Chapter 1

Scanning

Course “Compiler Construction”
Martin Steffen
Spring 2018

Chapter 1

Learning Targets of Chapter "Scanning".

1. alphabets, languages,
2. regular expressions
3. finite state automata / recognizers
4. connection between the two concepts
5. minimization

The material corresponds roughly to [1, Section 2.1–2.5] or ar large part of [3, Chapter 2]. The material is pretty canonical anyway.

Chapter 1

Outline of Chapter “Scanning”.

Section

Intro

Chapter 1 “Scanning”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Scanner section overview

What's a scanner?

- Input: source code.[1]
- Output: sequential stream of tokens

- *regular expressions* to describe various token classes
- (deterministic/non-determinstic) finite-state automata (FSA, DFA, NFA)
- implementation of FSA
- regular expressions → NFA
- NFA ↔ DFA

[1] The argument of a scanner is often a *file name* or an *input stream* or similar.

What's a scanner?

- other names: lexical scanner, lexer, tokenizer

A scanner's functionality

Part of a compiler that takes the source code as input and translates this stream of characters into a stream of tokens.

- char's typically language independent.[2]
- *tokens* already language-specific.[3]
- works always "left-to-right", producing one *single token* after the other, as it scans the input[4]
- it "segments" char stream into "chunks" while at the same time "classifying" those pieces ⇒ tokens

[2] Characters are language-independent, but perhaps the encoding (or its interpretation) may vary, like ASCII, UTF-8, also Windows-vs.-Unix-vs.-Mac newlines etc.

[3] There are large commonalities across many languages, though.

[4] No theoretical necessity, but that's how also humans consume or "scan" a source-code text. At least those humans trained in e.g. Western languages.

Typical responsibilities of a scanner

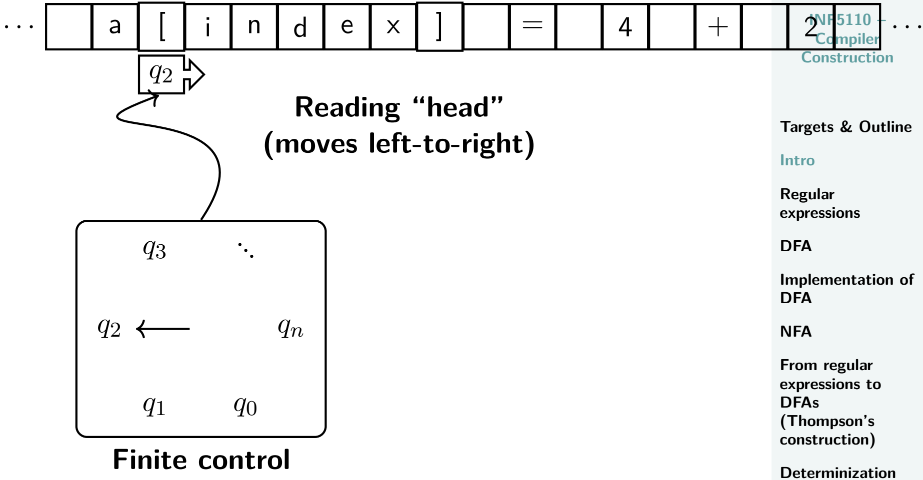
- segment & classify char stream into tokens
- typically described by "rules" (and regular expressions)
- typical language aspects covered by the scanner
  - describing *reserved words* or *key words*
  - describing format of *identifiers* (= "strings" representing variables, classes ...)
  - comments (for instance, between // and `NEWLINE`)
  - *white space*
    - to segment into tokens, a scanner typically "jumps over" white spaces and afterwards starts to determine a new token
    - not only "blank" character, also `TAB`, `NEWLINE`, etc.
- lexical rules: often (explicit or implicit) *priorities*
  - *identifier* or *keyword*? ⇒ keyword
  - take the *longest* possible scan that yields a valid token.

“Scanner = regular expressions (+ priorities)”

Rule of thumb

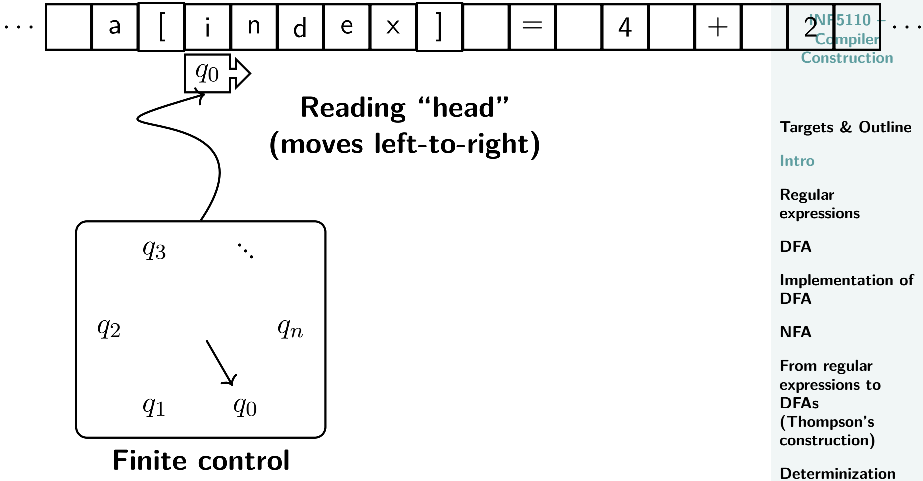
Everything about the source code which is so simple that it can be captured by reg. expressions belongs into the scanner.

How does scanning roughly work?



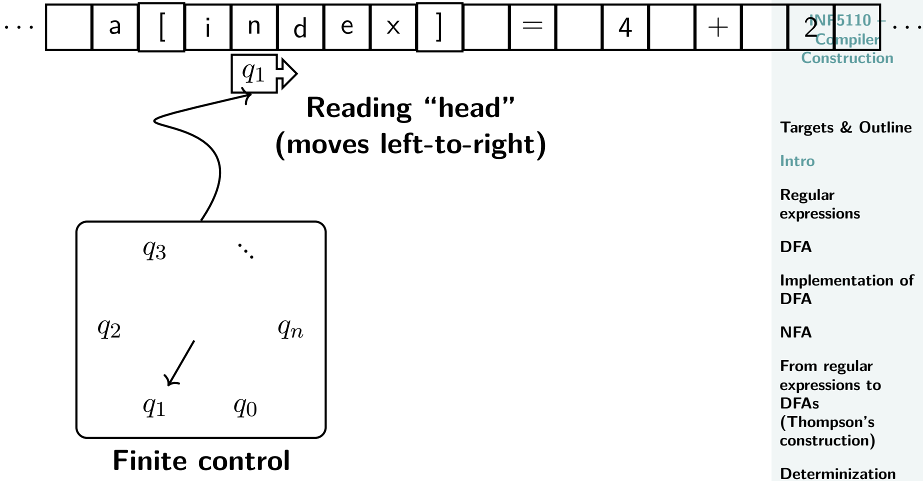
`a[index] = 4 + 2`

How does scanning roughly work?



`a[index] = 4 + 2`

How does scanning roughly work?




```
a[index] = 4 + 2
```

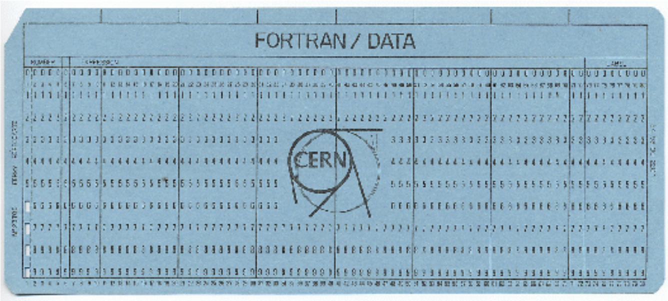
How does scanning roughly work?

- usual invariant in such pictures (by convention): arrow or head points to the *first* character to be *read next* (and thus *after* the last character having been scanned/read last)
- in the scanner *program* or procedure:
  - analogous invariant, the arrow corresponds to a *specific variable*
  - contains/points to the next character to be read
  - name of the variable depends on the scanner/scanner tool
- the *head* in the pic: for illustration, the scanner does not really have a "reading head"
  - remembrance of Turing machines, or
  - the old times when perhaps the program data was stored on a tape.[5]

[5] Very deep down, if one still has a magnetic disk (as opposed to SSD) the secondary storage still has "magnetic heads", only that one typically does not parse *directly* char by char from disk...

The bad(?) old times: Fortran

- in the days of the pioneers
- main memory was *smaaaaaaaaaall*
- compiler technology was not well-developed (or not at all)
- programming was for *very* few "experts".[6]
- Fortran was considered very high-level (wow, a language so complex that you had to compile it . . .)



[6] There was no computer science as profession or university curriculum.

(Slightly weird) lexical ascpects of Fortran

Lexical aspects = those dealt with a scanner

- whitespace *without* "meaning":

  `I F( X 2.  EQ. 0) TH E N` vs. `IF ( X2. EQ.0 ) THEN`

- no reserved words!

  `IF (IF.EQ.0) THEN THEN=1.0`

- general *obscurity* tolerated:

  `DO99I=1,10` vs. `DO99I=1.10`

```
DO 99 I=1,10
 -
 -
99 CONTINUE
```

Fortran scanning: remarks

- Fortran (of course) has evolved from the pioneer days ...
- no keywords: nowadays mostly seen as *bad* idea[7]
- treatment of white-space as in Fortran: not done anymore: `THEN` and `TH EN` *are* different things in all languages
- however:[8] both considered "the same":

```
if␣b␣then␣..
```

```
if␣␣␣b␣␣␣␣then␣..
```

- since concepts/tools (and much memory) were missing, Fortran scanner and parser (and compiler) were
  - quite simplistic
  - syntax: designed to "help" the lexer (and other phases)

[7] It's mostly a question of language *pragmatics*. Lexers/parsers would have no problems using `while` as variable, but humans tend to.

[8] Sometimes, the part of a lexer / parser which removes whitespace (and comments) is considered as separate and then called *screener*. Not

A scanner classifies

- "good" classification: depends also on later phases, may not be clear till later

Rule of thumb

Things being treated equal in the syntactic analysis (= parser, i.e., subsequent phase) should be put into the same category.

- terminology not 100% uniform, but most would agree:

Lexemes and tokens

Lexemes are the "chunks" (pieces) the scanner produces from segmenting the input source code (and typically dropping whitespace). Tokens are the result of *classifying* those lexemes.

- token = token name × token value

A scanner classifies & does a bit more

- token data structure in *OO* settings
  - token themselves defined by classes (i.e., as instance of a class representing a specific token)
  - token values: as attribute (instance variable) in its values
- often: scanner does slightly *more* than just classification
  - store names in some *table* and store a corresponding index as attribute
  - store text constants in some *table*, and store corresponding index as attribute
  - even: *calculate* numeric constants and store value as attribute

One possible classification

name/identifier	`abc123`
integer constant	`42`
real number constant	`3.14E3`
text constant, string literal	`"this is a text constant"`
arithmetic op's	`+ - * /`
boolean/logical op's	`and or not` (alternatively `/\ \/`)
relational symbols	`<= < >= > = == !=`
all other tokens:	`{ } ( ) [ ] , ; := .` etc.
every one it its own group	

- this classification: not the only possible (and not necessarily complete)
- note: *overlap*:
  - `"."` is here a token, but also part of real number constant
  - `"<"` is part of `"<="`

One way to represent tokens in C

```
typedef struct {
  TokenType tokenval;
  char * stringval;
  int numval;
} TokenRecord;
```

If one only wants to store one attribute:

```
typedef struct {
  Tokentype tokenval;
  union
  { char * stringval;
    int numval
  } attribute;
} TokenRecord;
```

How to define lexical analysis and implement a scanner?

- even for complex languages: lexical analysis (in principle) not hard to do
- "manual" implementation straightforwardly possible
- *specification* (e.g., of different token classes) may be given in "prosa"
- however: there are straightforward formalisms and efficient, rock-solid tools available:
  - easier to specify unambigously
  - easier to communicate the lexical definitions to others
  - easier to change and maintain
- often called parser generators typically not just generate a scanner, but code for the next phase (parser), as well.

Section

Regular expressions

Chapter 1 “Scanning”
Course “Compiler Construction”
Martin Steffen
Spring 2018

General concept: How to generate a scanner?

1. regular expressions to describe language's *lexical* aspects
   - like whitespaces, comments, keywords, format of identifiers etc.
   - often: more "user friendly" variants of reg-exprs are supported to specify that phase
2. *classify* the lexemes to tokens
3. translate the reg-expressions ⇒ NFA.
4. turn the NFA into a *deterministic* FSA (= DFA)
5. the DFA can straightforwardly be implementated

- step done automatically by a "lexer generator"
- lexer generators help also in other user-friendly ways of specifying the lexer: defining *priorities*, assuring that the longest possible token is given back, repeat the processs to generate a sequence of tokens[9]

[9] Maybe even prepare useful error messages if scanning (not scanner generation) fails.

Use of regular expressions

- regular languages: fundamental class of "languages"
- regular expressions: standard way to describe regular languages
- not just used in compilers
- often used for flexible " *searching* ": simple form of pattern matching
- e.g. input to search engine interfaces
- also supported by many editors and text processing or scripting languages (starting from classical ones like `awk` or `sed`)
- but also tools like `grep` or `find` (or general "globbing" in shells)

```
find . -name "*.tex"
```

- often *extended* regular expressions, for user-friendliness, not theoretical expressiveness

Alphabets and languages

Definition (Alphabet Σ)

Finite set of elements called "letters" or "symbols" or "characters".

Definition (Words and languages over Σ)

Given alphabet Σ, a word over Σ is a finite sequence of letters from Σ. A language over alphabet Σ is a *set* of finite *words* over Σ.

- in this lecture: we avoid terminology "symbols" for now, as later we deal with e.g. symbol tables, where symbols means something slighly different (at least: at a different level).
- Sometimes Σ left "implicit" (as assumed to be understood from the context)
- practical examples of alphabets: ASCII, Norwegian letters (capital and non-capitals) etc.

Languages

- note: Σ is finite, and words are of *finite* length
- languages: in general infinite sets of words
- simple examples: Assume $\Sigma = \{a, b\}$
- *words* as finite "sequences" of letters
  - ϵ: the empty word (= empty sequence)
  - ab means " first a then b "
- sample languages over Σ are
  1. $\{\}$ (also written as $\emptyset$) the empty set
  2. $\{a, b, ab\}$: language with 3 finite words
  3. $\{\epsilon\}$ ($\neq \emptyset$)
  4. $\{\epsilon, a, aa, aaa, \ldots\}$: infinite languages, all words using only a 's.
  5. $\{\epsilon, a, ab, aba, abab, \ldots\}$: alternating a's and b's
  6. $\{ab, bbab, aaaaa, bbabbabab, aabb, \ldots\}$: ?????

How to describe languages

- language mostly here in the abstract sense just defined.
- the "dot-dot-dot" (...) is not a good way to describe to a computer (and to many humans) what is meant
- enumerating explicitly all allowed words for an infinite language does not work either

Needed

A finite way of describing infinite languages (which is hopefully efficiently implementable & easily readable)

Beware

Is it apriori to be expected that *all* infinite languages can even be captured in a finite manner?

- small metaphor

$$2.727272727\ldots \qquad 3.1415926\ldots \tag{1}$$

Regular expressions

Definition (Regular expressions)

A *regular expression* is one of the following

1. a *basic* regular expression of the form $\mathbf{a}$ (with $a \in \Sigma$), or $\boldsymbol{\epsilon}$, or $\varnothing$
2. an expression of the form $r \mid s$, where r and s are regular expressions.
3. an expression of the form $r\,s$, where r and s are regular expressions.
4. an expression of the form r^*, where r is a regular expression.

Precedence (from high to low): $*$, concatenation, |

A “grammatical” definition

Later introduced as (notation for) context-free grammars:

$$
\begin{aligned}
r &\rightarrow \mathbf{a} \\
r &\rightarrow \boldsymbol{\epsilon} \\
r &\rightarrow \varnothing \\
r &\rightarrow r \mid r \\
r &\rightarrow r\, r \\
r &\rightarrow r^*
\end{aligned}
\qquad (2)
$$

Same again

Notational conventions

Later, for CF grammars, we use capital letters to denote "variables" of the grammars (then called *non-terminals*). If we like to be consistent with that convention, the definition looks as follows:

$$
\begin{array}{rcl}
R & \rightarrow & \mathbf{a} \\
R & \rightarrow & \boldsymbol{\epsilon} \\
R & \rightarrow & \varnothing \\
R & \rightarrow & R \mid R \\
R & \rightarrow & R\,R \\
R & \rightarrow & R^*
\end{array}
\tag{3}
$$

Symbols, meta-symbols, meta-meta-symbols ...

- regexprs: notation or "language" to describe "languages" over a given alphabet Σ (i.e. subsets of Σ^*)
- language being described $\Leftrightarrow$ language used to describe the language

$\Rightarrow$ language $\Leftrightarrow$ meta-language

- here:
  - regular expressions: notation to describe regular languages
  - English resp. context-free notation: notation to describe regular expression
- for now: carefully use *notational convention* for precision

Notational conventions

- notational conventions by *typographic* means (i.e., different fonts etc.)
- you need good eyes, but: difference between
  - $\boldsymbol{a}$ and a
  - $\boldsymbol{\epsilon}$ and ϵ
  - $\boldsymbol{\varnothing}$ and $\varnothing$
  - $\boldsymbol{|}$ and $|$ (especially hard to see :-))
  - ...
- later (when gotten used to it) we may take a more "relaxed" attitude toward it, assuming things are clear, as do many textbooks
- Note: in compiler *implementations*, the distinction between language and meta-language etc. is very real (even if not done by typographic means ...)

Same again once more

$$R \rightarrow \mathbf{a} \mid \epsilon \mid \emptyset \qquad \text{basic reg. expr.}$$
$$\mid \; R|R \mid R\,R \mid R^* \mid (R) \qquad \text{compound reg. expr.} \tag{4}$$

Note:

- symbol $|$: as symbol of regular expressions
- symbol $\mid$: meta-symbol of the CF grammar notation
- the meta-notation used here for CF grammars will be the subject of later chapters

Semantics (meaning) of regular expressions

Definition (Regular expression)

Given an alphabet Σ. The meaning of a regexp r (written $\mathcal{L}(r)$) over Σ is given by equation (5).

$$
\begin{array}{rcll}
\mathcal{L}(\varnothing) & = & \{\} & \text{empty language} \\
\mathcal{L}(\boldsymbol{\epsilon}) & = & \{\epsilon\} & \text{empty word} \\
\mathcal{L}(\boldsymbol{a}) & = & \{a\} & \text{single “letter” from } \Sigma \\
\mathcal{L}(rs) & = & \{w_1 w_2 \mid w_1 \in \mathcal{L}(r), w_2 \in \mathcal{L}(s)\} & \text{concatenation} \\
\mathcal{L}(r \mid s) & = & \mathcal{L}(r) \cup \mathcal{L}(s) & \text{alternative} \\
\mathcal{L}(r^{*}) & = & \mathcal{L}(r)^{*} & \text{iteration}
\end{array}
\tag{5}
$$

- conventional *precedences*: $*$, concatenation, |.
- Note: left of "=": reg-expr *syntax*, right of "=": semantics/meaning/math [10]

[10] Sometimes confusingly "the same" notation.

Examples

In the following:

- $\Sigma = \{a, b, c\}$.
- we don't bother to "boldface" the syntax

words with exactly one b	$(a \mid c)^* b (a \mid c)^*$
words with max. one b	$((a \mid c)^*) \mid ((a \mid c)^* b (a \mid c)^*)$
	$(a \mid c)^* \; (b \mid \epsilon) \; (a \mid c)^*$
words of the form $a^n b a^n$, i.e., equal number of a's before and after 1 b	

Another regexpr example

words that do not contain two b's in a row.

$(b\ (a \mid c))^*$		not quite there yet
$((a \mid c)^* \mid (b\ (a \mid c))^*)^*$		better, but still not there
	$=$	(simplify)
$((a \mid c) \mid (b\ (a \mid c)))^*$	$=$	(simplifiy even more)
$(a \mid c \mid ba \mid bc)^*$		
$(a \mid c \mid ba \mid bc)^*\ (b \mid \epsilon)$		potential b at the end
$(notb \mid b\ notb)^*(b \mid \epsilon)$		where $notb \triangleq a \mid c$

Additional "user-friendly" notations

$$
\begin{aligned}
r^+ &= rr^* \\
r? &= r \mid \epsilon
\end{aligned}
$$

Special notations for *sets* of letters:

$[0-9]$	range (for ordered alphabets)
$\sim a$	not a (everything except a)
$.$	all of Σ

naming regular expressions ("regular definitions")

$$
\begin{aligned}
digit &= [0-9] \\
nat &= digit^+ \\
signedNat &= (+|-)nat \\
number &= signedNat(\text{"."}nat)?(\texttt{E}\ signedNat)?
\end{aligned}
$$

Section

DFA

Chapter 1 “Scanning”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Finite-state automata

- simple "computational" machine
- (variations of) FSA's exist in many flavors and under different names
- other rather well-known names include finite-state machines, finite labelled transition systems,
- "state-and-transition" representations of programs or behaviors (finite state or else) are wide-spread as well
  - state diagrams
  - Kripke-structures
  - I/O automata
  - Moore & Mealy machines
- the logical behavior of certain classes of electronic circuitry with internal memory ("flip-flops") is described by finite-state automata.

FSA

Definition (FSA)

A FSA $\mathcal{A}$ over an alphabet Σ is a tuple $(\Sigma, Q, I, F, \delta)$

- Q: finite set of states
- $I \subseteq Q$, $F \subseteq Q$: initial and final states.
- $\delta \subseteq Q \times \Sigma \times Q$ transition relation

- final states: also called *accepting* states
- transition relation: can *equivalently* be seen as function $\delta : Q \times \Sigma \to 2^Q$: for each state and for each letter, give back the set of sucessor states (which may be empty)
- more suggestive notation: $q_1 \xrightarrow{a} q_2$ for $(q_1, a, q_2) \in \delta$
- we also use freely —self-evident, we hope— things like

$$q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3$$

FSA as scanning machine?

- FSA have slightly unpleasant properties when considering them as decribing an actual program (i.e., a scanner procedure/lexer)
- given the "theoretical definition" of acceptance:

Mental picture of a scanning automaton

The automaton eats one character after the other, and, when reading a letter, it moves to a successor state, if any, of the current state, depending on the character at hand.

- 2 problematic aspects of FSA
  - non-determinism: what if there is more than one possible successor state?
  - undefinedness: what happens if there's no next state for a given input
- the 2nd one is *easily* repaired, the 1st one requires more thought
- [1]: recogniser corresponds to DFA

DFA: deterministic automata

Definition (DFA)

A *deterministic, finite automaton* $\mathcal{A}$ (DFA for short) over an alphabet Σ is a tuple $(\Sigma, Q, I, F, \delta)$

- Q: finite set of states
- $I = \{i\} \subseteq Q$, $F \subseteq Q$: initial and final states.
- $\delta : Q \times \Sigma \to Q$ transition function

- transition function: special case of transition relation:
  - deterministic
  - left-total ("complete")

Meaning of an FSA

Semantics

The intended meaning of an FSA over an alphabet Σ is the set of all the finite words, the automaton accepts.

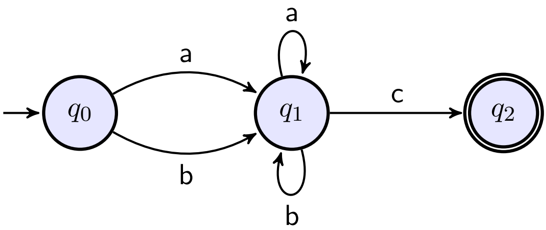
Definition (Accepting words and language of an automaton)

A word $c_1 c_2 \ldots c_n$ with $c_i \in \Sigma$ is accepted by automaton $\mathcal{A}$ over Σ, if there exists states $q_0, q_2, \ldots, q_n$ from Q such that

$$q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} q_2 \xrightarrow{c_3} \quad \ldots \quad q_{n-1} \xrightarrow{c_n} q_n \ ,$$

and were $q_0 \in I$ and $q_n \in F$. The language of an FSA $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$, is the set of all words that $\mathcal{A}$ accepts.

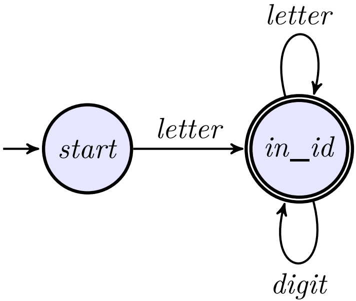
FSA example



Example: identifiers

Regular expression

$$identifier = letter(letter \mid digit)^* \tag{6}$$

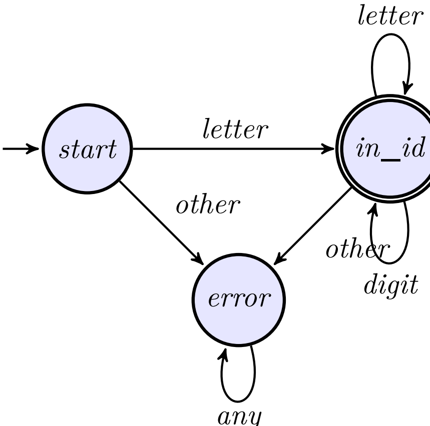


- transition *function*/relation δ *not* completely defined (= *partial* function)

Example: identifiers

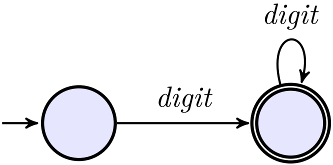
Regular expression

$$identifier = letter(letter \mid digit)^* \tag{6}$$



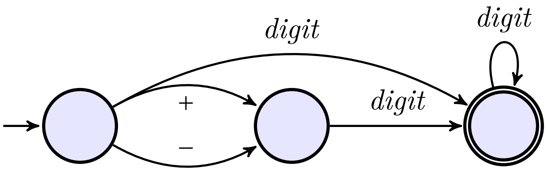
Automata for numbers: natural numbers

$$\begin{aligned} digit &= [0-9] \\ nat &= digit^{+} \end{aligned} \tag{7}$$

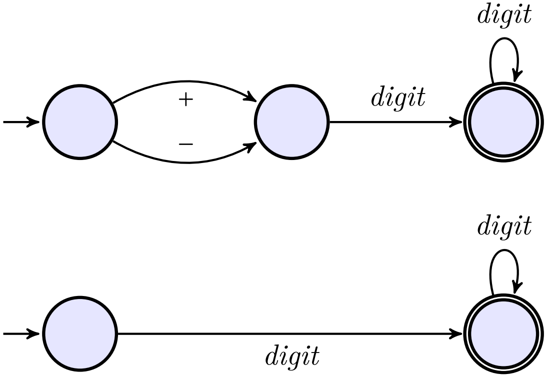


Signed natural numbers

$$signednat \quad = \quad (+ \mid -) nat \mid nat \tag{8}$$

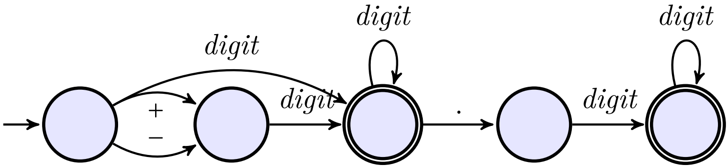


Signed natural numbers: non-deterministic



Fractional numbers

$$frac = signednat(\text{"."}nat)? \tag{9}$$

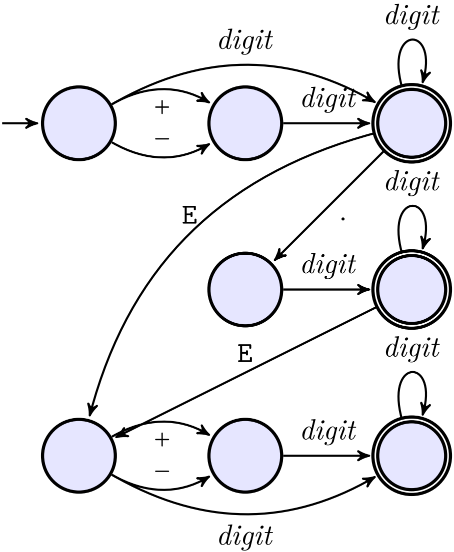


Floats

$$
\begin{aligned}
digit &= [0-9] \\
nat &= digit^{+} \\
signednat &= (+ \mid -)nat \mid nat \\
frac &= signednat(\text{"."}nat)? \\
float &= frac(\texttt{E}\ signednat)?
\end{aligned} \qquad (10)
$$

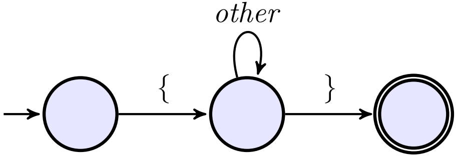
- Note: no (explicit) recursion in the definitions
- note also the treatment of *digit* in the automata.

DFA for floats

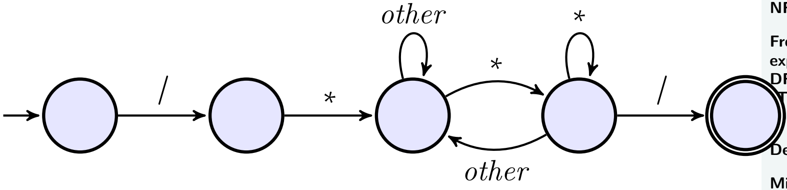


DFAs for comments

Pascal-style



C, C^{++}, Java



Section

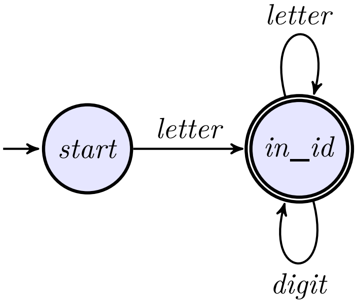
Implementation of DFA

Chapter 1 “Scanning”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Example: identifiers

Regular expression

$$identifier = letter(letter \mid digit)^* \qquad (6)$$

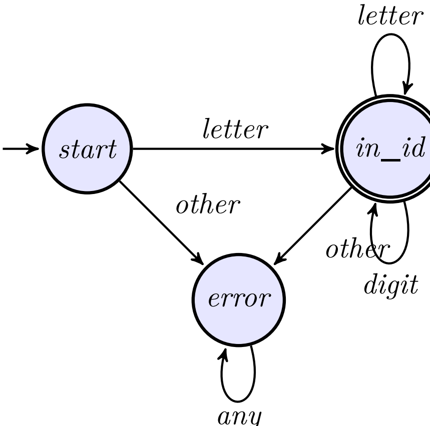


- transition *function*/relation δ *not* completely defined (= *partial* function)

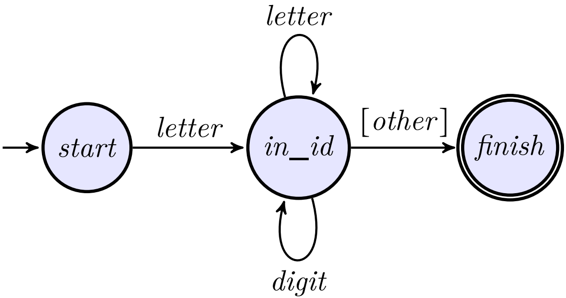
Example: identifiers

Regular expression

$$identifier = letter(letter \mid digit)^* \tag{6}$$



Implementation of DFA (1)



Implementation of DFA (1): “code”

```
{ starting state }

if  the next character is a letter
then
  advance the input;
  { now in state 2 }
  while the next character is a letter or digit
  do
     advance the input;
     { stay in state 2 }
  end while;
  { go to state 3, without advancing input }
  accept;
else
  { error or other cases }
end
```

Explicit state representation

```
state := 1  { start }
while state = 1 or 2
do
  case state of
  1:   case input character of
       letter: advance the input;
               state := 2
       else state := .... { error or other };
       end case;
  2: case input character of
      letter,digit: advance the input;
                    state := 2; { actually unessessary }
      else          state := 3;
      end case;
  end case;
end while;
if state = 3 then accept else error;
```

Table representation of a DFA

state \ input char	letter	digit	other
1	2		
2	2	2	3
3			

Better table rep. of the DFA

state \ input char	letter	digit	other	accepting
1	2			no
2	2	2	[3]	no
3				yes

add info for

- accepting or not
- " *non-advancing* " transitions
  - here: 3 can be reached from 2 via such a transition

Table-based implementation

```
state := 1  { start }
ch := next input character;
while not Accept[state] and not error(state)
do

while state = 1 or 2
do
  newstate := T[state,ch];
  {if Advance[state,ch]
   then ch:=next input character};
  state := newstate
end while;
if Accept [state] then accept;
```

Section

NFA

Chapter 1 “Scanning”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Non-deterministic FSA

Definition (NFA (with ϵ transitions))

A non-deterministic finite-state automaton (NFA for short) $\mathcal{A}$ over an alphabet Σ is a tuple $(\Sigma, Q, I, F, \delta)$, where

- Q: finite set of states
- $I \subseteq Q$, $F \subseteq Q$: initial and final states.
- $\delta : Q \times \Sigma \to 2^Q$ transition function

In case, one uses the alphabet $\Sigma + \{\epsilon\}$, one speaks about an NFA with ϵ-transitions.

- in the following: NFA mostly means, allowing ϵ transitions[11]
- ϵ: treated *differently* than the "normal" letters from Σ.
- δ can *equivalently* be interpreted as *relation*: $\delta \subseteq Q \times \Sigma \times Q$ (transition relation labelled by elements from Σ).

[11] It does not matter much anyhow, as we will see.

Language of an NFA

- remember $\mathcal{L}(\mathcal{A})$ (Definition 7 on page 42)
- applying definition directly to $\Sigma + \{\epsilon\}$: accepting words "containing" letters ϵ
- as said: *special* treatment for ϵ-transitions/ϵ-"letters". ϵ rather represents absence of input character/letter.

Definition (Acceptance with ϵ-transitions)

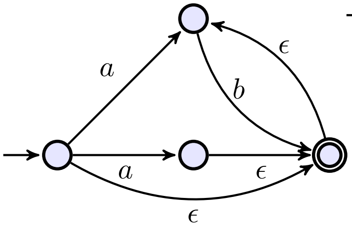
A word w over alphabet Σ is accepted by an NFA with ϵ-transitions, if there exists a word w' which is accepted by the NFA with alphabet $\Sigma + \{\epsilon\}$ according to Definition 7 and where w is w' with all occurrences of ϵ removed.

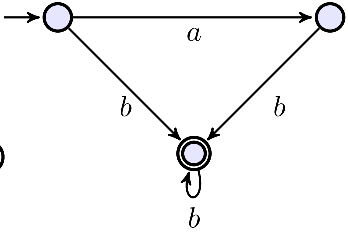
Alternative (but equivalent) intuition

$\mathcal{A}$ reads one character after the other (following its transition relation). If in a state with an outgoing ϵ-transition, $\mathcal{A}$ can move to a corresponding successor state *without* reading an input symbol.

NFA vs. DFA

- *NFA*: often easier (and smaller) to write down, esp. starting from a regular expression
- non-determinism: not *immediately* transferable to an *algo*





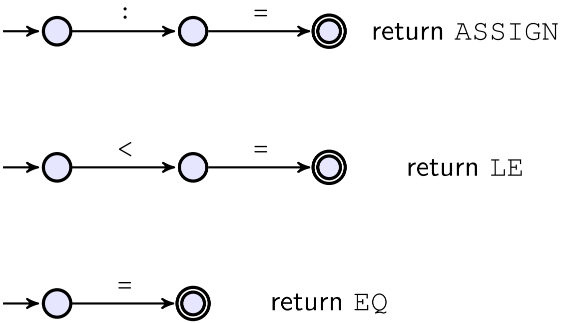
Section

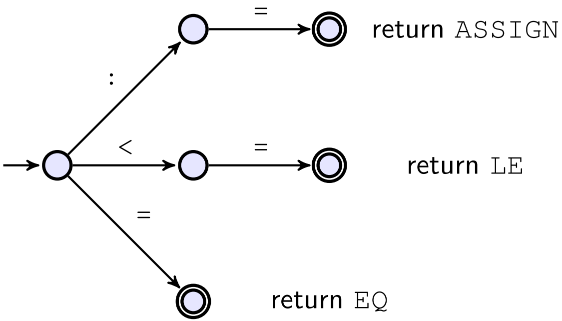
From regular expressions to DFAs (Thompson's construction)

Chapter 1 "Scanning"
Course "Compiler Construction"
Martin Steffen
Spring 2018

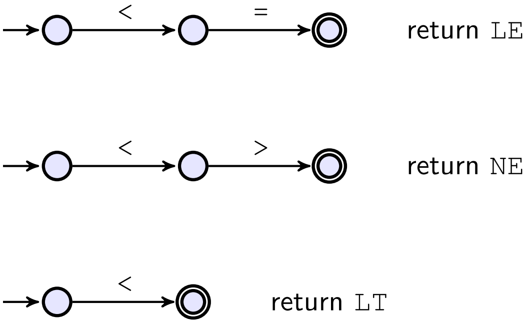
Why non-deterministic FSA?

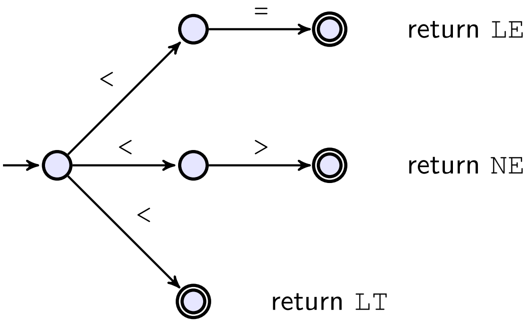
Task: recognize :=, <=, and = as three different tokens:

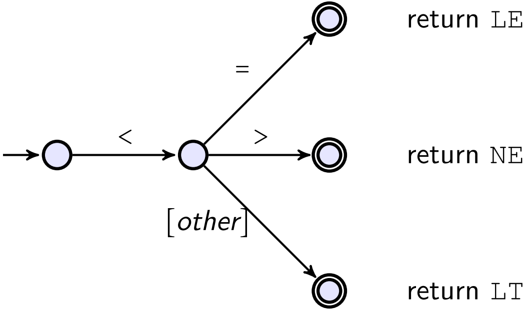



:
=
return ASSIGN
<
=
return LE
=
return EQ

What about the following 3 tokens?




=
<
return LE
<
>
return NE
<
return LT



<
=
>
[other]
return LE
return NE
return LT

Regular expressions → NFA

- needed: a *systematic* translation (= algo, best an efficient one)
- conceptually easiest: translate to NFA (with ϵ-transitions)
  - postpone determinization for a second step
  - (postpone minimization for later, as well)

Compositional construction [5]

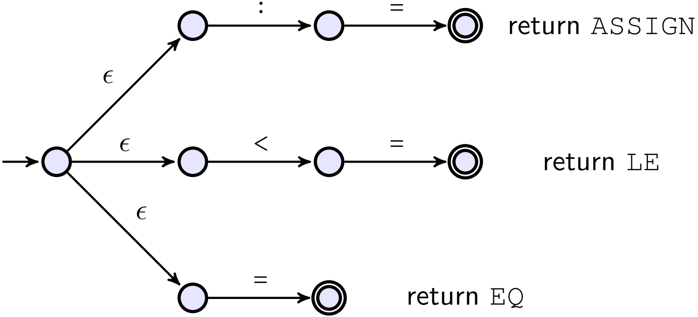
Design goal: The NFA of a compound regular expression is given by taking the NFA of the immediate subexpressions and connecting them appropriately.

- construction slightly[12] simpler, if one uses automata with one start and one accepting state

⇒ ample use of ϵ-transitions

[12] It does not matter much, though.

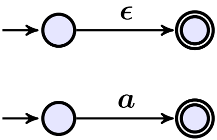
Illustration for ϵ-transitions



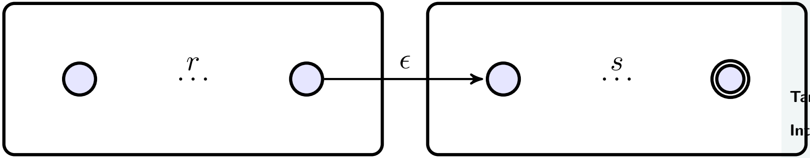
Thompson's construction: basic expressions

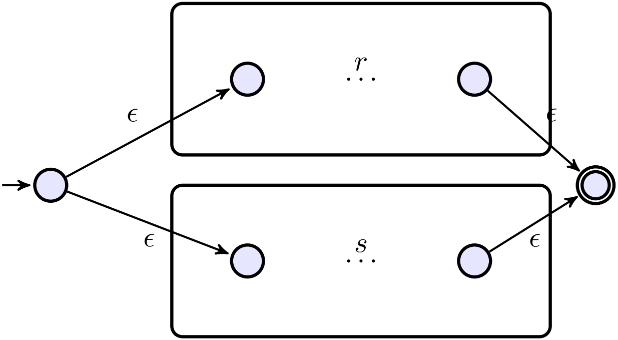
basic regular expressions

basic (= non-composed) regular expressions: ϵ, $\emptyset$, a (for all $a \in \Sigma$)

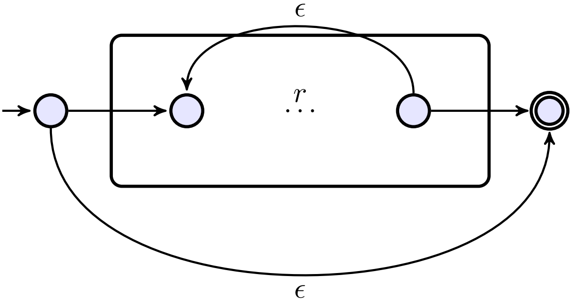


Thompson's construction: compound expressions

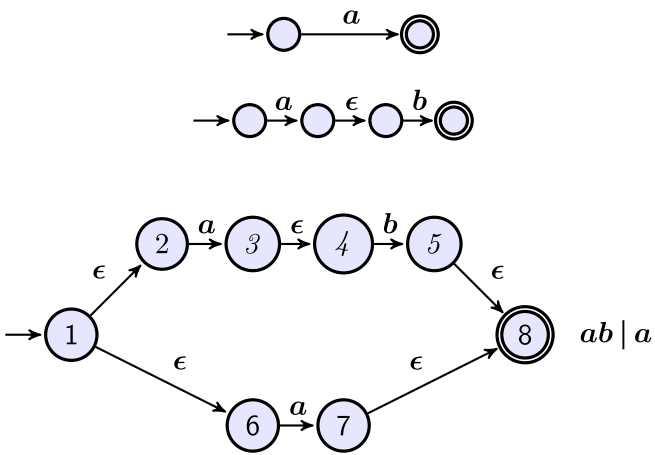




Thompson's construction: compound expressions: iteration



Example: $ab \mid a$



Section

Determinization

Chapter 1 “Scanning”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Determinization: the subset construction

Main idea

- Given a non-det. automaton $\mathcal{A}$. To construct a DFA $\overline{\mathcal{A}}$: instead of *backtracking*: explore all successors "at the same time" ⇒
- each state q' in $\overline{\mathcal{A}}$: represents a *subset* of states from $\mathcal{A}$
- Given a word w: "feeding" that to $\overline{\mathcal{A}}$ leads to *the* state representing *all* states of $\mathcal{A}$ *reachable* via w

- side remark: this construction, known also as *powerset* construction, seems straightforward enough, but: analogous constructions works for some other kinds of automata, as well, but for others, the approach does *not* work.[13]
- origin: [4]

[13] For some forms of automata, non-deterministic versions are strictly more expressive than the deterministic one.

Some notation/definitions

Definition (ϵ-closure, a-successors)

Given a state q, the ϵ-closure of q, written $close_\epsilon(a)$, is the set of states reachable via zero, one, or more ϵ-transitions. We write q_a for the set of states, reachable from q with one a-transition. Both definitions are used analogously for sets of states.

Transformation process: sketch of the algo

Input: NFA $\mathcal{A}$ over a given Σ
Output: DFA $\overline{\mathcal{A}}$

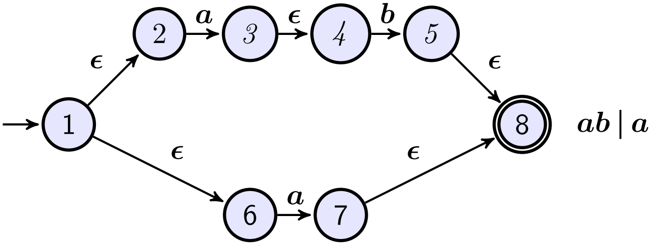
1. the *initial* state: $close_\epsilon(I)$, where I are the initial states of $\overline{\mathcal{A}}$
2. for a state Q' in $\overline{\mathcal{A}}$: the *a-successor* of Q is given by $close_\epsilon(Q_a)$, i.e.,

$$Q \xrightarrow{a} close_\epsilon(Q_a) \tag{11}$$

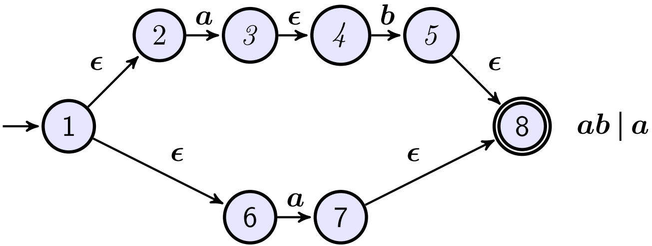
3. repeat step 2 for all states in $\overline{\mathcal{A}}$ and all $a \in \Sigma$, until no more states are being added
4. the *accepting* states in $\overline{\mathcal{A}}$: those containing *at least* one accepting state of $\mathcal{A}$

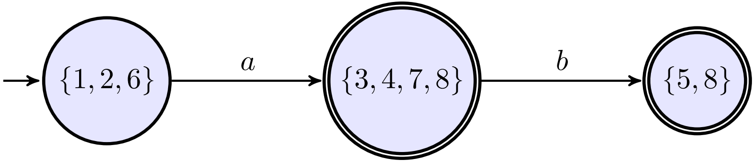
Note: [1]: slightly more "concrete" formulation using a work-list.

Example $ab \mid a$



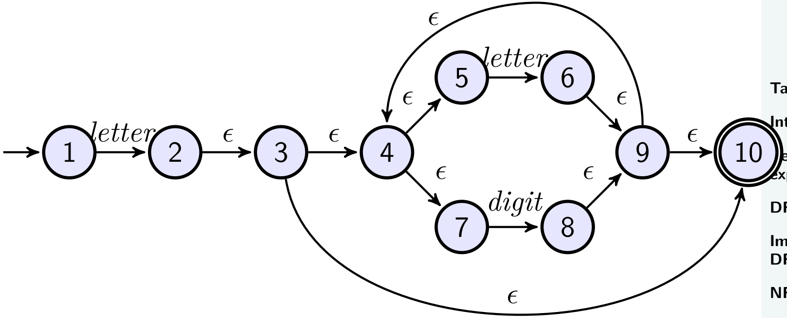
Example $ab \mid a$



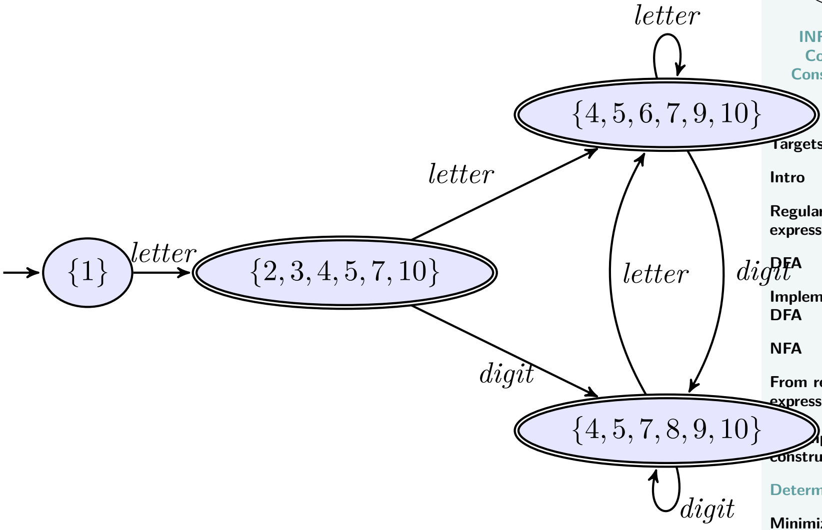


Example: identifiers

Remember: regexpr for identifies from equation (6)



Identifiers: DFA



Section

Minimization

Chapter 1 “Scanning”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Minimization

- automatic construction of DFA (via e.g. Thompson): often many superfluous states
- goal: "combine" states of a DFA without changing the accepted language

Properties of the minimization algo

Canonicity: all DFA for the same language are transformed to the *same* DFA

Minimality: resulting DFA has *minimal* number of states

- "side effects": answers to *equivalence* problems
  - given 2 DFA: do they accept the same language?
  - given 2 regular expressions, do they describe the same language?
- modern version: [2].

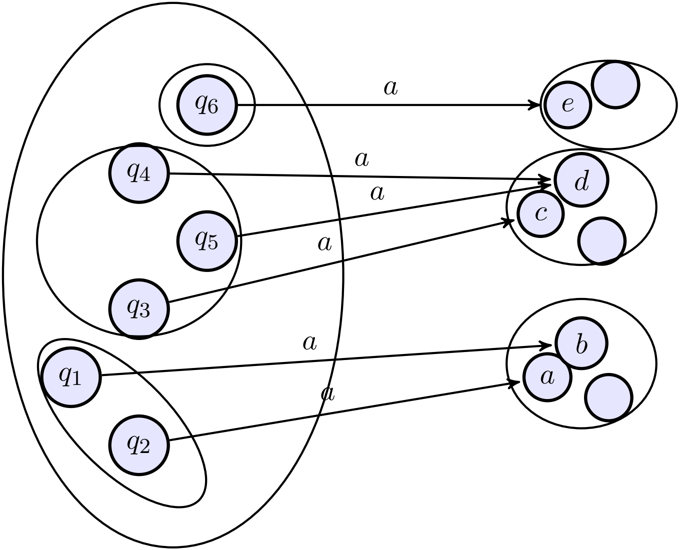
Hopcroft's partition refinement algo for minimization

- starting point: *complete* DFA (i.e., *error*-state possibly needed)
- first idea: *equivalent* states in the given DFA may be *identified*
- equivalent: when used as starting point, accepting the same language
- partition refinement:
  - works "the other way around"
  - instead of collapsing equivalent states:
    - start by "collapsing as much as possible" and then,
    - iteratively, detect *non-equivalent* states, and then *split* a "collapsed" state
    - stop when no violations of "equivalence" are detected
- *partitioning* of a set (of states):
- *worklist*: data structure of to keep non-treated classes, termination if worklist is empty

Partition refinement: a bit more concrete

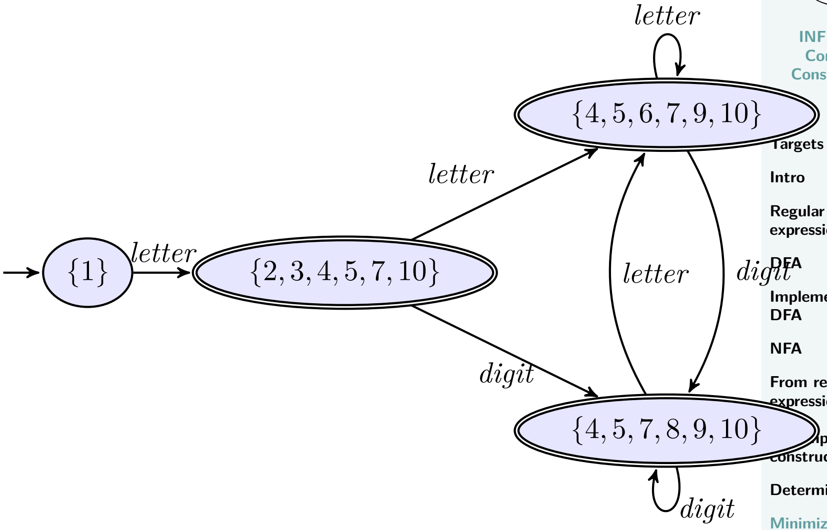
- Initial partitioning: 2 partitions: set containing all *accepting* states F, set containing all *non-accepting* states $Q \backslash F$
- Loop do the following: pick a current equivalence class Q_i and a symbol a
  - if for all $q \in Q_i$, $\delta(q, a)$ is member of the *same* class Q_j $\Rightarrow$ consider Q_i as done (for now)
  - else:
    - split Q_i into $Q_i^1, \ldots Q_i^k$ s.t. the above situation is repaired for each Q_i^l (but don't split more than necessary).
    - be aware: a split may have a "cascading effect": other classes being fine before the split of Q_i need to be reconsidered $\Rightarrow$ *worklist* algo
- stop if the situation stabilizes, i.e., no more split happens (= worklist empty, at latest if back to the original DFA)

Split in partition refinement: basic step

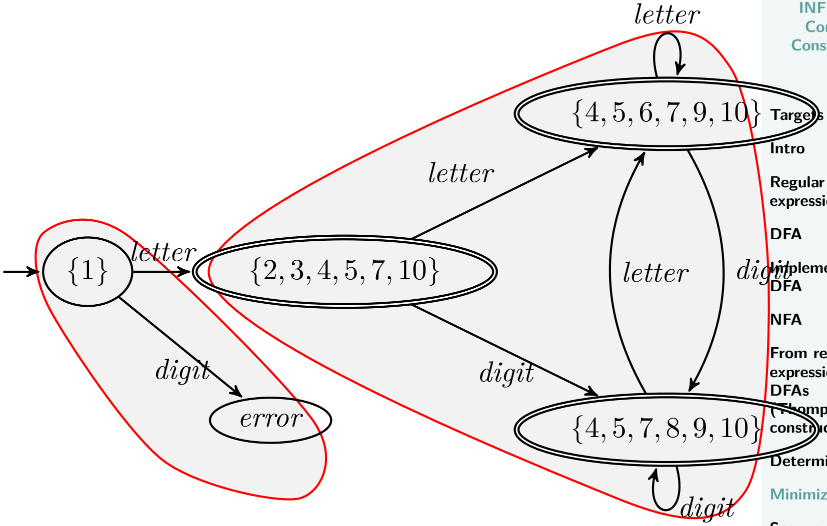


- before the split $\{q_1, q_2, \ldots, q_6\}$
- after the split on a: $\{q_1, q_2\}, \{q_3, q_4, q_5\}, \{q_6\}$

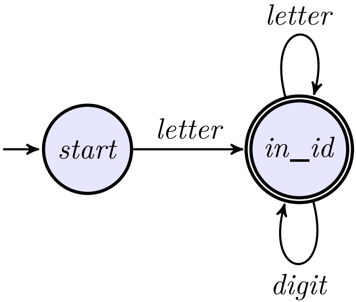
Identifiers: DFA



Completed automaton

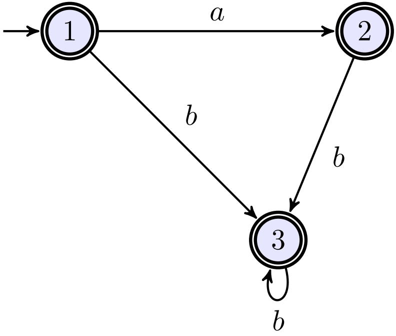


Minimized automaton (error state omitted)



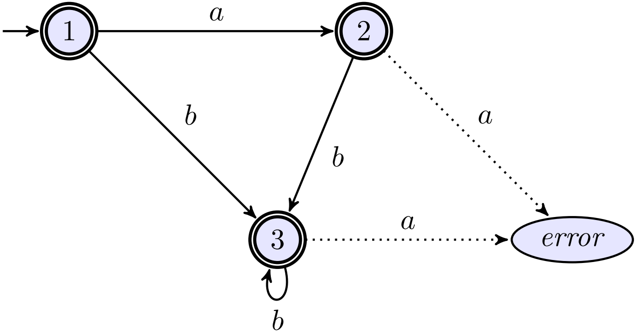
Another example: partition refinement & error state

$$(a \mid \epsilon) b^* \tag{12}$$



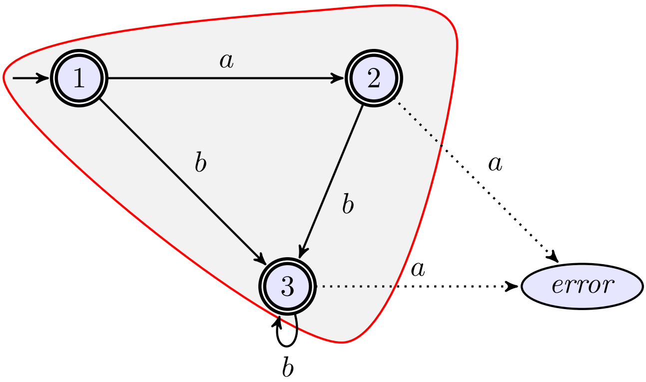
Partition refinement

error state added



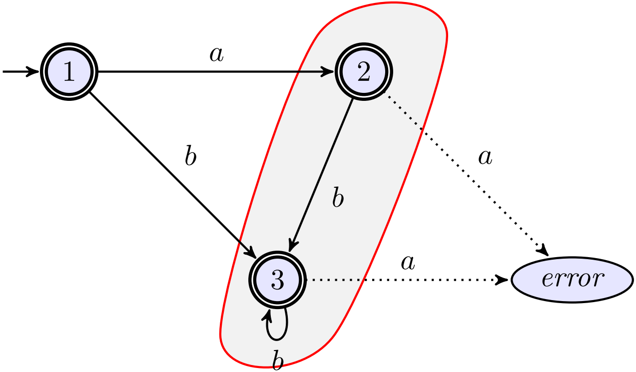
Partition refinement

initial partitioning

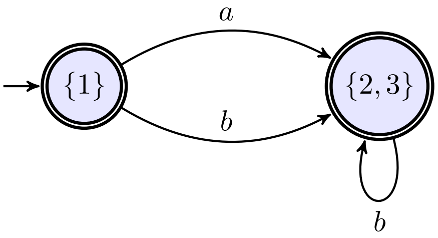


Partition refinement

split after a



End result (error state omitted again)



Section

Scanner implementations and scanner generation tools

Chapter 1 “Scanning”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Tools for generating scanners

- scanners: simple and well-understood part of compiler
- hand-coding possible
- mostly better off with: generated scanner
- standard tools lex / flex (also in combination with *parser* generators, like yacc / bison
- variants exist for many implementing languages
- based on the results of this section

Main idea of (f)lex and similar

- output of lexer/scanner = input for parser
- programmer specifies regular expressions for each token-class and corresponding actions[14] (and whitespace, comments etc.)
- the spec. language offers some conveniences (extended regexpr with priorities, associativities etc) to ease the task
- automatically translated to NFA (e.g. Thompson)
- then made into a deterministic DFA ("subset construction")
- minimized (with a little care to keep the token classes separate)
- implement the DFA (usually with the help of a *table* representation)

[14] Tokens and actions of a parser will be covered later. For example, identifiers and digits as described but the reg. expressions, would end up in two different token classes, where the actual string of characters (also known as *lexeme*) being the value of the token attribute.

Sample flex file (excerpt)

```
DIGIT       [0-9]
ID          [a-z][a-z0-9]*

%%

{DIGIT}+      {
               printf( "An integer: %s (%d)\n", yytext,
                       atoi( yytext ) );
               }

{DIGIT}+"."{DIGIT}*        {
               printf( "A float: %s (%g)\n", yytext,
                       atof( yytext ) );
               }

if|then|begin|end|procedure|function         {
               printf( "A keyword: %s\n", yytext );
               }
```

Section

References

Chapter 1 “Scanning”
Course “Compiler Construction”
Martin Steffen
Spring 2018

References I

Bibliography

[1] Cooper, K. D. and Torczon, L. (2004). *Engineering a Compiler*. Elsevier.

[2] Hopcroft, J. E. (1971). An $n \log n$ algorithm for minimizing the states in a finite automaton. In Kohavi, Z., editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, New York.

[3] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.

[4] Rabin, M. and Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research Developments*, 3:114–125.

[5] Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419.