Course Script

INF 5110: Compiler construction

INF5110/ spring 2018

Martin Steffen

Contents

Part I
Front end

Chapter 2 Scanning

What is it about?

Learning Targets of this Chapter

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 [4, Chapter 2]. The material is pretty canonical anyway.

Contents

2.1 Intro

2.1.1 Scanner section overview

What's a scanner?

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

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

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

2.1.2 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**.

More info

- 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**

2.1.3 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*

[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.

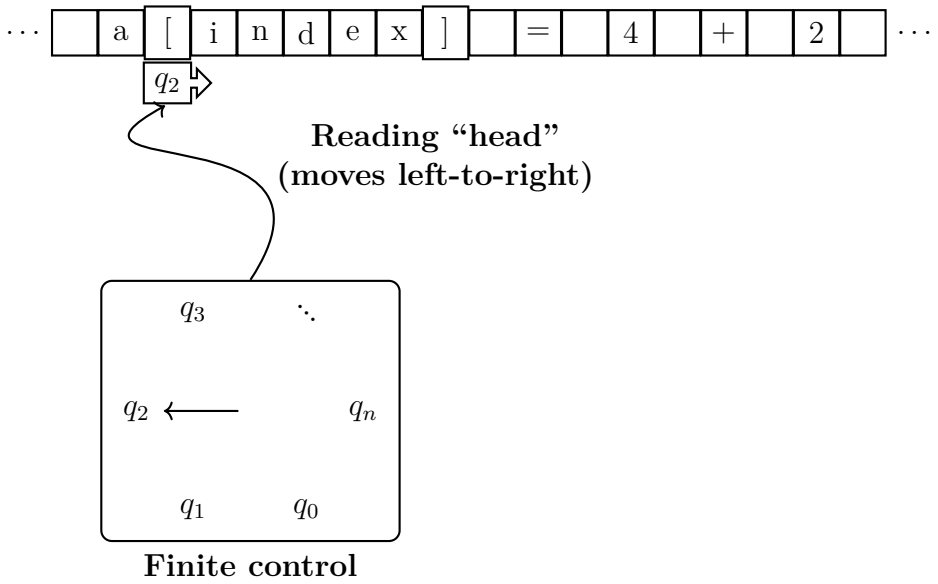
    - to segment into tokens, a scanner typically "jumps over" white spaces and afterwards starts to determine a new token
    - not only "blank" character, also `TAB`, `NEWLINE`, etc.
- lexical rules: often (explicit or implicit) *priorities*
  - *identifier* or *keyword*? ⇒ keyword
  - take the *longest* possible scan that yields a valid token.

2.1.4 "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.

2.1.5 How does scanning roughly work?




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

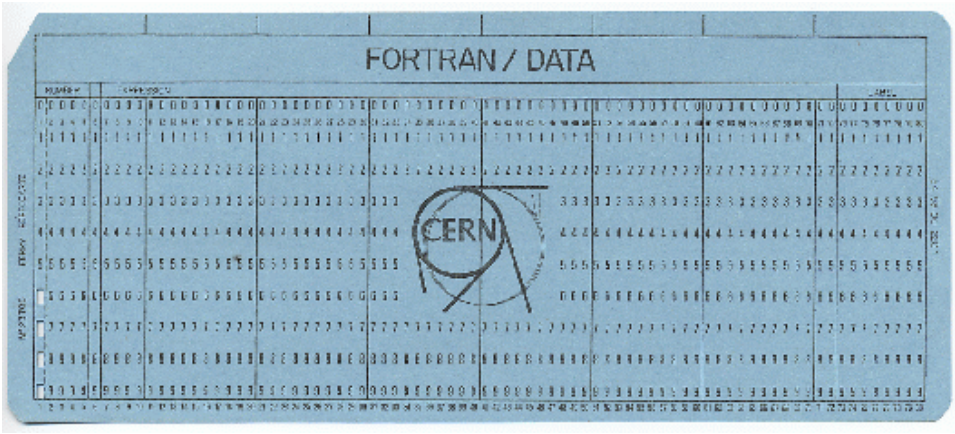
2.1.6 How does scanning roughly work?

- 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]

2.1.7 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 . . .)



2.1.8 (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`

[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. . .

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

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

2.1.9 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":

Ifthen

```
if b then ...
```

Ifthen2

```
if   b    then ...
```

Rest

- 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)

2.1.10 A scanner classifies

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

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

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

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

2.1.11 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

2.1.12 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: every one it its own group	`{ } ( ) [ ] , ; := .` etc.

- 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 `"<="`

2.1.13 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;
```

2.1.14 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.

Prosa specification

A precise prosa specification is not so easy to achieve as one might think. For ASCII source code or input, things are basically under control. But what if dealing with unicode? Checking "legality" of user input to avoid SQL injections or format string attacks is largely done by lexical analysis/scanning. If you "specify" in English: " *Backlash is a control character and forbidden as user input* ", which characters (besides `char 92` in ASCII) in Chinese Unicode represents actually other versions of backslash?

Parser generator

The most famous pair of lexer+parser is called "compiler compiler" (lex/yacc = "yet another compiler compiler") since it generates (or "compiles") an important part of the front end of a compiler, the lexer+parser. Those kind of tools are seldomly called compiler compilers any longer.

2.2 Regular expressions

2.2.1 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

- Above steps are 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]
- Step 2 is actually *not* covered by the classical Reg-expr = DFA = NFA results, it's something extra.

2.2.2 Use of regular expressions

- **regular languages**: fundamental class of "languages"
- **regular expressions**: standard way to describe regular languages
- origin of regular expressions: one starting point is Kleene [3] but there had been earlier works outside "computer science"
- 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)

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

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

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

Remarks

Kleene was a famous mathematician and influence on theoretical computer science. Funnily enough, regular languages came up in the context of neuro/brain science. See the following for the origin of the terminology. Perhaps in the early years, people liked to draw connections between between biology and machines and used metaphors like "electronic brain" etc.

2.2.3 Alphabets and languages

Definition 2.2.1 (Alphabet Σ)**.** Finite set of elements called "letters" or "symbols" or "characters".

Definition 2.2.2 (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.

Remark: Symbols in a symbol table

In a certain way, symbols in a symbol table can be seen similar to symbols in the way we are handled by automata or regular expressions now. They are simply "atomic" (not further dividable) members of what one calls an alphabet. On the other hand, in practical terms inside a compiler, the symbols here in the scanner chapter live on a different level compared to symbols encountered in later sections, for instance when discussing symbol tables). Typically here, they are *characters*, i.e., the alphabet is a so-called character set, like for instance, ASCII. The lexer, as stated, segments and classifies the sequence of characters and hands over the result of that process to the parser. The results is a sequence of *tokens*, which is what the parser has to deal with

later. It's on that parser-level, that the pieces (notably the identifiers) can be treated as atomic pieces of some language, and what is known as the symbol table typcially operates on symbols at that level, not at the level of individual characters.

2.2.4 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 $\varnothing$) the empty set
  2. $\{a, b, ab\}$: language with 3 finite words
  3. $\{\epsilon\}$ ($\neq \varnothing$)
  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. $\{ab, bbab, aaaaa, bbabbabab, aabb, \ldots\}$: ?????

Remarks

Remark 1 (Words and strings)**.** *In terms of a real implementation: often, the letters are of type character (like type* `char` *or* `char32` *...)* words *then are "sequences" (say arrays) of characters, which* may *or* may not *be identical to elements of type* `string`*, depending on the language for implementing the compiler. In a more conceptual part like here we do not write words in "string notation" (like "ab"), since we are dealing abstractly with* sequences *of letters, which, as said, may not actually be strings in the implementation. Also in the more conceptual parts, it's often good enough when handling alphabets with 2 letters, only, like* $\Sigma = \{a, b\}$ *(with one letter, it gets unrealistically trivial and results may not carry over to the many-letter alphabets). After all, computers are using 2 bits only, as well ....*

Finite and infinite words

There are important applications dealing with infinite words, as well, or also even infinite alphabets. For traditional scanners, one mostly is happy with finite Σ 's and especially sees no use in scanning infinite "words". Of course, some charactersets, while not actually infinite, are large (like Unicode or UTF-8)

Sample alphabets

Often we operate for illustration on alphabets of size 2, like $\{a, b\}$. One-letter alphabets are uninteresting, let alone 0-letter alphabets. 3 letter alphabets may not add much as far as "theoretical" questions are concerned. That may be compared with the fact that computers ultimately operate in words over two different "bits" .

2.2.5 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 (what was meant in the last example?)
- 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)

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

- small metaphor

$$2.727272727\ldots \qquad 3.1415926\ldots \tag{2.1}$$

Remarks

Remark 2 (Programming languages as "languages")**.** *Well, Java etc., seen syntactically as all possible strings that can be compiled to well-formed byte-code, also is a* language *in the sense we are currently discussing, namely a a set of words over unicode. But when speaking of the "Java-language" or other programming languages, one typically has also other aspects in mind (like what a program does when it is executed), which is not covered by thinking of Java as an infinite set of strings.*

Remark 3 (Rational and irrational numbes)**.** *The illustration on the slides with the two numbers is partly meant as that: an illustration drawn from a field you may know. The first number from equation (2.1) is a* rational number*. It corresponds to the fraction*

$$\frac{30}{11}\ . \tag{2.2}$$

That fraction is actually an acceptable finite representation for the "endless" notation 2.72727272... *using "..." As one may remember, it may pass as a decent definition of rational numbers that they are exactly those which can be represented finitely as fractions of two integers, like the one from equation (2.2). We may also remember that it is characteristic for the "endless" notation as the one from equation (2.1), that for rational numbers, it's* periodic. *Some may have learnt the notation*

$$2.\overline{72} \tag{2.3}$$

for finitely representing *numbers with a* periodic *digit expansion (which are exactly the rationals). The second number, of course, is* π*, one of the most famous numbers which do* not *belong to the rationals, but to the "rest" of the reals which are not rational (and hence called irrational). Thus it's one example of a "number" which cannot represented by a fraction, resp. in the periodic way as in (2.3).*

Well, fractions may not work out for π *(and other irrationals), but still, one may ask, whether* π *can otherwise be represented finitely. That, however, depends on what actually one accepts as a "finite representation". If one accepts a finite description that describes how to* construct *ever closer approximations to* π*, then there is a finite representation of* π*. That construction basically is very old (Archimedes), it corresponds to the limits one learns in analysis, and there are computer algorithms, that spit out digits of* π *as long as you want (of course they can spit them out* all *only if you had infinite time). But the* code *of the algo who does that is finite.*

The bottom line is: it's possible to describe infinite "constructions" in a finite manner, but what exactly *can be captured depends on what precisely is allowed in the description formalism. If only fractions of natural numbers are allowed, one can describe the rationals but not more.*

A final word on the analogy to regular languages. The set of rationals (in, let's say, decimal notation) can be seen as language over the alphabet $\{0, 1, \ldots, 9\ .\}$*, i.e., the decimals and the "decimal point". It's however, a language containing* infinite *words, such as* $2.727272727\ldots$*. The syntax* $2.\overline{72}$ *is a* finite *expression but denotes the mentioned infinite word (which is a decimal representation of a rational number). Thus, coming back to the regular languages resp. regular expressions,* $2.\overline{72}$ *is similar to the* Kleene-*star, but* not the same. *If we write* $2.(72)^*$*, we mean the* language *of* finite words

$$\{2, 2.72, 2.727272, \ldots\}\ .$$

In the same way as one may conveniently define *rational number (when represented in the alphabet of the decimals) as those which can be written using periodic expressions (using for instance* $\overline{\text{overline}}$*),* regular *languages over an alphabet are simply those sets of finite words that can be written by* regular

expressions *(see later). Actually, there are deeper connections between regular languages and rational numbers, but it's not the topic of compiler constructions. Suffice to say that it's not a coincidence that* regular *languages are also called* rational *languages (but not in this course).*

2.2.6 Regular expressions

Definition 2.2.3 (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 $\boldsymbol{\varnothing}$
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, $\mid$

Regular expressions

In [1], $\varnothing$ is not part of the regular expressions. For completeness sake it's included here even if it does not play a practically important role.

In other textbooks, also the notation + instead of $\mid$ for "alternative" or "choice" is a known convention. The $\mid$ seems more popular in texts concentrating on *grammars*. Later, we will encounter *context-free* grammars (which can be understood as a generalization of regular expressions) and the $\mid$-symbol is consistent with the notation of alternatives in the definition of rules or productions in such grammars. One motivation for using + elsewhere is that one might wish to express "parallel" composition of languages, and a conventional symbol for parallel is $\|$. We will not encounter parallel composition of languages in this course. Also, regular expressions using lot of parentheses and $\mid$ seems slightly less readable for humans than using +.

Regular expressions are a language in itself, so they have a syntax and a semantics. One could write a lexer (and parser) to parse a regular language. Obviously, tools like parser generators *do* have such a lexer/parser, because their input language are regular expression (and context free grammars, besides syntax to describe further things). One can see regular languages as a domain-specific language for tools like (f)lex (and other purposes).

2.2.7 A "grammatical" definition

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

$$\begin{array}{lcl} r & \to & \mathbf{a} \\ r & \to & \boldsymbol{\epsilon} \\ r & \to & \varnothing \\ r & \to & r \mid r \\ r & \to & r\, r \\ r & \to & r^* \end{array} \tag{2.4}$$

2.2.8 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:

Grammar

$$\begin{array}{lcl} R & \to & \mathbf{a} \\ R & \to & \boldsymbol{\epsilon} \\ R & \to & \varnothing \\ R & \to & R \mid R \\ R & \to & R\, R \\ R & \to & R^* \end{array} \tag{2.5}$$

2.2.9 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:[10] notation to describe regular expression

[10] To be careful, we will (later) distinguish between context-free languages on the one hand and notations to denote context-free languages on the other, in the same manner that we *now* don't want to confuse regular languages as concept from particular notations (specifically, regular expressions) to write them down.

- for now: carefully use *notational convention* for precision

2.2.10 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 $\varnothing$
  - $\mathbf{|}$ 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 ...)

Remarks

Remark 4 (Regular expression syntax)**.** *Later there will be a number of examples using regular expressions. There is a slight "ambiguity" about the way regular expressions are described (in this slides, and elsewhere). It may remain unnoticed (so it's unclear if one should point it out). On the other had, the lecture is, among other things, about scanning and parsing of syntax, therefore it may be a good idea to reflect on the* syntax *of regular expressions themselves.*

In the examples shown later, we will use regular expressions using parentheses, like for instance in $\boldsymbol{b(ab)^*}$*. One question is: are the parentheses* **(** *and* **)** *part of the definition of regular expressions or not? That depends a bit. In the presentation here typically one would not care, one tells the readers that parentheses will be used for disambiguation, and leaves it at that (in the same way one would not tell the reader it's fine to use "space" between different expressions (like* $a \mid b$ *is the same expression as* $a \quad \mid \quad b$*). Another way of saying that is that textbooks, intended for human readers, give the definition of regular expressions as* abstract syntax *as opposed to* concrete syntax*. Those 2 concepts will play a prominent role later in the grammar and parsing sections and may be more clear then. Anyway, it's thereby assumed that the reader can interpret parentheses as grouping mechanism, as is common elsewhere as well and they are left out from the definition not to clutter it.*

Of course, computers *and programs (i.e., in particular scanners or lexers), are not as good as humans to be educated in "commonly understood" conventions (such as "paretheses are not really part of the regular expressions but can be added for disambiguation".* Abstract *syntax corresponds to describing the*

output *of a parser (which are* abstract syntax trees*). In that view, regular expressions (as all notation represented by abstract syntax) denote* trees*. Since trees in texts are more difficult (and space-consuming) to write, one simply use the usual* linear notation *like the* $\boldsymbol{b(ab)^*}$ *from above, with parentheses and "conventions" like precedences, to disambiguate the expression. Note that a tree representation represents the grouping of sub-expressions in its structure, so for grouping purposes, parentheses are not needed in abstract syntax.*

Of course, if one wants to implement a lexer or to use one of the available ones, one has to deal with the particular concrete *syntax of the particular scanner. There, of course, characters like* $'($ *and* $')'$ *(or tokens like* `LPAREN` *or* `RPAREN`*) might occur.*

Using concepts which will be discussed in more depth later, one may say: whether paretheses are considered as part of the syntax of regular expressions or not depends on the fact whether the definition is wished to be understood as describing concrete syntax *trees or /abstract syntax trees!*

See also Remark 5 later, which discusses further "ambiguities" in this context.

2.2.11 Same again once more

$$\begin{array}{rcll} R & \rightarrow & \mathbf{a} \mid \boldsymbol{\epsilon} \mid \boldsymbol{\varnothing} & \text{basic reg. expr.} \\ & \mid & R \,|\, R \mid R\,R \mid R^* \mid \mathbf{(}R\mathbf{)} & \text{compound reg. expr.} \end{array} \tag{2.6}$$

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

2.2.12 Semantics (meaning) of regular expressions

Definition 2.2.4 (Regular expression)**.** Given an alphabet Σ. The meaning of a regexp r (written $\mathcal{L}(r)$) over Σ is given by equation (2.7).

$$\begin{array}{rcll} \mathcal{L}(\boldsymbol{\varnothing}) & = & \{\} & \text{empty language} \\ \mathcal{L}(\boldsymbol{\epsilon}) & = & \{\epsilon\} & \text{empty word} \\ \mathcal{L}(\boldsymbol{a}) & = & \{a\} & \text{single "letter" from } \Sigma \\ \mathcal{L}(r \mid s) & = & \mathcal{L}(r) \cup \mathcal{L}(s) & \text{alternative} \\ \mathcal{L}(r^*) & = & \mathcal{L}(r)^* & \text{iteration} \end{array} \tag{2.7}$$

- conventional *precedences*: $*$, concatenation, $|$.

- Note: left of "=": reg-expr *syntax*, right of "=": semantics/meaning/math [11]

2.2.13 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	

2.2.14 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 $notb \triangleq a \mid c$

Remarks

Remark 5 (Regular expressions, disambiguation, and associativity)**.** *Note that in the equations in the example, we silently allowed ourselves some "sloppyness" (at least for the nitpicking mind). The slight ambiguity depends on how we* exactly *interpret definitions of regular expressions. Remember also Remark 4 on page 17, discussing the (non-)status of parentheses in regular expressions. If we think of Definition 2.2.3 on page 14 as describing abstract syntax and a concrete regular expression as representing an abstract syntax*

[11] Sometimes confusingly "the same" notation.

tree, then the constructor | *for alternatives is a* binary *constructor. Thus, the regular expression*

$$a \mid c \mid ba \mid bc \tag{2.8}$$

which occurs in the previous example is ambiguous. *What is meant would be one of the following*

$$a \mid (c \mid (ba \mid bc)) \tag{2.9}$$
$$(a \mid c) \mid (ba \mid bc) \tag{2.10}$$
$$((a \mid c) \mid ba) \mid bc \ , \tag{2.11}$$

corresponding to 3 different trees, where occurences of | *are inner nodes with two children each, i.e., sub-trees representing subexpressions. In textbooks, one generally does not want to be bothered by writing all the parentheses. There are typically two ways to disambiguate the situation. One is to state (in the text) that the operator, in this case* |, associates to the left *(alternatively it* associates to the right*). That would mean that the "sloppy" expression without parentheses is meant to represent either (2.9) or (2.11), but not (2.10). If one really wants (2.10), one needs to indicate that using parentheses. Another way of finding an excuse for the sloppyness is to realize that it (in the context of regular expressions)* does not matter*, which of the three trees (2.9) – (2.11) is actually meant. This is specific for the setting here, where the symbol* | *is* semantically *represented by* set union $\cup$ *(cf. Definition 2.2.4 on page 18) which* is *an associative operation on sets. Note that, in principle, one may choose the first option —disambiguation via fixing an associativity— also in situations, where the operator is* not *semantically associative. As illustration, use the '–' symbol with the usal intended meaning of "subtraction" or "one number minus another". Obviously, the expression*

$$5 - 3 - 1 \tag{2.12}$$

now can be interpreted in two semantically different ways, one representing the result 1*, and the other* 3*. As said, one* could *introduce the convention (for instance) that the binary minus-operator associates to the left. In this case, (2.12) represents* $(5 - 3) - 1$*.*

Whether or not in such a situation one wants symbols to be associative or not is a judgement call (a matter of language pragmatics). On the one hand, disambiguating may make expressions more readable by allowing to omit parenthesis or other syntactic markers which may make the expression or program look cumbersome. On the other, the "light-weight" and "programmer-friendly" syntax may trick the unexpecting programmer into misconceptions about what the program means, if unaware of the rules of assorciativity (and other priorities). Disambiguation via associativity rules and priorities is therefore a double-edged sword and should be used carefully. A situation where most would agree associativity is useful and completely unproblematic is the one illustrated for |

in regular expression: it does not matter anyhow semantically. Decisions concerning using a-priori ambiguous syntax plus rules how to disambiguate them (or forbid them, or warn the user) occur in many situations in the scannning and parsing phases of a compiler.

Now, the discussion concerning the "ambiguity" of the expression $(a \mid c \mid ba \mid bc)$ *from equation (2.8) concentrated on the* $\mid$*-construct. A similar discussion could obviously be made concerning concatenation (which actually here is not represented by a readable concatenation operator, but just by juxtaposition (=writing expressions side by side)). In the concrete example from (2.8), no ambiguity wrt. concatenation actually occurs, since expressions like* ba *are not ambiguous, but for longer sequences of concatenation like* abc*, the question of whether it means* $a(bc)$ *or* $a(bc)$ *arises (and again, it's not critical, since concatenation is semantically associative).*

Note also that one might think that the expression suffering from an ambiguity concerning combinations of operators, for instance, combinations of $\mid$ *and concatenation. For instance, one may wonder if* $\boldsymbol{ba \mid bc}$ *could be interpreted as* $(ba) \mid (bc)$ *and* $b(a \mid (bc))$ *and* $b(a \mid b)c$*. However, on page 18, we stated* precedences *or priorities 2.2.4 on page 18, stating that concatenation has a* higher *precedence over* $\mid$*, meaning that the correct interpretation is* $(ba) \mid (bc)$*. In a text-book the interpretation is "suggested" to the reader by the typesetting* $ba \mid bc$ *(and the notation it would be slightly less "helpful" if one would write* $ba|bc$*... and what about the programmer's version* `a␣b|a␣c`*?). The situation with precedence is one where difference precedences lead to semantically different interpretations. Even if there's a danger therefore that programmers/readers mis-interpret the real meaning (being unaware of precedences or mixing them up in their head), using precedences in the case of regular expressions certainly is helpful, The alternative of being forced to write, for instance*

$$((a(b(cd))) \mid (b(a(ad)))) \quad \textit{for} \quad abcd \mid baad$$

is not even appealing to hard-core Lisp-programmers (but who knows ...).

A final note: all this discussion about the status of parentheses or left or right assocativity in the interpretation of (for instance mathematical) notation is mostly is over-the-top for most mathematics or other fields where some kind of formal notations or languages are used. There, notation is introduced, perhaps accompanied by sentences like "parentheses or similar will be used when helpful" or "we will allow ourselves to omit parentheses if no confusion may arise", which means, the educated reader is expected to figure it out. Typically, thus, one glosses over too detailed syntactic conventions to proceed to the more interesting and challenging aspects of the subject matter. In such fields one is furthermore sometimes so used to notational traditions ("multiplication binds stronger than addition"), perhaps established since decades or even centuries, that one does not even think about them consciously. For scanner and parser designers, the situation is different; they are requested to come up with the

notational (lexical and syntactical) conventions of perhaps a new *language, specify them precisely and implement them efficiently. Not only that: at the same time, one aims at a good balance between expliciteness ("Let's just force the programmer to write all the parentheses and grouping explicitly, then he will get less misconceptions of what the program means (and the lexer/parser will be easy to write for me...)") and economy in syntax, leaving many conventions, priorities, etc. implicit without confusing the target programmer.* □

2.2.15 Additional "user-friendly" notations

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

Special notations for *sets* of letters:

$$\begin{aligned} [0-9] &\quad \text{range (for ordered alphabets)} \\ \sim a &\quad \text{not } a \text{ (everything except } a) \\ . &\quad \text{all of } \Sigma \end{aligned}$$

naming regular expressions ("regular definitions")

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

2.3 DFA

2.3.1 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.[12]

Remark 6 (Finite states)**.** *The distinguishing feature of FSA (as opposed to more powerful automata models such as push-down automata, or Turing-machines), is that they have "* finitely many states *". That sounds clear enough at first sight. But one has too be a bit more careful. First of all, the set of states of the automaton, here called Q, is finite and fixed for a given automaton, all right. But actually, the same is true for pushdown automata and Turing machines! The trick is: if we look at the illustration of the finite-state automaton earlier, where the automaton had a* head*. The picture corresponds to an* accepting *use of an automaton, namely one that is fed by letters on the tape, moving internally from one state to another, as controlled by the different letters (and the automaton's internal "logic", i.e., transitions). Compared to the full power of Turing machines, there are* two *restrictions, things that a finite state automaton cannot do*

- *it moves on one direction only (left-to-right)*
- *it is* read-only.

All non-finite state machines have some additional *memory they can use (besides $q_0, \ldots, q_n \in Q$). Push-down automata for example have additionally a stack, a Turing machine is allowed to* write *freely (= moving not only to the right, but back to the left as well) on the tape, thus using it as external memory.*

2.3.2 FSA

Definition 2.3.1 (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

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

[12] Historically, design of electronic circuitry (not yet chip-based, though) was one of the early very important applications of finite-state machines.

2.3.3 FSA as scanning machine?

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

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

Non-determinism

Sure, one could try **backtracking**, but, trust us, you don't want that in a scanner. And even if you think it's worth a shot: how do you scan a program directly from magnetic tape, as done in the bad old days? Magnetic tapes can be rewound, of course, but winding them back and forth all the time destroys hardware quickly. How should one scan network traffic, packets etc. on the fly? The network definitely cannot be rewound. Of course, buffering the traffic would be an option and doing then backtracking using the buffered traffic, but maybe the packet-scanning-and-filtering should be done in hardware/firmware, to keep up with today's enormous traffic bandwith. Hardware-only solutions have no dynamic memory, and therefore actually *are* ultimately finite-state machine with no extra memory.

2.3.4 DFA: deterministic automata

Definition 2.3.2 (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[13] (“complete”)

2.3.5 Meaning of an FSA

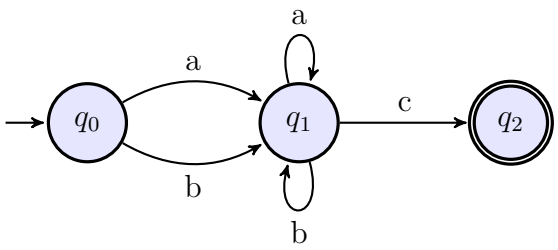
The intended **meaning** of an FSA over an alphabet Σ is the set of all the finite words, the automaton **accepts**.

Definition 2.3.3 (Accepting words and language of an automaton)**.** A word $c_1 c_2 \dots c_n$ with $c_i \in \Sigma$ is *accepted* by automaton $\mathcal{A}$ over Σ, if there exists states $q_0, q_2, \dots, q_n$ from Q such that

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

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.

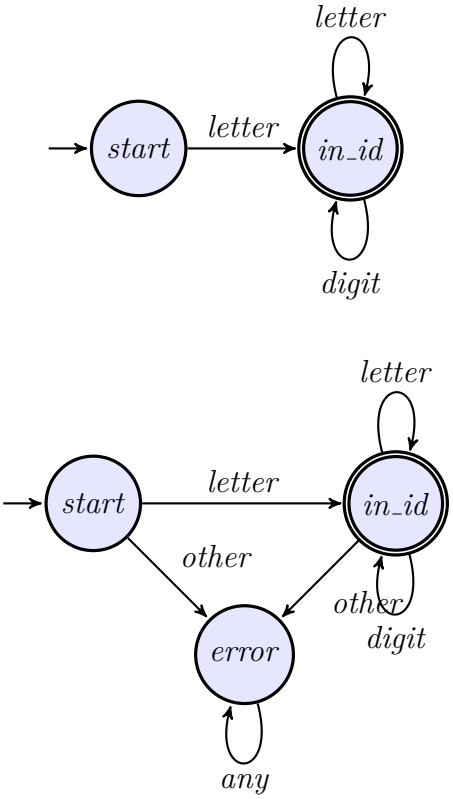
2.3.6 FSA example



2.3.7 Example: identifiers

$$\mathit{identifier} = \mathit{letter}(\mathit{letter} \mid \mathit{digit})^* \tag{2.13}$$

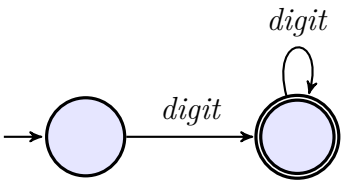
[13] That means, for each pair q, a from $Q \times \Sigma$, $\delta(q, a)$ is defined. Some people call an automaton where δ is not a left-total but a determinstic relation (or, equivalently, the function δ is not total, but partial) still a deterministic automaton. In that terminology, the DFA as defined here would be determinstic *and* total.



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

2.3.8 Automata for numbers: natural numbers

$$\begin{aligned} digit &= [0-9] \\ nat &= digit^{+} \end{aligned} \tag{2.14}$$

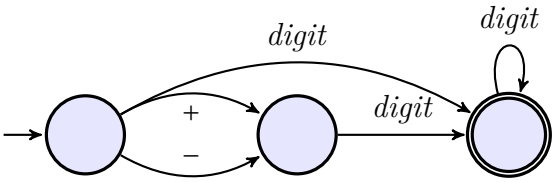


Remarks

One might say, it's not really the natural numbers, it's about a decimal *notation* of natural numbers (as opposed to other notations, for example Roman numeral notation). Note also that initial zeroes are allowed here. It would be easy to disallow that.

2.3.9 Signed natural numbers

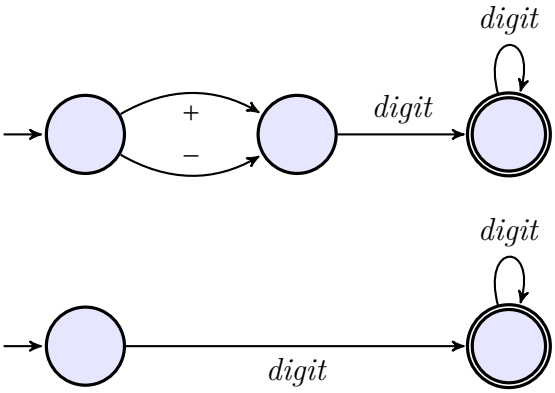
$$signednat \quad = \quad (+ \mid -)nat \mid nat \tag{2.15}$$



Remarks

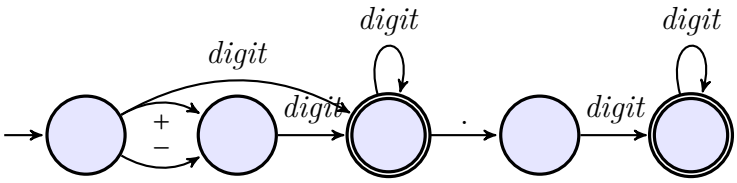
Again, the automaton is *deterministic*. It's easy enough to come up with this automaton, but the *non-deterministic* one is probably more straightforward to come by with. Basically, one informally does two "constructions", the "alternative" which is simply writing "two automata", i.e., one automaton which consists of the union of the two automata, basically. In this example, it therefore has two initial states (which is disallowed obviously for deterministic automata). Another implicit construction is the *"sequential composition"*.

2.3.10 Signed natural numbers: non-deterministic



2.3.11 Fractional numbers

$$frac \quad = \quad signednat(\text{"."}nat)? \tag{2.16}$$

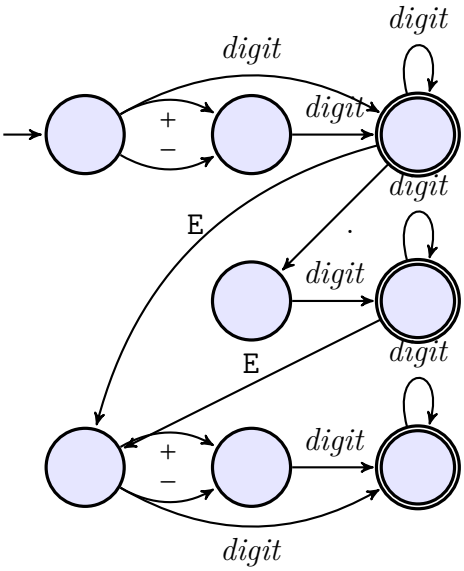


2.3.12 Floats

$$\begin{aligned} digit &= [0-9] \\ nat &= digit^{+} \\ signednat &= (+ \mid -)nat \mid nat \\ frac &= signednat(\text{"."}nat)? \\ float &= frac(\texttt{E}\ signednat)? \end{aligned} \tag{2.17}$$

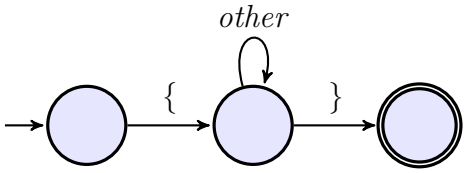
- Note: no (explicit) recursion in the definitions
- note also the treatment of *digit* in the automata.

2.3.13 DFA for floats

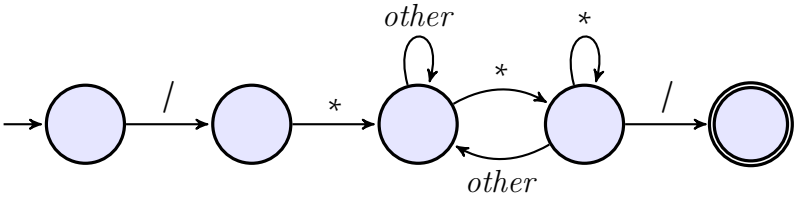


2.3.14 DFAs for comments

Pascal-style



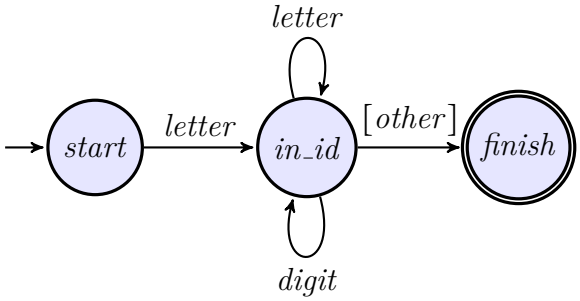
C, C++, Java



2.4 Implementation of DFA

2.4.1 Repeat frame: Example: identifiers

2.4.2 Implementation of DFA (1)



Unlike the previous automaton, this one is *deterministic*, but it's *not* total. The transition function is only *partial*. The "missing" transitions are often not shown (the make the pictures more compact). It is then implicitly assumed, that encountering a character not covered by a transition leads to some extra "error" state (which also is not shown).

The [] around the transition *other* at the end means that the scanner does *not move forward* in the input there (but the automaton proceeds to the accepting state). Note also that the accepting state has changed. Longest prefix.

2.4.3 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
```

2.4.4 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;
```

2.4.5 Table representation of a DFA

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

2.4.6 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

2.4.7 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;
```

2.5 NFA

2.5.1 Non-deterministic FSA

Definition 2.5.1 (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[14]
- 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).

Finite state machines

Remark 7 (Terminology (finite state automata))**.** *There are slight variations in the definition of (deterministic resp. non-deterministic) finite-state automata. For instance, some definitions for non-deterministic automata might not use ϵ-transitions, i.e., defined over Σ, not over $\Sigma + \{\epsilon\}$. Another word for FSAs are finite-state machines. Chapter 2 in [4] builds in ϵ-transitions into the definition of NFA, whereas in Definition 2.5.1, we mention that the NFA is not just non-deterministic, but "also" allows those specific transitions. Of course, ϵ-transitions lead to non-determinism as well, in that they correspond to "spontaneous" transitions, not triggered and determined by input. Thus, in the presence of ϵ-transition, and starting at a given state, a fixed input may* not determine *in which state the automaton ends up in.*

Deterministic or non-deterministic FSA (and many, many variations and extensions thereof) are widely used, *not only for scanning. When discussing scanning, ϵ-transitions come in handy, when translating regular expressions to FSA, that's why [4] directly builds them in.*

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

2.5.2 Language of an NFA

- remember $\mathcal{L}(\mathcal{A})$ (Definition 2.3.3 on page 24)
- 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.

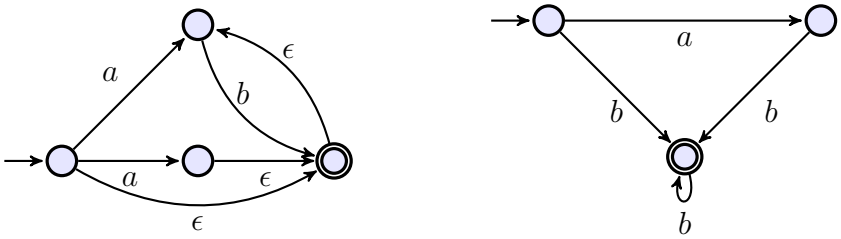
Definition 2.5.2 (Acceptance with ϵ-transitions)**.** A word w over alphabet Σ is *accepted* by an NFA with ϵ-transitions, if there exists a word w' which is accepted by the NFA with alphabet $\Sigma + \{\epsilon\}$ according to Definition 2.3.3 and where w is w' with all occurrences of ϵ removed.

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.

2.5.3 NFA vs. DFA

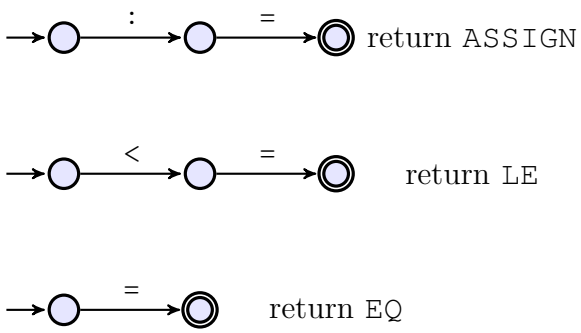
- *NFA*: often easier (and smaller) to write down, esp. starting from a regular expression
- non-determinism: not *immediately* transferable to an *algo*



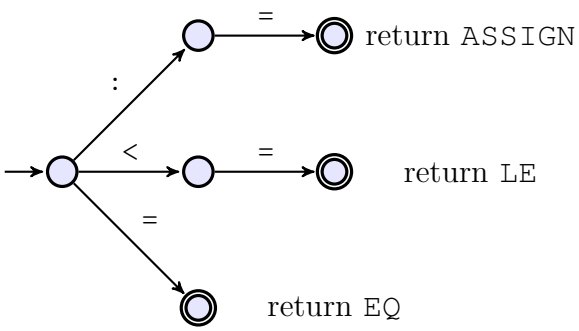
2.6 From regular expressions to DFAs (Thompson's construction)

2.6.1 Why non-deterministic FSA?

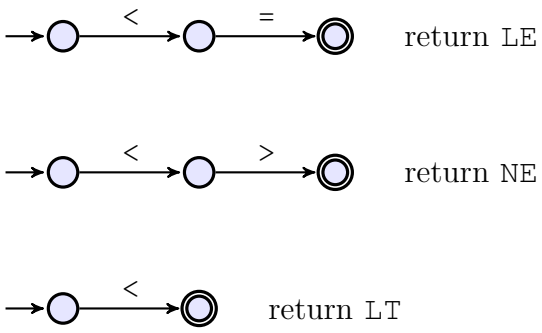
Task: recognize :=, <=, and = as three different tokens:



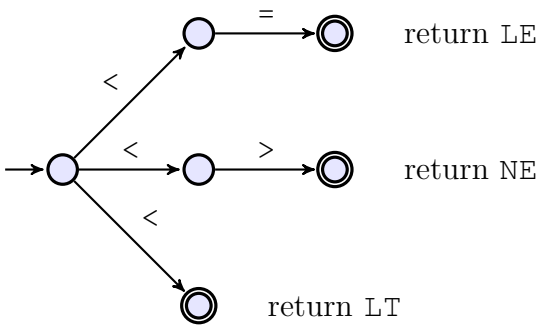
2.6.2 FSA (1-2)



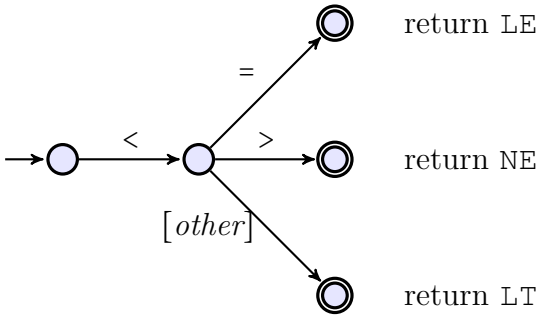
2.6.3 What about the following 3 tokens?



2.6.4 Non-det FSA (2-2)



2.6.5 Non-det FSA (2-3)



2.6.6 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 [?]

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

Compositionality

- construction slightly[15] simpler, if one uses automata with one start and one accepting state

⇒ ample use of ϵ-transitions

Compositionality

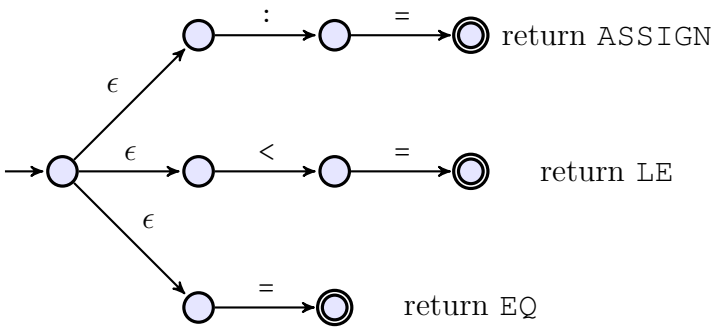
Remark 8 (Compositionality)**.** *Compositional concepts (definitions, constructions, analyses, translations . . .) are* immensely *important and pervasive in compiler techniques (and beyond). One example already encountered was the definition of the language of a regular expression (see Definition 2.2.4 on page 18). The design goal of a compositional translation here is the underlying reason why to base the construction on* non-deterministic *machines.*

Compositionality is also of practical importance ("component-based software"). In connection with compilers, separate compilation *and (static / dynamic) linking (i.e. "composing") of separately compiled "units" of code is a crucial feature of modern programming languages/compilers. Separately compilable units may vary, sometimes they are called modules or similarly. Part of the success of C was its support for separate compilation (and tools like* `make` *that helps organizing the (re-)compilation process). For fairness sake, C was by far not the first major language supporting separate compilation, for instance FORTRAN II allowed that, as well, back in 1958.*

Btw., Ken Thompson, the guy who first described the regexpr-to-NFA construction discussed here, is one of the key figures behind the UNIX operating system and thus also the C language (both went hand in hand). Not suprisingly, considering the material of this section, he is also the author of the grep *-tool ("globally search a regular expression and print"). He got the Turing-award (and many other honors) for his contributions.* □

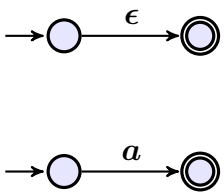
[15] It does not matter much, though.

2.6.7 Illustration for ϵ-transitions



2.6.8 Thompson's construction: basic expressions

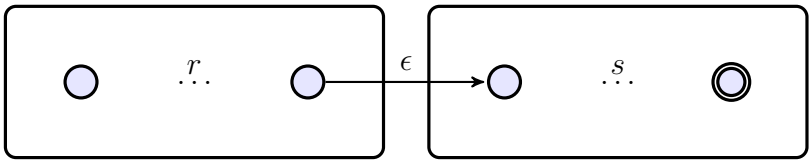
basic (= non-composed) regular expressions: $\boldsymbol{\epsilon}$, $\boldsymbol{\varnothing}$, $\boldsymbol{a}$ (for all $a \in \Sigma$)

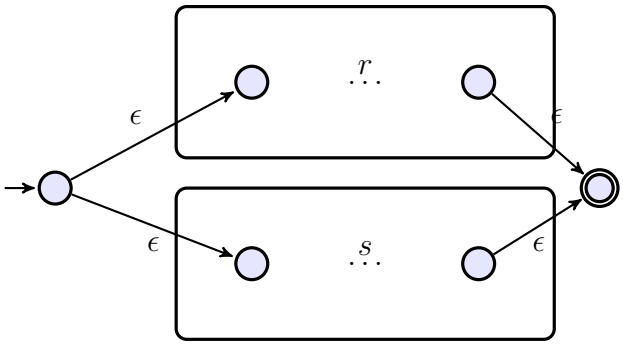


Remarks

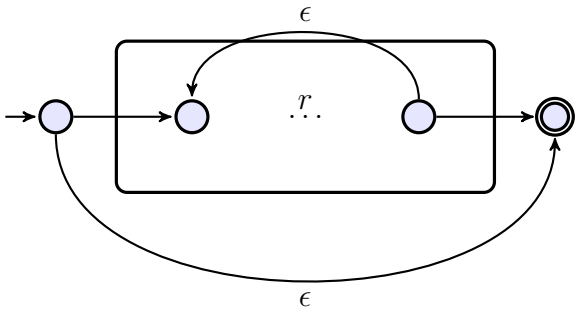
The $\varnothing$ is slightly odd: it's sometimes not part of regular expressions. If it's lacking, then one cannot express the empty language, obviously. That's not nice, because then the regular languages are not closed under complement. Also: obviously, there exists an automaton with an empty language. Therefore, $\varnothing$ should be part of the regular expressions, even if practically it does not play much of a role.

2.6.9 Thompson's construction: compound expressions

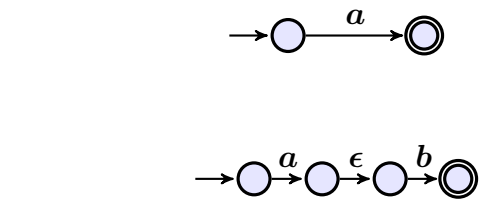


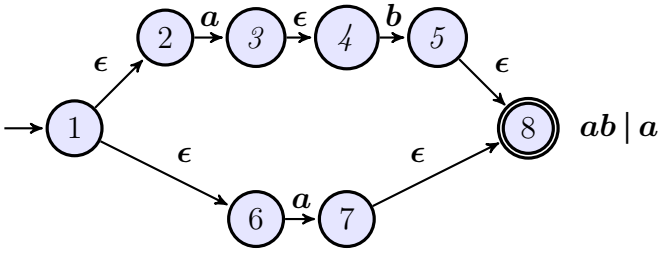


2.6.10 Thompson's construction: compound expressions: iteration



2.6.11 Example: $ab \mid a$





2.7 Determinization

2.7.1 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" $\Rightarrow$
- 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

Remarks

- 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.[16]
- origin: [5]

2.7.2 Some notation/definitions

Definition 2.7.1 (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.

ϵ-closure

Remark 9 (ϵ-closure)**.** *[4] does not sketch an algorithm but it should be clear that the ϵ-closure is easily implementable for a given state, resp. a given finite set of states. Some textbooks also denote λ instead of ϵ, and consequently speak of λ-closure. And in still other contexts (mainly not in language theory and recognizers), silent transitions are marked with τ.*

It may be obvious but: the set of states in the ϵ-closure of a given state are not "language-equivalent". However, the union of languages for all states from the ϵ-closure corresponds to the language accepted with the given state as initial one. However, the language being accepted is not the property which is relevant here in the determinization. The ϵ-closure is needed to capture the set of all states reachable by a given word. But again, the exact characterization of the set need to be done carefully. The states in the set are also not equivalent *wrt. their reachability information: Obviously, states in the ϵ-closure of a given state may be reached by* more *words. The set of reaching words for a given state, however, is* not *in general the* intersection *of the sets of corresponding words of the states in the closure.* □

2.7.3 Transformation process: sketch of the algo

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

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

Output: DFA $\overline{\mathcal{A}}$

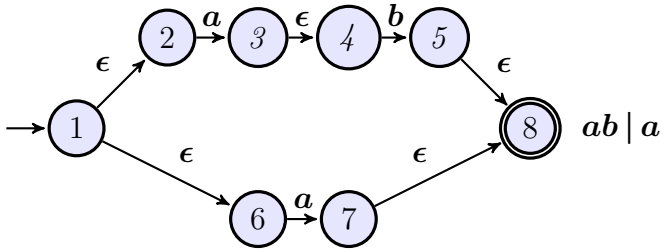
1. the *initial* state: $close_\epsilon(I)$, where I are the initial states of $\overline{\mathcal{A}}$
2. for a state Q' in $\overline{\mathcal{A}}$: the *a-successor* of Q is given by $close_\epsilon(Q_a)$, i.e.,

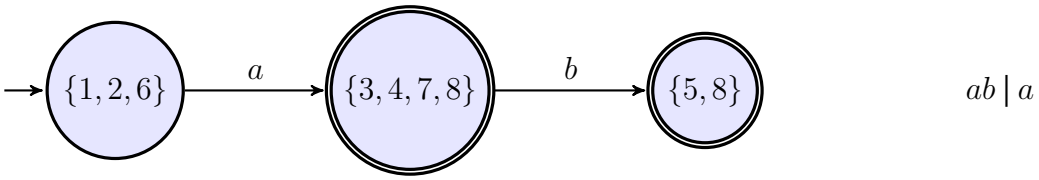
$$Q \xrightarrow{a} close_\epsilon(Q_a) \tag{2.18}$$

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.

2.7.4 Example $ab \mid a$

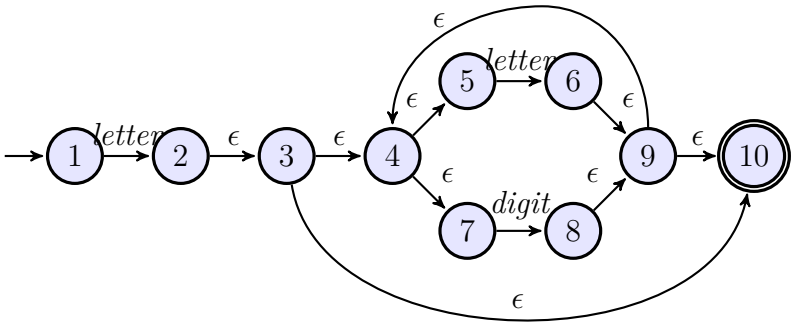




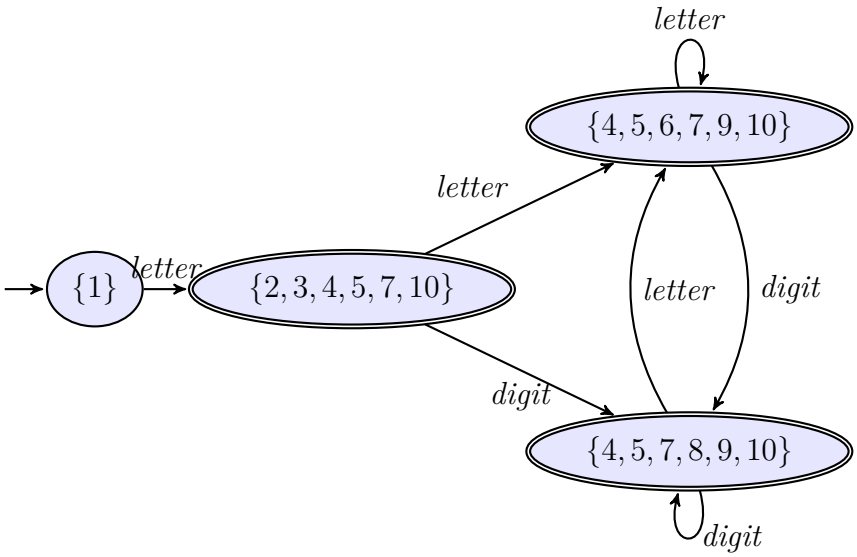
$ab \mid a$

2.7.5 Example: identifiers

Remember: regexpr for identifies from equation (2.13)



2.7.6 Identifiers: DFA



2.8 Minimization

Minimization

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

1. 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

2. Remarks
   - "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].

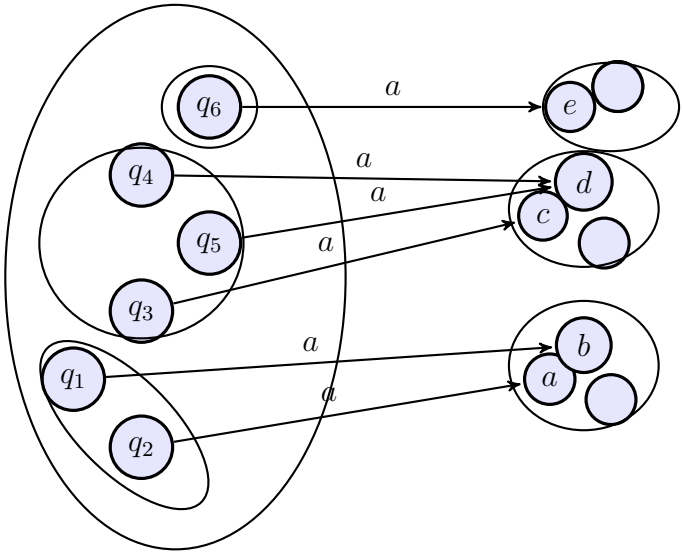
Hopcroft's partition refinement algo for minimization

- 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

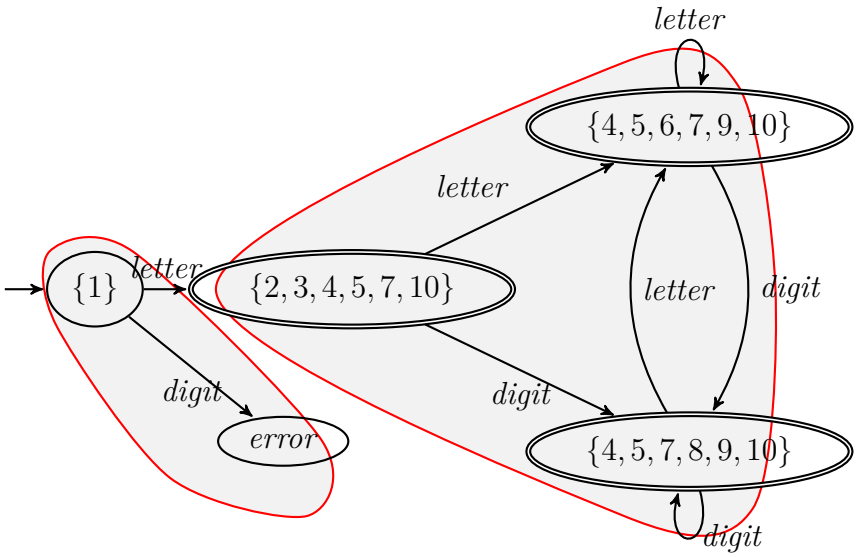


- before the split $\{q_1, q_2, \ldots, q_6\}$
- after the split on a: $\{q_1, q_2\}, \{q_3, q_4, q_5\}, \{q_6\}$

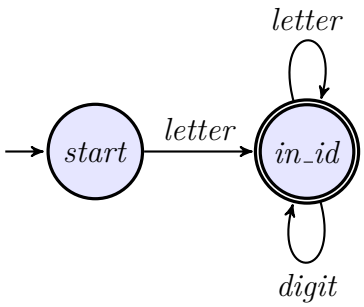
1. Note
   The pic shows only one letter a, in general one has to do the same construction for all letters of the alphabet.

Again: DFA for identifiers

Completed automaton

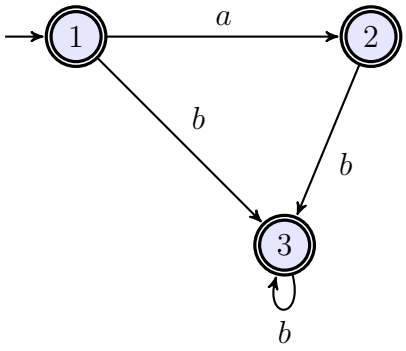


Minimized automaton (error state omitted)



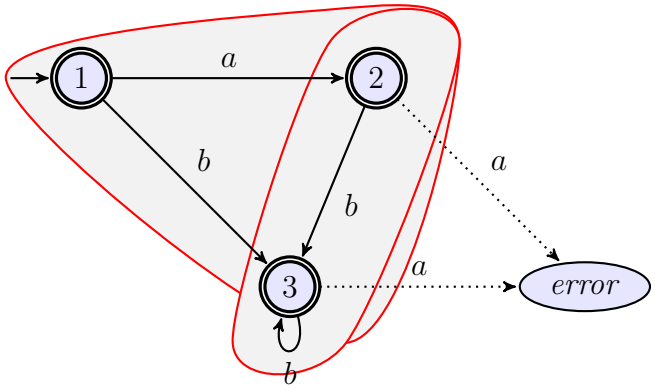
Another example: partition refinement & error state

$$(\boldsymbol{a} \mid \boldsymbol{\epsilon})\boldsymbol{b}^* \tag{2.19}$$

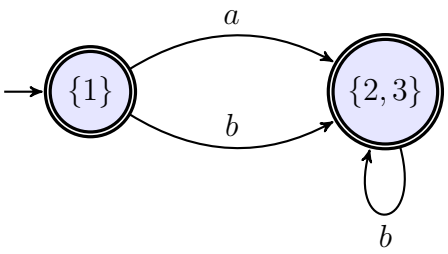


Partition refinement

error state added initial partitioning split after a



End result (error state omitted again)



2.9 Scanner implementations and scanner generation tools

This last section contains only rather superficial remarks concerning how to implement as scanner or lexer. A few more details can be found in [1, Section 2.5]. The oblig will include the implementation of a lexer/scanner.

2.9.1 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

2.9.2 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[17] (and whitespace, comments etc.)

[17] 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.

- 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)

2.9.3 Sample flex file (excerpt)

```
1
2  DIGIT    [0-9]
3  ID       [a-z][a-z0-9]*
4
5  %%
6
7  {DIGIT}+    {
8                printf( "An integer: %s (%d)\n", yytext,
9                        atoi( yytext ) );
10               }
11
12 {DIGIT}+"."{DIGIT}*        {
13               printf( "A float: %s (%g)\n", yytext,
14                       atof( yytext ) );
15               }
16
17 if|then|begin|end|procedure|function        {
18               printf( "A keyword: %s\n", yytext );
19               }
```

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] Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–42. Princeton University Press.

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

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

Index