program (called by syntactic analyzer)
- `nextChar()`
  - Read the next character in the input program
- `backupChar()`
  - Back up one character in the input file in case we have just read the next character in order to resolve an ambiguity
- `isFinalState(state)`
  - Returns TRUE if state is a final state
- `table(state, column)`
  - Returns the value corresponding to `[state, column]` in the state transition table.
- `createToken(state)`
  - Creates and returns a structure that contains the token type, its location in the source code, and its value (for literals), for the token kind corresponding to a state, as found in the state transition table.

Hand-written scanner

```
nextToken()
  c = nextChar()
  case (c) of
    "[a..z],[A..Z]":
        c = nextChar()
        while (c in {[a..z],[A..Z],[0..9]}) do
          s = makeUpString()
          c = nextChar()
        if ( isReservedWord(s) )then
          token = createToken(RESWORD,null)
        else
          token = createToken(ID,s)
        backupChar()
    "[0..9]":
        c = nextChar()
        while (c in [0..9]) do
          v = makeUpValue()
          c = nextChar()
        token = createToken(NUM,v)
        backupChar()
```

Hand-written scanner

```
"{":
    c = nextChar()
    while ( c != "}" ) do
      c = nextChar()
"(":
    c = nextChar()
    if ( c == "*" ) then
      c = nextChar()
      repeat
        while ( c != "*" ) do
          c = nextChar()
        c = nextChar()
      until ( c != ")" )
    else
      token = createToken(LPAR,null)
":":
    c = nextChar()
    if ( c == "=" ) then
      token = createToken(ASSIGNOP,null)
    else
      token = createToken(COLON,null)
      backupChar()
```

Hand-written scanner

```
        "<":
            c = nextChar()
            if ( c == "=" ) then
              token = createToken(LEQ,null)
            else if ( c == ">" ) then
              token = createToken(NEQ,null)
            else
              token = createToken(LT,null)
              backupChar()
        ">":
            c = nextChar()
            if ( c == "=" ) then
              token = createToken(GEQ,null)
            else
              token = createToken(GT,null)
              backupChar()
        ")":
            token = createToken(RPAR,null)
        "*":
            token = createToken(STAR,null)
        "=":
            token = createToken(EQ,null)
      end case
    return token
```

Error-recovery in lexical analysis

Possible lexical errors

- Depends on the accepted conventions:
  - Invalid character
  - letter not allowed to terminate a number
  - numerical overflow
  - identifier too long
  - end of line before end of string
  - Are these lexical errors?

`123a`

<Error> or <num><id>?

`12345678901234567890123456`7

<Error> related to machine's limitations

`“Hello <CR> world`

Either <CR> is skipped or <Error>

`ThisIsAVeryLongVariableNameThatIsMeantToConveyMeaning = 1`

Limit identifier length?

Lexical error recovery techniques

- Finding only the first error is not acceptable
- Panic Mode:
  - Skip characters until a valid character is read
- Guess Mode:
  - do pattern matching between erroneous strings and valid strings
  - Example: (beggin vs. begin)
  - Rarely implemented

Conclusions

Possible implementations

- Lexical Analyzer Generator (e.g. Lex)
  - \+ safe, quick
  - \- Must learn software, unable to handle unusual situations
- Table-Driven Lexical Analyzer
  - \+ general and adaptable method, same function can be used for all table-driven lexical analyzers
  - \- Building transition table can be tedious and error-prone
- Hand-written
  - \+ Can be optimized, can handle any unusual situation, easy to build for most languages
  - \- Error-prone, not adaptable or maintainable

Lexical analyzer's modularity

- Why should the Lexical Analyzer and the Syntactic Analyzer be separated?
  - Modularity/Maintainability : system is more modular, thus more maintainable
  - Efficiency : modularity = task specialization = easier optimization
  - Reusability : can change the whole lexical analyzer without changing other parts

References

- R. McNaughton, H. Yamada (Mar 1960). "Regular Expressions and State Graphs for Automata". IEEE Trans. on Electronic Computers 9 (1): 39–47. doi:10.1109/TEC.1960.5221603
- Ken Thompson (Jun 1968). "Programming Techniques: Regular expression search algorithm". Communications of the ACM 11 (6): 419–422. doi:10.1145/363347.363387
- Rabin, M. O.; Scott, D. (1959). "Finite automata and their decision problems". IBM Journal of Research and Development 3 (2): 114–125. doi:10.1147/rd.32.0114
- Russ Cox. Implementing Regular Expressions.
- Russ Cox. Regular Expression Matching Can Be Simple And Fast.
- CyberZHG. Regular Expression to NFA, to DFA.