Course Script

INF 5110: Compiler construction

INF5110, spring 2021

Martin Steffen

Contents

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 a large part of [4, Chapter 2]. The material is pretty canonical, anyway.

Contents

2.1 Introduction

The scanner or lexer is the first phase of a typical compiler (leaving out preprocessing, which is more seen as something that happens "before" the compiler does its job). What a lexer does is also called *lexical analysis*, basically chopping up the input string into smaller units (so-called lexemes), classifying them according to the lexical rules of the language one implements, and handing over the results of that chopping-up-and-classification to the parser in a stream of so-called tokens. The theory underlying lexers is that of *regular languages*. Typically, lexical aspects of a language are specified using some variant of *regular expression*. The lexer program then has to implement that specification, in that it is able to read in the source program (it *scans* it) and the checks it for compliance with the specification and, at the same time, does the chopping-and-classification task mentioned (it *tokenizes* the input string). Checking for compliance with regular expression is done via *finite-state machines*. Finite-state machines are equivalent to regular expressions insofar that they can describe the same class of languages. Here, language is meant as sequence of characters from an alphabet. Regular expressions are declarative in nature (hence more useful for specification), whereas finite state automata are more *operational* in nature, hence used in implementing a scanner. We discuss how to translate regular expressions to automata. The reverse translation is also possible (and easy), but we don't discuss that, as it's not needed for a compiler. The `lex` tool actually does just that: the users specifies the

lexical aspects of the language to compile and `lex` generates from that the lexer for that language, based on the theory of regular expression and finite state automata. Actually, tools like `lex` do a bit more, which mostly has to do with with support to generate tokens and to interface properly with the parser. Parsing will be covered in subsequent chapters.

Scanner section overview

What's a scanner?

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

regular expressions

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

We said the input of a scanner is the "source code". That's a bit unspecific. It's often a "character stream" or a "string" (of characters). Practically, the argument of a scanner is often a *file name* or an *input stream* or similar. Or the scanner in its basic form takes a character stream, but it "alternatively" also accepts a file name as argument (or even an url). In that case, of course, the string of the file name is not scanned as source code, but it's used to access the corresponding file, whose content is then read in in the form of a "string" or whatever.

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.
- *tokens* already language-specific.
- works always "left-to-right", producing one *single token* after the other, as it scans the input
- it "segments" char stream into "chunks" while at the same time "classifying" those pieces $\Rightarrow$ **tokens**

Characters are typically language-independent, but perhaps the encoding (or its interpretation) may vary, like ASCII, UTF-8, also Windows-vs.-Unix-vs.-Mac newlines etc. In contrast, tokens are already language dependent, in particular, specific for the grammar used to describe the language. There are, however, large commonalities across many languages. Many languages support, for instance, strings and integers, and consequently, it's plausible that the grammar will make use of corresponding tokens (perhaps called `INT` and `STRING`, the names are arbitrary, like variable names, but it is a good idea to call the token representing strings `STRING` or similar...). Tokens are not just language-specific wrt. the language being implemented. They are also show up in the implementation, i.e., they are specific to the meta-language used to implement the compiler. As in this lecture, we will use a parser and lexer generating tool (a variant of lex and yacc), and the representation of the tokens is also specific to the chosen tool.

The slides also mention that scanning works from left to right. That is not a 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

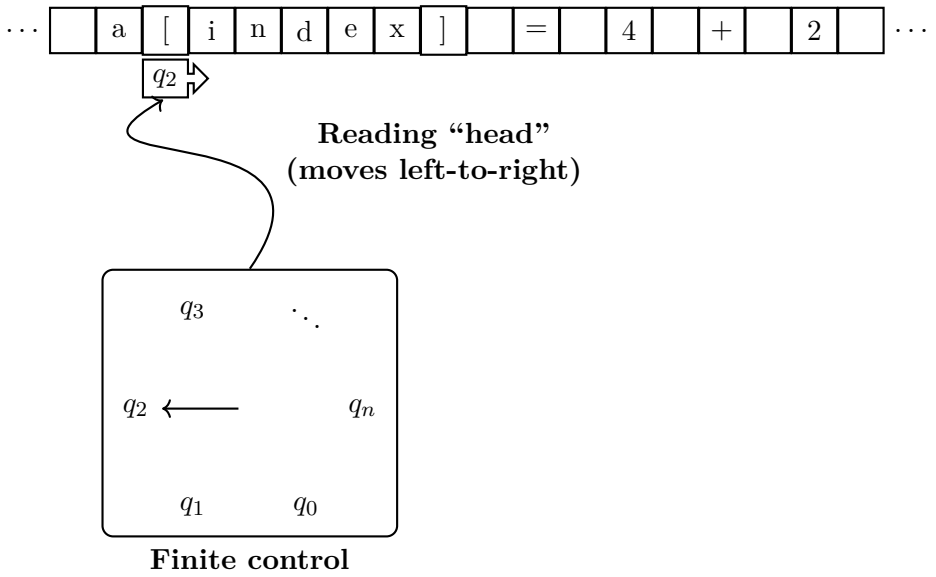
- segment & classify char stream into tokens
- typically described by "rules" (and **regular expressions**)
- typical language aspects covered by the scanner
  - describing *reserved words* or *key words*
  - describing format of *identifiers* (= "strings" representing variables, classes ...)
  - comments (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?

- 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"

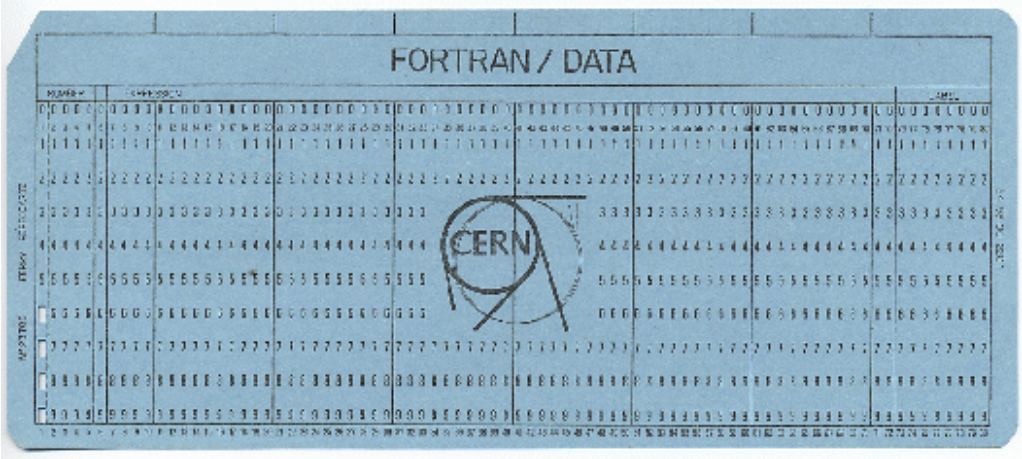
The picture of a reading head may be reminiscent of the typical picture illustrating Turing machines (which is not a coincidence). But a "reading head" is is not just a theoretical construct. In the old times, program data may have been stored and read from magnetic tape. Very deep down, if one still has a magnetic disk as opposed to an SSD, the secondary storage still has "magnetic heads", only that the compiler typically does not scan or 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".[1]

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

- Fortran was considered high-level (wow, a language so complex that you had to compile it . . .)



(Slightly weird) lexical ascpects of Fortran

Lexical aspects = those dealt with by 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
```

We have a look at Fortan to get a feeling for "alternative" ways how to deal with lexical aspects of a language (no more supported). It's in a way like in the super-old days when there was no "white space" in writing (for instance ancient Latin) and it entered manuscripts (in Latin or emerging Western European languages) only slowly. It proved helpful in reading for humans, of course.

Over time, also programming languages adopted a more "helpful" treatment of lexical aspects. Remember that one core task of scanning is segmenting the input, and white space can help there. If one treats white space as "basically not there" and thus absolutely meaningless, one does a big disfavor for human readability: humans are used to white space since the the times of no-white-space texts are long gone. One reason why initially Fortran treated white space like that was perhaps: it may have been the easiest thing to do: if the scanner reads a white space, do nothing and proceed. Or perhaps the motivation was to allow "compact programs". It allowed the expert programmers to write programs without wasting precious memory for "white space" in the source code. Note that in the conventional interpretation of white space nowadays, white space does not *exactly*

represent “nothing” in that one can put it in or out without changing the meaning. White space has no meaning by itself but *terminates* preceding non white space.

That treatment is so conventional, that most compilers use more or less the same definition of “white space” though there is typically not only one “white space” character. There is tabs, spaces, and then there is different “end-of-line” representations (carriage-return, end-of-line, newline). In a way, things like “carriage-return” `CR` and "tabulation command” is anyway a hold-over from the times of the mechanical and electrical type-writer era: at the beginning, the input and output peripheral devices connected to a computer were not just trying to behave like type-writer in software, there were *actually* sort of type-writers (and built in many cases by IBM at any rate...) and one needed some encoding of files to drive them or read input from them: the encoding for the “bell character” actually may result in banging on a small copper bell... Standard devices like `tty` are likewise remembrance of reat hardware teletype (typewrite-style) terminals, Since those codes are part of the ASCII-code (from the 60ies), those “characters” or “symbols” are here to stay...

There are other “unconventional” ways to deal with white space. For instance, one could make the decision that "indentation" (via "tabbing" or otherwise) has a meaning, as opposed to be another example of whitespace. Python is an example of a language, where indentation is “meaningful” and the lexer (and parser) must be aware of that.

Proper and improper indentation is sometimes also layed down in style guides. It’s more like a recommendation for programmers to follow for writing “pretty programs”. Iimproper indentation would make no semantic difference on the compilation (as would be the case in Python), it’s just frowned up as bad taste. Perhaps the compiler would utter some critizism or warning about the uglyness of the program. A lexer that to support checking of some stylistic guidelines and proper formatting would have to distinguish between different things commonly just treated as whitespace. That’s not surprising, as those tools focus on how nice the program looks (to the user). There are *formatting* tools (for instance `gofmt` for go) that transforms a program is a nicely written one that follow the stylictic guidelines.

The lecture will neither be concerned with the stone-age treatment of white-space as in old Fortran, not with the more elaborate ways discussed afterward.

Fortran scanning: remarks

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

The treatment of white space is mostly a question of language *pragmatics*. Pragmatics deals with non-formal questions like "what's helpful for *humans* that program in a language", like what's "user-friendly syntax". Lexers/parsers would have no problems using `while` as variable, but humans tend to. Pragmatics for instance also deals with questions like "how verbose should a language design be", how much syntactic sugar it should offer, etc. There is mostly no commonly agreed best answer, and it depends also on what kind of user one targets and to some large part on the personal taste and education or experience of the programmer.

Sometimes, the part of a lexer / parser which removes whitespace (and comments) is considered as separate and then called *screener*. It's not a very common terminology, though.

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 `"<="`

The remark about the "overlap" refers to some aspect of the lexical analysis, that was refered to ealier as that the scanner has to deal with *priorities*. If one has some sequence <= (and the classification from the slide), then, without further elaboration, the < part of the string could be seen as representing the relation "less" and the subsequent symbol as `equality`. Whether or not that makes sense is not for the scanner to decide, the scanner just chops up the string into pieces, and then the subsequent parser may complain that it's syntactically not allowed to have to relation symbols side by side (or not complain, depending on the grammar).

In that particular situation, language pragmatics would suggest that <= is *not* chopped-up but treated as one chunk, i.e., one *lexeme*, namely representing the relation less-or-equal.

The same principle also apply to other entries in the classification. For instance `abc123` is intended in most languages as *one* identifier, not as `abc` followed by the number `123`, or even more weiredly by three identifiers followed by three separate digits).

So, the "priority" is: prefer *longer* lexemes over shorter ones (with white-space as possible "terminator", unlike as in old Fortran, as discussed earlier).

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;
```

The second version makes use of so-called union-types in C. The "union-type" of C has some deficiencies (as some nowaways would say). We will shortly look at union types and others in a later chapter about type checking. In any case, the concrete C implementation here is not very relevant (maybe not even recommended...).

One can do it analogously in Java using classes. When discussing the Oblig, we will give hints on how to do it and how it's concretely represented in the version of lex/yacc we suggest to use.

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 similar format string attacks can involve 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? Note: unclarities about "what's a backslash" have been used for security attacks. Remember that "the" backslash-character in OSs often has a special status, like it *cannot* be part of a file-name but used as separator between file names, denoting a *path* in the file system. If one can "smuggle in" an inofficial ("chinese") backslash into a file-name, one can potentially access parts of the file directory tree in some OS which are supposed to be inaccessible. Attacks like that have been used.

Parser generator

The most famous pair of lexer+parser tools 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. lex/yacc originate from C, there are also gnu-versions around (called flex and bison). Those kinds of tools are seldomly called compiler compilers any longer. Many other languages ship with a corresponding pair of tools. In the lecture, someone from the audience mentioned Alex & Happy (for Haskell), ocaml has ocamllex/ocamlyacc, similar for other ML versions and other languages. Java, for some reason, does not ship with such a pair of tools; they exist though, for instance JLex and CUP.

Those tools are all based on the same principles, they work roughly similar and can generate parsers for the same class of languages (language in the theoretical meaning of sets of words over an alphabet): the lexers cover some (extended) form of regular expressions and the parser does some form of bottom-up parsing, known as LARL(1) parsing. This form of parser/lexer generators inspired by lex/yacc is the bread-and-butter, standard version. The overview at Wikipedia over different such tools is pretty long.

Sample prosa spec

The following is a excerpt from this year's oblig concerning the lexical conventions for *compila 20*. Actually, the lexical part does not really change over the years, or only in minor aspects. Anyway, it's an example of a prosa specification, and the oblig involves in writing a lexer for that. Since the lexer is supposed to be based on a lex-style tool, this means, one has to capture the prosa text in the regular language format of the chosen tool (for instance JLex).

2 Lexical aspects

2.1 Identifiers and literals

- `NAME` must start with a letter, followed by a (possibly empty) sequence of numeric characters, letters, and underscore characters; the underscore is not allowed to occur at the end. Capital and small letters are considered different.
- All *keywords* of the languages are written in with lower-case letters. Keyword *cannot* be used for standard identifiers.
- `INT_LITERAL` contains one or more numeric characters.
- `FLOAT_LITERAL` contains one or more numeric characters, followed by a decimal point sign, which is followed by one or more numeric characters.
- `STRING_LITERAL` consists of a string of characters, enclosed in quotation marks (`"`). The string is not allowed to contain line shift, new-line, carriage return, or similar. The semantic *value* of a `STRING_LITERAL` is only the string itself, the quotation marks are not part of the string value itself.

2.2 Comments

Compila supports *single line* and *multi-line* comments.

1. Single-line comments start with `//` and the comment extends until the end of that line (as in, for instance, Java, C++, and most modern C-dialects).
2. Multi-line comments start with `(*` and end with `*)`.

The latter form cannot be nested. The first one is allowed to be "nested" (in the sense that a commented out line can contain another `//` or the multi-line comment delimiters, which are then ignored).

2.2 Regular expressions

Regular expressions are a very well-known concept and, with variations, used in different applications, inside compilers, editors, as system tools and utilities, for specifying search patterns, and many more. Many programming languages, notably "scripting" languages offer extensive support for working with regular expression and extended regular expressions. Besides that, they have been studied theoretically and there are also imporant generalizations (which are outside of the scope of the lecture). There are also "practical" variations. In the lecture, we start focusing on the "classic, vanilla core regular expressions". They capture the relevant parts. For usability, one often likes to offer extra syntax, that makes the use of regular expression more convenient. The lex-style tools would like to do that. Adding more constructs to a language for convenience without really extending the expressivity of a languages in this way is sometime called "syntactic sugar". Other practical extensions of regular expressions may fall outside that classification: one really likes to add *expressivity*. Those extensions are sometimes called "extended regular expression", and those may still keep central aspects of regular expression, but being actually more expressive, fall outside the formalisms that captures *regular languages*. We will look at some abbreviations that fall into the "syntactic sugar" category, but won't venture into genuine extentions (which are technically not longer pure regular expressions, as said). Practically, for the oblig, one has to cope with the concrete syntax and posibilities of the chosen lexer generator, for instance JLex.

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 lexeme is tokenized

A lexer generator may even prepare useful error messages if scanning (not scanner generation) fails, i.e., running the scanner on a lexically illegal program. Of course, if the scanner generation itself fails, also there meaingful errors messages and giving reasons for the failure are welcome. A final source of error could be: the scanner generation produces a scanner, which is a Java/C/whatever program, and that one is incorrect, starting from being syntactically incorrect or ill-typed.

The classification in step 2 is actually *not* directly covered by the classical results that stating reg-expr = DFA = NFA, it's something extra. The classical constructions presented here are used to *recognise* (or reject) words. As a "side effect", in an actual implementation, the "class" of the word needs to be given back as well, i.e., the corresponding *token* needs to be constructed and handed over (step by step) to the next compiler phase, the parser.

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

As for the origin of regular expressions: one starting point is Kleene [3] and there had been earlier works *outside* "computer science".

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 link 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", artificial intelligence, etc. Oh, wait, AI we still have (and the word and discipline dates back to the 50ies...)

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.

- practical examples of alphabets: ASCII, Norwegian letters (capital and non-capitals) etc.

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, the Σ is left "implicit" (as assumed to be understood from the context).

Remark: Symbols in a symbol table (see later)

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.

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. $\{ab, bbab, aaaaa, bbabbabab, aabb, \ldots\}$: ?????

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). But 2 letters are often enough to illustrate some concepts, 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 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 character sets, while not actually infinite, are large or extendable (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" .

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

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

Remark 2 (Programming languages as "languages")**.** *When seen syntcactically as all possible strings that can be compiled to well-formed byte-code, Java etc is also is a* language *in the sense we are currently discussing, namely 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 be represented by a fraction, resp. in the periodic way as in equation (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).*

Regular expressions

Definition 2.2.3 (Regular expressions)**.** A *regular expression* is one of the following

1. a *basic* regular expression of the form **a** (with $a \in \Sigma$), or $\boldsymbol{\epsilon}$, or $\boldsymbol{\emptyset}$
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, | By "concatenation", the third point in the enumeration is meant. It is written or represented without explicit concatenation operator, just as juxtaposition, like **ab** is the concatenation of the characters **a** and **b**, and also for concatenating whole words: $w_1\ w_2$.

Regular expressions

In Cooper and Torczon [1], $\boldsymbol{\emptyset}$ 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 | for "alternative" or "choice" is a known convention. The | 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 |-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 | 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).

A "grammatical" definition

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

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

We will see enough context-free grammars of the form given on this slide and the following. They will be central to parsing and their definition and format will be explained in detail there. Here, it's in "preview", we use the context-free grammar notation (known as EBNF) to describe one particular notation, namely the notation known as *regular expressions*.

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 in the parsing chapters and use capitals for non-terminals, the grammar for regular expression looks as follows:

Grammar

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

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 expressions (a notation itself)
- for now: carefully use *notational* or *typographic* conventions for precision

To be careful: we will *later* (when dealing with parsers) distinguish between context-free languages on the one hand and *notations* to denote context-free languages on the other.

In the same manner here: we *now* don't want to confuse regular languages as concept from particular notations (specifically, regular expressions) to write them down.

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
  - $\boldsymbol{\emptyset}$ and $\emptyset$
  - $\mathbf{|}$ and $|$ (especially hard to see `:-)`)
  - ...
- later (when gotten used to it) we may take a more "relaxed" attitude towards it, assuming things are clear, as do many textbooks.

Remark 4 (Regular expression syntax)**.** *We are rather careful with notations and meta-notations, especially at the beginning. Note: in compiler* implementations*, the distinction between language and meta-language etc. is very real (even if not done by typographic means as in the script here or textbooks ...): the programming language being implemented need not be the programming language used to implement that language (the latter would be the "meta-language"). For example in the oblig: the language to implement is called "Compila", and the language used in the implementation will (for most) be Java. Both languages have concepts like "types", "expressions", "statements", which are often quite similar. For instance, both languages support an integer type at the user level. But one is an integer type in Compila, the other integers at the meta-level.*

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 I should point it out here). 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 itself.*

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 bother to tell the reader that it's fine to use "space" between different expressions (like* $a \mid b$ *is the same expression as* $a \quad | \quad b$*). 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 two concepts will play a prominent role later in the grammar and parsing sections and will become clearer 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 the instruction for the reader that "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`*) will typically occur.*

To sum up the discussion: 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.

Same again once more

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

Note:

- symbol $\mathbf{|}$: (bold) as symbol of regular expressions
- symbol $|$: (normal, non-bold) meta-symbol of the CF grammar notation
- the meta-notation used here for CF grammars will be the subject of later chapters
- this time: parentheses "added" to the syntax.

This is just a more "condensed" representation of the grammar we have seen before. We will see many examples later when discussing context-free grammars. In particular note the two "different" versions of the | symbol: one as syntactic element for regular expressions, one as symbol used in context-free grammars on the meta-level, used to *describe* the syntax of regular expressions. Though these levels are clearly separated, the intended meaning of the symbol is kind of the same, it represents "or".

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).

$$
\begin{array}{rcll}
\mathcal{L}(\boldsymbol{\emptyset}) & = & \{\} & \text{empty language} \\
\mathcal{L}(\boldsymbol{\epsilon}) & = & \{\epsilon\} & \text{empty word} \\
\mathcal{L}(\boldsymbol{a}) & = & \{a\} & \text{single “letter” from } \Sigma \\
\mathcal{L}(rs) & = & \{w_1 w_2 \mid w_1 \in \mathcal{L}(r), w_2 \in \mathcal{L}(s)\} & \text{concatenation} \\
\mathcal{L}(r \mid s) & = & \mathcal{L}(r) \cup \mathcal{L}(s) & \text{alternative} \\
\mathcal{L}(r^*) & = & \mathcal{L}(r)^* & \text{iteration}
\end{array}
\tag{2.7}
$$

- conventional *precedences*: $*$, concatenation, $|$.
- Note: left of “=”: reg-expr *syntax*, right of “=”: semantics/meaning/math [2]

Explanations

The definition may seem a bit over the top. One could say, the meaning of the regular expression is clear enough when described in simple prose. That may actually be the case. But it actually means, regular expressions and the meaning of an expression, which is the set of words it describes, is likewise straightforward. Nonethless, we make the “effort” to define the meaning. First of all, precision does not hurt, within a compiler lecture and outside. In other situations, the question of “what does it mean”, i.e., the question of semantics, become more pressing. One can ask the same question about later other formalism, like the meaning of context-free grammars. Thirdly, in this simple situation, the description of the meaning of a language hopefully makes the different levels more clear: the syntactic level (symbols) and the semantic level resp. the meta-level (math). Of course, “math” is a discipline which has its own symbols and notations. In this particular case of regular expressions, they are pretty close. And of course the description of the semantics using math assumes that the reader is familiar with those notations, so that a definition like $\mathcal{L}(r \mid s) = \mathcal{L}(r) \cup \mathcal{L}(s)$ is helpful or more compact than an English description. But of course, it just a way of saying “the regular expression symbol $|$ **means** set union”. Indeed, another motivation is that this form of semantic definition is a form of translation, i.e., “compilation”. In this case from one notational form (regular expression) to another one (mathematical notation, whose meaning is assumed to be clear). Semantics and translations from one level of abstraction to another one are also needed for *programming languages* themselves, though we don’t go there in this lecture. For instance, in the oblig, the compila language has to be translated to a lower level. We could have specified the semantics of compila more formally, though the definition would be much more complicated (and probably use different techniques) than the semantics of regular languages. We could even go more ambition: not only define the semantics of compila, but also define the semantics of the language it is compiled to. That would be some form of “byte-code”. After having defined both levels of semantics, one could establish that both semantics do the same. That would be the question of *compiler correctness*. There are attempts of having a provably (!) correct compiler, or *verifying compiler*, though that is exceedingly complex, it’s seen as one of the so-called *grand challenges* in computer science.

[2] 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 $notb \triangleq a \mid c$

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 18, discussing the (non-)status of parentheses in regular expressions. If we think of Definition 2.2.3 on page 16 as describing abstract syntax and a concrete regular expression as representing an abstract syntax tree, then the constructor* $\mid$ *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 $\mid$ *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 19) 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 parentheses or other syntactic markers which may make the expression or program look cumbersome. On the other hand, the "light-weight" and "easy-on-the-eye" syntax may trick the unsuspecting programmer into misconceptions about what the program means, if unaware of the rules of associativity and 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 when to use 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* |*-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 | *and concatenation. For instance, one may wonder if* $\mathbf{ba \mid bc}$ *could be interpreted as* $(ba) \mid (bc)$ *and* $b(a \mid (bc))$ *and* $b(a \mid b)c$*. However, in Definition 2.2.4 on page 20, we stated* precedences *or priorities, stating that concatenation has a* higher *precedence over* |*, 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 unappealing even 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.* □

Additional "user-friendly" notations

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

Special notations for *sets* of letters:

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

naming regular expressions ("regular definitions")

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

The additional syntactic constructs may come in handy when using regular expressions, but they don't extend the expressiveness of the formalism. That's pretty obvious by the way the extensions are defined. Note that we don't explain the meaning or semantics of the new constructs in the same way as for the core constructs (defining $\mathcal{L}$ and giving their

mathematical interpretation). Instead, we expand the new constructs and express them in terms of the old syntax. They are treated as *syntactic sugar*, as one says.

Tools, utilities, and libraries working with regular expression (like `lex`) typically support sugared versions, though the exact choice of notation for the construct may vary.

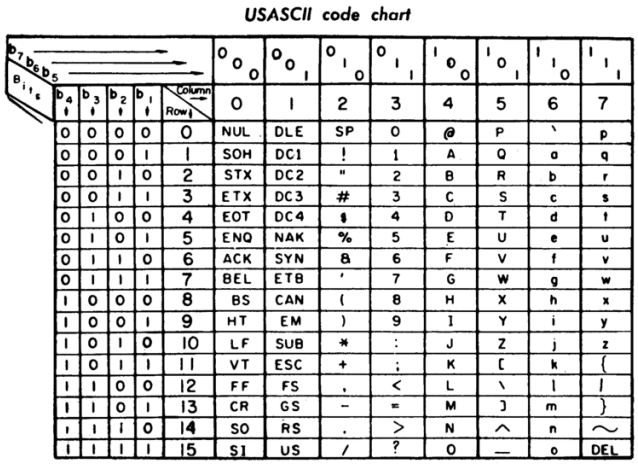
As mentioned, there are also so called *extended* regular expresions, where the extensions make the formalism more expressive than the core formalism (so those extensions are not syntactic sugar then).

One could look at the collection of constructors for the syntax of regular language, including the sugar, and wonder whether there aren't some missing. For example, we have in the language a form of "or" (disjunction), written |, one could ask, why not an "and" (conjuction, intersection), for instance. That's indeed an interesting, insofar it is an example which is not syntactic sugar on the one hand, but on the other hand does not extend the expressiveness for real. If one had regular expressions containing an "and", then one can alway find a different regular expression with the same meaning, without the "and". However, the transformation would not be of the same nature than for the syntactic sugar we added: the conjunction cannot just be expanded away; consequently one would not call that addition syntactic sugar. There exists other constructs that are non-sugar but do not add expressiveness (negation or complementation for example).

Mostly, such constructs like intersection or complementation are *not* part of the regular expression syntax, though theoretically, one would not leave the class of regular languages (= languages that can be expressed by regular expressions). Why are those then left out? It's probably a matter of pragmantics. One does not really need them for many things one want to do with regular expression, like describing lexical aspects of a language for a lexer. One wants to classify strings, and one is content by saying "It's whitespace (which is is this or this or this), *or* it's a number, *or* it's an identifier, *or* a bracket . . . ". Given also the fact that adding conjuction or negation or other non-sugar ingredient would make the some following constructions more complex, there is no real motivation to support conjunction. By the "following constructions" I mean basically the translation of a regular expressions into a (non-deterministic) finite state automaton. This translation, called *Thompson's construction*, will be covered later in this chapter. A tool like `lex` does this construction (followed by other steps). The construction is fairly simple, but if one conjuctions and complementations would drive up the size of the resulting automata. For intersection, for instance, one would needed a form of *product construction*, which is also conceptually more complex than the straightforward, compositional algorithm underlying Thompon's construction. Actually, it would not be so bad, since if one avoid using conjuction or negation, the size of the result would not blow up, so the reason, why regular expressions don't typically support those more complex operators is that pragmatically, no one misses them for the task at hand, at least not for lexers.

Ordered alphabet

We have defined an alphabet as a (finite) *set* of symbols. In practice, alphabets or character sets are not just sets, which are unordered, but are seen as ordered. Each symbol of the alphabet has a "number" associated to it (a binary pattern) which corresponds to its place in order in the sequence of symbols. One of the simplest and earliest established ordered



USASCII code chart

Bits b_7 b_6 b_5 →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 ↓ Row / Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

Figure 2.1: ASCII reference card

alphabets in the context of electronic computers is the well-known ascii alphabet. See Figure 2.1.

Having the alphabet ordered is one thing, having a "good" order or arrangement is a different one. The reference card shows some welcome properties, for instance, that all lower-case letters are contiguous and in the "expected" order, same for the capital case letters. Since the designers of ascii arranged it in that way, one can support specifying all lower-case letters as `[a-z]` and capital case letters as `[A-Z]`. What does *not* work is having all letters as `[a-Z]`, since, in ascii, the letters are not arranged like that. The capital letters come before the lower case letters, but also `[A-z]` would not work as intended, as there is a "gap" of other symbols between the lower-case and the upper-case letters. Isn't that stupid? Actually not, the arrangement as made clear in the figure, is such that the operation of turning a lower-case letter to a upper-case latter a matter of flipping bits. Another rational decision is to place the decimal numbers "align" partly with their binary representions. It's not that 0, 1, etc. are exactly the corresponding bit patterns, but at least parts of the word correspond to the binary pattern. Anyway, details like that don't matter too much for us, but one has to be aware of the concept of ordered alphabets as such, in order to specify, for example, all letters as `[a-zA-Z]` (or `[A-Za-z]`). Many encodings are nowadays extensions or variations of ascii, and also for those, specifications like `[a-z]` work. For instance, UTF-8. As a side remark: Ken Thompson (the one from Thompon's construction) was involved in working out UTF-8, an encoding that includes ASCII insfar that it's identical with ASCII in its first part. Of course, there are very many variations of UTF (and Unicode symbol set, of which UTF is a encoding scheme),

There are however, also *alternatives* to ascii, not just extension. One is *Extended Binary Coded Decimal Interchange Code (EBCDIC)* (actually also for EBCDIC, there are many variation). EBCDIC is perhaps mostly of historic interests as it is supported mainly by IBM mainframes and larger such computers. I mention EBCDIC here, because the encoding has the unfortunate property, that, for instance, capital letters are not contiguious.

For such an encoding, of course, things like `[A-Z]`, make no sense. The encoding would have other negative consequences, for instance, sorting a list of words is more tricky (or at least less efficient). However, EBCDIC still lives on, there exists Unicode encodings based on that (as opposed to based in extensions of ascii like UTF-8), which concequently are called UTF-EBCDIC. Once, some things are standardized, they never die out completely (and actually EBCDIC just inherits properties of punchcards, which existed before the modern electronic computer, to the new area. In that context, it's not a coincidence that IBM which was a big name in "punchcard processing equipment" and stuck to aspects of the encoding when it became a big name in electronic computers and mainframes.

2.3 DFA

In this section and the following we introduce the very central notion of finite state automata and cover their close relation to regular expression. Finite state automata are well-studied and play an important role also beyond their use for lexing. There are many different variations of finite state automata, also under different names. Some of them are mentioned on the slides. Such automata in their classic form are pretty simple objects, basically some graphs with labelled edges, and some nodes are singled out as start or initial nodes and some as final or accepting nodes. What makes such "graphs" automata or machines is their operational interpretation, they are seen as mechanisms that "run" or do steps. The nodes of the "graph" are seen as *states* the machine can be in. The edges are *transition*. It's assumed that "executions" of the machine starts in one of the initial states, and when in one final state, the execution ends, more precisely, *can end*. The mental picture of some entity, being in some discrete state, starting somewhere, doing steps or transitions one after the other is of course super-general and very unspecific. Basically all mechanized computing can be thought of operationally that way, going from one state to a next one and so on.

As the name indicates, specific here is that the number of states are *finite*. That's a strong restriction. Finite state automata are an important model of computation. It is also a model for hardware circuits, more specificially discrete, "boolean" circuits not analogue hardware. It's clear that a binary circuit can be only in a finite number of states, and finite state machine are a good model for describing such hardware. The automata in that case are a bit more elaborate than the ones we use here, in particular, one would use automata that don't have a unstructured alphabet, but one would conceptually distinguish between input and output (though possible on the same alphabet). There are different ways one can do that. Typically, the edges carry the output, whereas on can connect the output to the states, or alternatively to the edges, as well. Those two styles of finite-state input/output automata are called Moore-machines (= output on the state) resp. Mealy-machines (= output on the transitions). The two different models would also require different styles of hardware realization, but those things are not important for us.

For lexing, we are handling automata with an unstructured alphabet, without distinguishing input from output. Such single-alphabet automaton can be "mentally seen" that the edges *generate* the letters (i.e., the letters are the output). With this view, a given automaton *generates* a language, i.e., the set of all sequences of letters that lead from an initial to an accepting state. Alternatively, one can see the letters on the edges as input;

in this view, such machines are seen as *recognizers* or *acceptors*. The final states of an automaton are also called *accepting* states. Anyway, that view of acceptors is also the appropriate one for lexing or scanning. The letters of the alphabet are the characters from the input and the machine moves along and accepts a word (a lexeme of the language being scanned), and the accepting state corresponds to the token classes (for instance, an indentifier, or a number etc).

Coming back to the issue of finite state I/O automata we brushed. Actually, the lexer in the context of a compiler *can* be seen as involving input *and* output. The characters are the input, and the token (token class and token value) are outputs and parsing a file means making iterated use of that arrangement, handing over a token stream to the parser.

We (as basically all compiler books) focus in the classic theory of finite state automata, ignoring as far as the theory is concerned, the token-output part. This is also the part, which connects with the regular expression from before. Regular expressions specify the lexical aspects of the language, and finite state automata are the execution mechanism to *accept* the corresponding lexemes. If course, concretely, tools like `lex` need to arrange also for the token-output part, but if one has the input-part under control, there is not much to understand there.

One aspect that is important is the question determinism vs non-determinism. Determism in computational situations mean: there is (at most) one next state. Non-determinism means, there is potentially more than one, the future is not determined. For finite state automata, it's more precisely like as follows: Given a state and given an (input) symbol, say a, there is (at most) one successor reachable via an a-transition. One can also say: there is at most one a-successor. In other words, the current state *and* the input *determines* the next state (if any).

That's highly-desirable in a lexer: the lexer scans one letter after the other, and its not supposed to make guesses how to proceed. Doing so would lead do the danger of backtracking: in case the guess turns out to rejecting the input later down the line the lexer has to try to explore alternatives to find out if any of this could lead to accepting the input nonetheless. That's a horrible way to scan the input.

The good news is: one can avoid that. Intuitively the way to do it is to replace an non-determistic automaton by a different, but equivalent one, that conceptually explores all alternatives "at the same time". The determinization algorithm is known as *powerset construction* and is pretty straightforward and pretty natural.

Determinisation of automata-like formalisms

As some side remark concening the naturalness of the determinization procedure, or a word of cautioning. It's true, it's natural. However, strangely perhaps, it works not "universally". For instance, there are other automata-based formalisms that look quite similar. One such is finite-state automata that doen't work in finite words (as we do) but infinite words. Or finite-state automata that work in trees (either working top-down or bottom-up). We will not encounter those. What we will encounter, though, is a particular form of "infinite state automaton" known as *push-down automaton*. Those, by having an infinite about of memory, there are more expressive than finite state automata. That

are central for *parsing* (not lexing) of context-free languages. The amount of memory for push-down automata is infinite, but not "random access", i.e. one can only access the top of a stack (by pushing and popping content), and this restriction fits with context-free languages (in the same way that the finite-state restriction fits with regular languages).

Anyway, for all those automata-like constructions, there are deterministic and non-deterministic variants in that the respective input determines their reaction or not. However, the powerset construction would not work for those, which means, non-deterministic versions for those are strictly more expressive than deterministic ones (with except of bottom-up tree automata where determinism vs. non-determinism does not matter as for finite-state machine we are dealing with). Perhaps also interesting: for Turing machines, which can be seen as machines with finite control and infinite amount of random access memory (not just a stack), again determinism is not a restriction.

All that is meant just as a cautening not to assume that the powerset consumption can be transported "obviously" to other settings...

Finite-state automata

- simple "computational" machine
- (variations of) FSA's exist in many flavors and under different names
- other 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.

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

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.*

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

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

The definition given is fairly standard and whether one see δ as relation of function is, of course, equivalent. One often uses graphical representations to illustrate such automata; we will encounter numerous examples.

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:

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

We touched upon the issue in the introduction of the chapter already: non-determinism is "problematic". One could try **backtracking**, but, you definitel 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. As hinted at in the introducton: there is a way to turn a non-deterministic finite-state automaton into a deterministic version. We start by first defining the concept of determinism, resp. what constitutes a deterministic automaton

DFA: deterministic and total 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 ("complete")

Depending on which text one consults, the definition of DFA slightly disagrees. It's not a fundamental disagreement, it's more a question of terminology. It concern the notion of determinism, namely if one thinks whether being a deterministic automaton includes "totality" of the transition relation/transition function or not. Or in other words: for each state and each letter a, is there *exactly one* a-successor or *at most one*. One could make the argument, determinism means the latter: at each state, and for each input, the reaction is fixed: one either moves to one particular successor state, or else is "stuck." That corresponds to a definition where δ is a *partial function*, unlike the definition given, where δ is a *total function*. So, our definition of DFA means, the automaton is deterministic and total. Some would say a deterministic finite state automaton need not be total (being a separate aspect the automaton enjoys or not). But no one would say, a automaton who has only a partial successor function is non-deterministic.

Actually, it's a terminology question and does not matter much, basically it says: A DFA is a determistic *and* total finite state automaton (but we won't bother to call it DTFA or something). The reason why it does not matter much is that really there is no much difference anyway. A automaton with a partial transition function can always be completed into a total one by adding an extra non-accepting state, covering the situations when the partial automaton would otherwise be "stuck". That's so obvious, that one need bother talk about it much. Also later, when showing graphical representation of automata: when talking about DFA (and when we want to really stress that they are total), we still might leave out to show the extra state in the figure, it's just assumed that one understands that it's there.

As far as implementations of automata is concerned (for instance for lexing purposes): the "partial transition function" is also not too realistic. If the lexer eats one symbol which, at that point, is illegal, and for which there is no successor state, the lexer (and the overall compiler) whould not simply stop or deadlock. It will eat the symbol and inform the

surrounding program (the parser, the compiler) that this situation occured. It's indicates a form of error (a lexical error in the input), since we are dealing with an *deterministic* automaton, so there cannot be an alternative reading of the input that would have avoided that the lexer is stuck (or moved to a non-accepting state, or raised an exception etc). So, turning an automaton into a "total" or "complete" one is a non-issue, but removing non-determinism from an automaton is an issue. We will discuss *determinization* later.

For a relation, being *left-total* means, for each pair q, a from $Q \times \Sigma$, $\delta(q, a)$ is defined. When talking about functions (not relations), it simply means, the function is *total*, not partial.

Some people call an automaton where δ is not a left-total but a deterministic 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 deterministic *and* total.

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**.

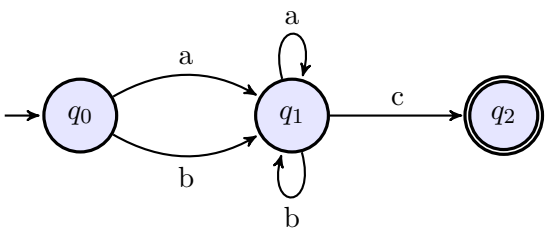
Definition 2.3.3 (Accepted words and language of an automaton)**.** A word $c_1 c_2 \dots c_n$ with $c_i \in \Sigma$ is *accepted* by automaton $\mathcal{A}$ over Σ, if there exists states $q_0, q_2, \dots, q_n$ from Q such that

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

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.

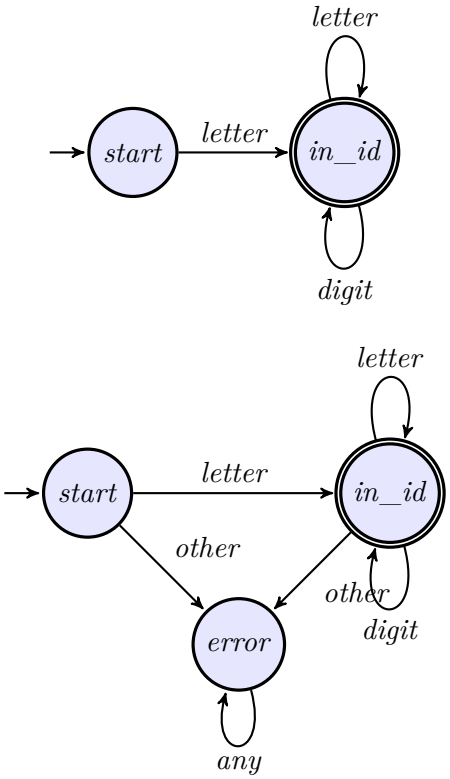
FSA example

The figure shows convention we will use throughout. The initial states, in this case only one, are marked by a "incoming" arrow. The accepting states, in this case also only one, is marked by having a double ring. The automaton is deterministic, though not complete. We could interpret the automaton as complete DFA, with one extra non-accepting state.



Example: identifiers

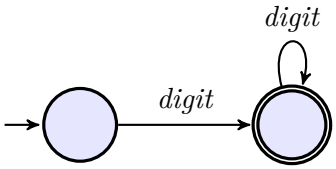
$$identifier = letter(letter \mid digit)^* \tag{2.13}$$



The example shows an automaton for identifiers, as they could appear in this or similar form in the lexer specifications for typical programming languages. They are specified using regular expressions, so it's also an illustration how regular expression can be translated into an automaton. The exact construction (which will be presented in three stages) will be covered later in this chapter, but the example is so simple, that one can easily come up with a deterministic automaton corresponding to the regular expression. There are two versions shown, one which is incomplete, and the second one with an extra state added (called *error*, but names of the states don't matter, they are only there to help the human reader).

Automata for numbers: natural numbers

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

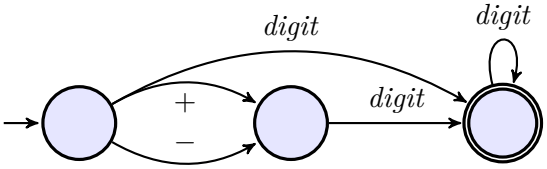


One might say, it's not really the natural numbers, it's about a decimal *notation* of natural numbers (as opposed to other notations, for example Roman numeral notation). Note also that initial zeroes are allowed here. It would be easy to disallow that. Another remark: we make use of some user-friendly aspect supported in many applied versions of regular expression, some form of syntactic sugar. That's the possiblity to use *definitions* or *abbreviations*. We give a name to the regular expression $[0-9]$ and that abbreviation *digit* is used for defining *nat*. That certainly makes regular expressions more readable, and we will continue that form of building larger concepts from simpler ones in the following slides.

Also $[0-9]$ and the $+$ symbol are syntactic sugar on top of the core regular expression syntax.

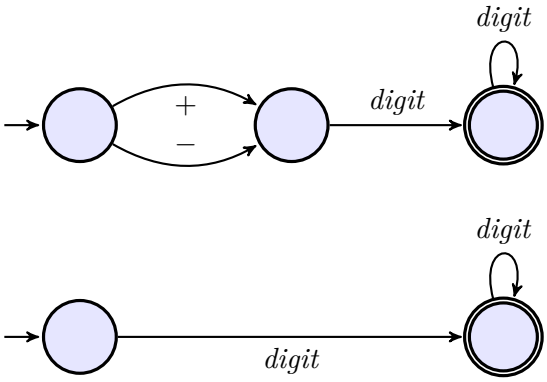
Signed natural numbers

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



Again, the automaton is *deterministic* (but not total). It's easy enough to come up with this automaton, but the *non-deterministic* one (on the next slide) 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"*.

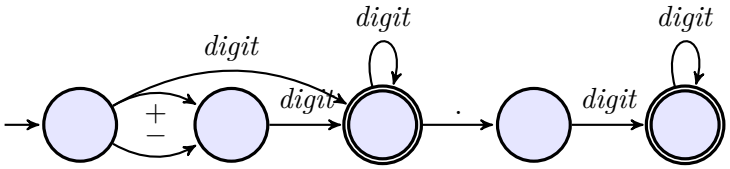
Signed natural numbers: non-deterministic



The automaton is non-deterministic by the fact that there are two initial states.

Fractional numbers

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



Note the “optional” clause in the regular expression and the corresponding fact that the automaton has multiple accepting states. Note that this does not count as *non-determinism*. If one considers the automaton as some abstract form describing a scanner, one could make the argument that there is non-determinism involved. It's true that, if one gives the automaton a sequence of letters, that *determines* its end-state (and thus whether the word is accepted or not). However, lexer's task will *also* have to segment the input and decide when a word if done (and then tokenized) and when not. In the given automaton, after having reached the first accepting state, one can make the argument that, if there is a dot following, the automaton has to make the decision whether to accept the word or to continue. That sounds like a non-deterministic choice (actually, seen like that it *would be* a non-determistic choice).

It just means, an deterministic automaton is not in itself a scanner. A scanner deals *repeateadly* with accepting words (and segmenting), a finite state automaton is dealing only with the question, whether a given word (already segmented, so to say) is acceptable or not. The current section deals just with acceptance or rejection of one word, and also

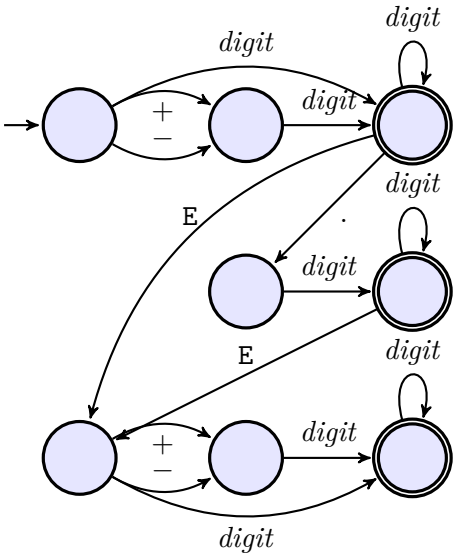
the standard, classical definition of determisim for automata is only concerned with that question.

Floats

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

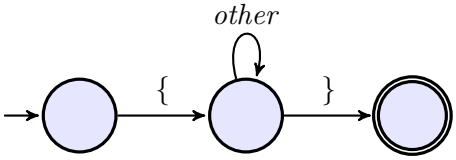
- Note: no (explicit) recursion in the definitions
- note also the treatment of *digit* in the automata.

DFA for floats

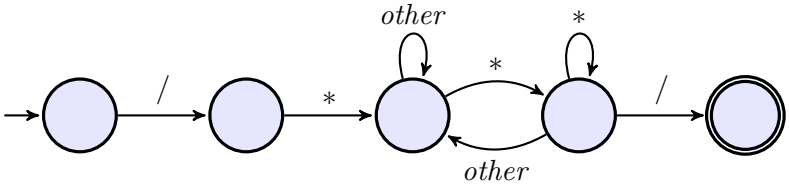


DFAs for comments

Pascal-style



C, C^{++}, Java



2.4 Implementation of DFAs

DFAs underly the implentation of lexers. The notion as such is simple enough, but a concrete lexer has to cover slightly more things than the purely theoretical coverage so far. One is that the lexer needs to be coupled up with the parser, feeding it with one token after the other. I.e., in the implementation, the "automaton" has not just sequences of characters as input, but also sequences of tokens as output. Related to that, the lexer is not just the implementation of one DFA, but it's a loop that repeatedly invokes the DFAa. Another aspect of the regular expressions resp. the DFA is the need for "priorities". We have mentioned the issue when discussing regular expression, for instance, when confronted with the string <=, then that is conventionally scanned as less-or-equal, and not as a < followed by a =.

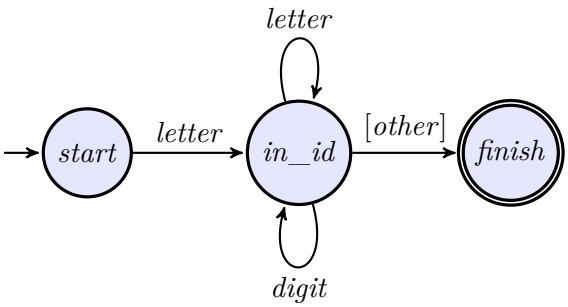
That aspect of "longest scan" is not really covered by the notion of DFA (or non-deterministic automaton as well). That has to do with the way, automata "accept" words. They start in their initial state, eat through the input, but when it come to acceptance in a lexer, it misleading to think like that: if you hit an accepting state, *accept the word* (and return the corresponding token). Sure, if a automaton hits an accepting state, this word seens so far is accepted, resp. belongs to the language the automaton describes. But it's presumably the only word, so there may be *another* run of the automaton (even if it is a deterministic one and is fed the same word as prefix), that reaches the accepting state, *and then continues*, perhaps accepting later down the road *a longer word* which extends the one the automaton could accept right now. Of course there may not be a guarantee that there exists a longer word. Anyway, a lexer explores one word only and makes decisions "on the spot" (being deterministic), preferring longer scans over shorter. In case of hitting an accepting state, it checks if one can proceed, still accepting the (extended) word. If *not* it accepts the word as is. This priority of proceed as long as possible and favoring longer words over shorter prefixes is *not* directly covered by the theoretical treatment of FSA so far, but it's an aspect an implementation would have to do.

The section brushes also on *data structures* one can use to implement DFAs, resp. scanners. We don't go too deep, bascially we sketch how one could use *tables* to represent automata (which can be realized for instance by two-dimensional arrays or in other ways). Tools like `lex` allows the compiler writer to ignore details there, as that's what those tools do: generate appropopriate data structures representing the DFA, taking care also of the other aspects mentioned, and interfacing with the parser component of a compiler. Later in this

section, we introduce a notation labelling edges of the DFA with `[` and `]`, for instance, writing `[other]`. The meaning will be, that is a way to describe the "longest match" discipline. For instance, assume an automaton designed to accept a word defined as a sequence of letters; that could be described as $[a - zA - Z]$*, making use of ranges in ordered alphabets. The edge-label *other* will be used to abbreviate all "other" symbols. More technically, in a state with labels on outgoing edges labeled by some symbols, an outgoing edge labelled *other* represents all symbols *not* covered by the outgoing edges. It should be self-evident, especially for a determistic automaton, that can be only one outgoing edge labeled *other*. So far, that has nothing yet to do with the point discussed here, namely prioritizing longer matches. To do that, one uses edges, annotated with [and], as said. Note: it may be unfortunate, but the notation is meant to do something else than defining a letter as regular expression $[a - zA - Z]$*. What is meant then? Well, a transition labeled $[a]$ means: if a is next, move to the next state /without actually consuming or "eating" the a as input. Concretely, we use often transitions labelled $[other]$, and moving to an accepting state. That is the way to represent the longest match. We continue eating symbols, like lower and upper-case letters, but without accepting the string as yet. When we hit a symbol other than a letter, we proceed and accept the string, but the very last symbol is not part of the word we just processed.

Implementation of DFA (1)

Remember the DFA for a possible form of identifiers from earlier, in connection with equation (2.13). We had two versions, an incomplete one and *complete* one with an extra "error" state.



This one is *deterministic*, but it's *not* total or complete. The transition function is only *partial*. The "missing" transitions are often not shown (to 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 simply is not shown).

The [] around the transition *other* at the end means that the scanner does *not move forward* on the input there (but the automaton proceeds to the accepting state). That is something that is *not* 100% in the "mathematical theory" of FSA, but is how the *implementation* in the scanner will behave. Note also that the accepting state has changed: we have an extra state what we move to by the special kind of transition $[other]$. As the name

implies, "other" means all symbols different from the ones already covered by the other outgoing edges. This is used to realized the *longest prefix*: The shown DFA not just accepts "some" identifier it spots on the input, i.e., an arbitrary sequence of letters and digits (starting with a letter). No, it takes as many letters and digits as possible until it encounters a character *not* fitting the specification but not earlier. Only at that point does the automaton accepts but without advancing the input, in that this character will have to be scanned and classified as the "next chunk" and this "the next automaton".

Implementations

The following shows rather "sketchy" pseudo-code about how part of a lexer can be programmed or represented. It's one loop and it represents how to accept one lexeme. As mentioned at the beginning of this section, the task of a lexer in the context of a compiler is to *repeatedly* accept one lexeme after the other (or reject an input and stop) and hand over a corresponding stream of tokens. This need of repeated acceptance does not mean that there is another loop around the while-loop shown in the pseudo-code. At least the mentioned outermost loop is not part of the lexer. The lexer and the parser work hand in hand, and often that's arranged in that lexer works "on demand" from the parser: the parser invokes the lexer "give me a new token", the lexer has of course remembered the position in the input from the last invokation, and, starting from there tries to determine the next lexeme and, if successful, gives back the corresponding token to the parser. The parser determines if, at that point in the parsing process, the token fits to the syntactic description of the language, and if so, adds a next piece (at least implicitly) in building the parse tree, and then asks the lexer for the next token etc. The pseudo-codes of the lexer therefore contain only one loop, the one for accepting *one* word.

DFA implementation: 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;
```

The state is here represented as some integer variable. The reaction of the autmaton is a nested case switch, first one which state one is in, and secondly on which input. One could of course also do the "case nesting" the other way around, or making one flat case switch, with all combinations of state and input on the same level. We also see that the "error" state in the *complete* DFA is also represented in some form. At any rate, there is a else-case if the previous case(s) don't match. In the following slides, we show how the decision-information can be "centralized" in one table. In the table, empty slots represent missing reactions, i.e., the move to an error state.

Table rep. of the DFA

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

added 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;
```

2.5 NFA

Non-deterministic FSA

Actually, we have covered already non-deterministic finite state automata in Definition 2.3.1 (where we called it just FSA). Here we kind of repeat the definition, with δ slightly differently, but equivalently represented. What we add, however, are so-called ϵ-transitions,

which allows the machine to move to a new state without eating a letter. That is a form on "spontaneous" move, not being triggered by the input, which renders the automaton non-deterministic. It will turn out, that by adding this kind of transitions does not matter, as far the expressiveness of the NFAs is concerned. Why do we then bother adding them? Well, ϵ-transitions come in handy in some situations, in particular in the construction we will present afterwards: how to turn a regular expression into an NFA. It's slightly more conventient when one allows such transitions. It's easy to understand also why. As a preview to that construction: it will be a *compositional* construction. To construct the automton for a compound regular expression, for instance for the sequential composition $r_1 r_2$, one assumes one has the automata for the component regular expressions r_1 and r_2, and then one glues them together with ϵ-transitions, i.e., connects the accepting states of r_1 with the initial states of r_3. That's pretty easy, the use if those transitions facilitates a straightforward, *compositional* construction.

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
- ϵ: treated *different* from 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).

The version of NFA presented here includes ϵ-transitions. Depending on the source, and notion of NFA may or may not include such transitions. It does not matter anyhow, as far as the expressiveness is concerned.

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.*

Language of an NFA

- remember $\mathcal{L}(\mathcal{A})$ (Definition 2.3.3 on page 31)
- 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.

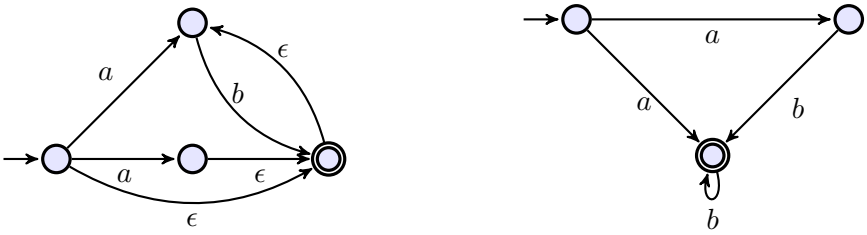
Definition 2.5.2 (Acceptance with ϵ-transitions)**.** A word w over alphabet Σ is *accepted* by an NFA with ϵ-transitions, if there exists a word w' which is accepted by the NFA with alphabet $\Sigma + \{\epsilon\}$ according to Definition 2.3.3 and where w is w' with all occurrences of ϵ removed.

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.

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*



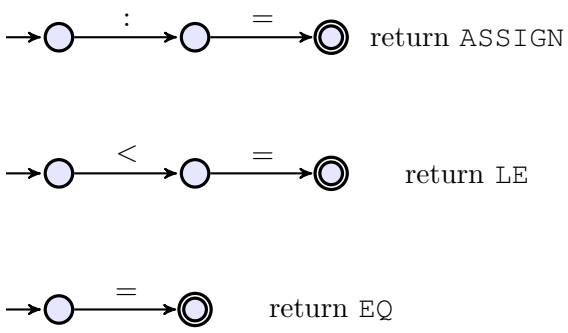
The example is used as illustration of an NFA and a corresponding DFA. In this small example, it's straightforward to come up with a deterministic version of the automaton. In a later section, we discuss a systematic way of turning an NFA to a DFA, i.e., an algorithm.

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

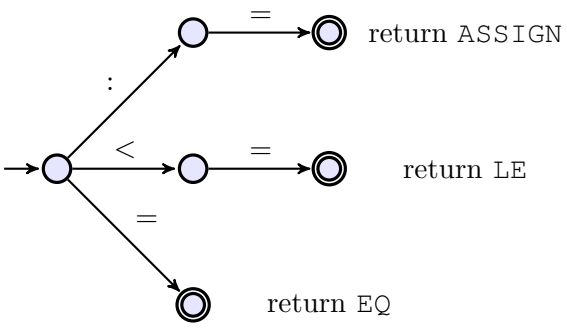
Before showing the construction itself, we show a few examples, highlighting some regular expression and corresponding NFAs.

Why non-deterministic FSA?

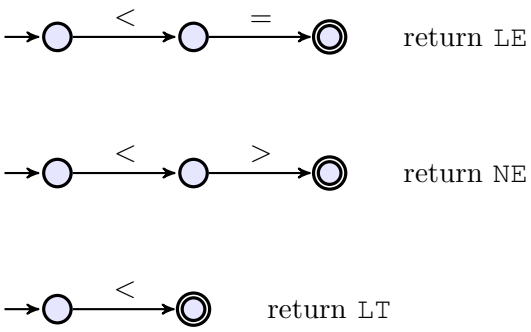
Task: recognize :=, <=, and = as three different tokens:



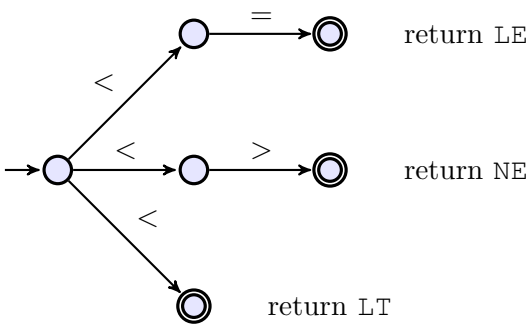
FSA (1-2)



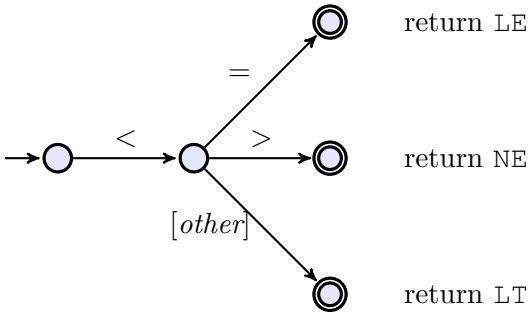
What about the following 3 tokens?



Non-det FSA (2-2)



Non-det FSA (2-3)



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 [6]

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[3] simpler, if one uses automata with one start and one accepting state

$\Rightarrow$ ample use of ϵ-transitions

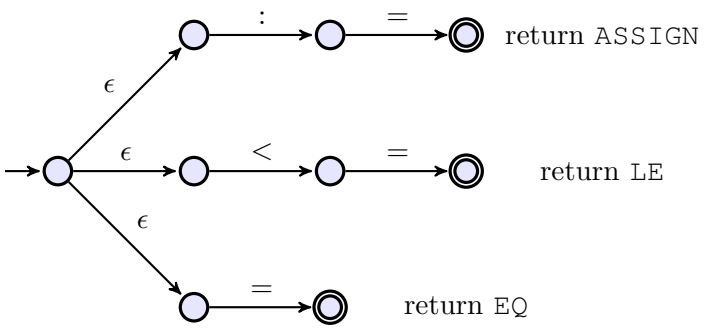
Compositionality

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 19). 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.* □

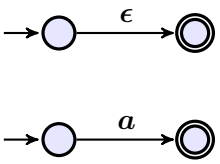
Illustration for ϵ-transitions



[3] It does not matter much, though.

Thompson's construction: basic expressions

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



The $\boldsymbol{\emptyset}$ is slightly odd: it's sometimes not part of regular expressions. We can see it as represented as the empty automaton (which has no states and which therefore was not drawn pictorially). 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, $\boldsymbol{\emptyset}$ should be part of the regular expressions, even if practically it does not play much of a role.

The representation of $\boldsymbol{\emptyset}$ as empty automaton is ok. If we do that, however, it's not the case that in Thompson's construction all automata have one start and one final state, the empty automaton would be an exception. It's not so obvious how to represent the empty language with an NFA with one initial and one accepting state (if one wants that invariant throughout). Note, the automaton with one state which is, at the same time, initial *and* accepting does not work: that accepts the language $\{\epsilon\}$. A state without accepting states sure represent the empty languge $\{\}$ but it violates the planned invariant, that there is exactly one final state. Trying two states, one the initial one and one the accepting one, but not connected by an edge does not work in the overall construction. If one constructed the automaton for $\boldsymbol{a\emptyset b}$ compositionally, there would be no connection from initial to final state.

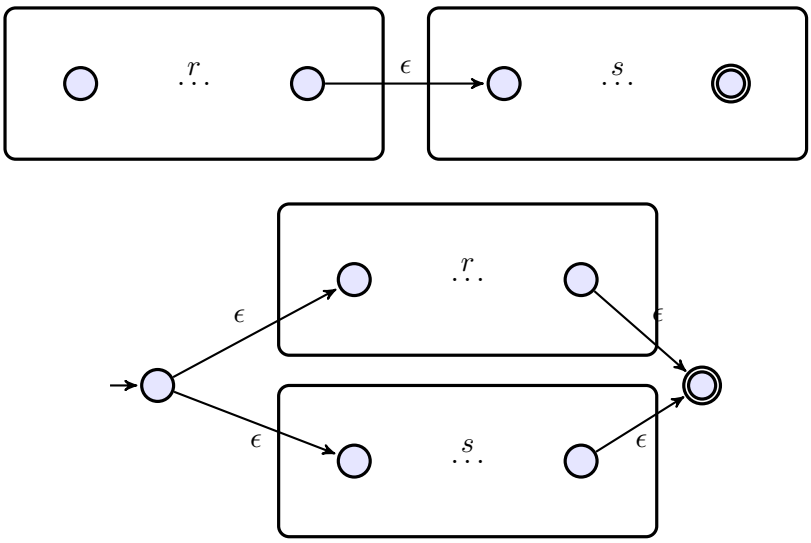
Therefore, the invariant that the construction maintains exactly one initial and one final state applies to all situations *except* the automaton for $\boldsymbol{\emptyset}$. That also means: the construction of compound automata illustrated the following is preseted by referring to "the" unique initual state of one or of two involved automata, and "the" unique accepting state of each of them. To cover also the case for $\boldsymbol{\emptyset}$, one would be more precise: connect *the* initial resp. *final* states *if they exist* in the following way.... (as shown in the pictures).

If that may seem overly nitpicking: if one wants to implement the algo (perhaps as part of lex), and if one supports $\boldsymbol{\emptyset}$ as syntax, one better takes care of all cases, including the corner cases, even if they seem not so relevant in practice, I mean, who actually has the need to use $\boldsymbol{\emptyset}$ in a lecture. However, ignoring them, assuming there is the invariant there there always one initial state and one final state, may derail the program, for instance leading to an uncaught nil-pointer exception (if one had stored the states using references). An alternative would be *not* to support $\boldsymbol{\emptyset}$, which is perfectly fine in practive, but then

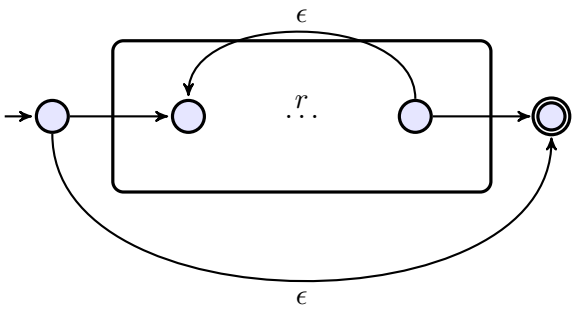
the "theoretician" will complain: There are then automata for which one cannot write a regular expression. So the proof of equivalence between automata and regular expression has corner case, which does not work...

Thompson's construction: compound expressions

In the picture, by convention, the state on the left is the unique initial one, the state on the right is the unique initial one (if they exist). By building the larger automaton, the "status" of the initial states and final states may changed, of course. For instance, in the case of |: a *new* initial state and a *new* accepting state is introduced for the automaton, but the initial and final states from the two component automata loose there special status, of course.



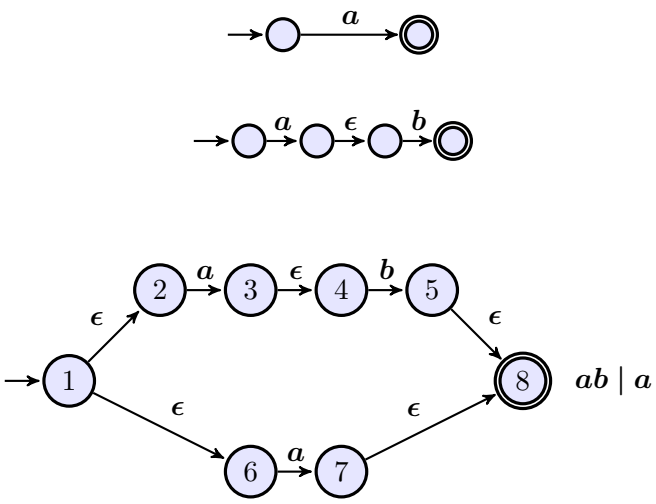
Thompson's construction: compound expressions: iteration



Example: $ab \mid a$

Intro

Here is a small example illustrating the construction. In the exercises, there will be more.



2.7 Determinization

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

- *powerset construction*
- origin of the construction: Rabin and Scott [5]

The construction, known also as *powerset* construction, seems straightforward enough. Analogous constructions works for some other kinds of automata, as well, but for still others, the approach does *not* work: For some forms of automata, non-deterministic versions are strictly more expressive than the deterministic one, for instance for some automata working with languages on infinite words, not finite words as here.

Some notation/definitions

Definition 2.7.1 (ϵ-closure, a-successors)**.** Given a state q, the *ϵ-closure* of q, written $close_\epsilon(q)$, 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.

We often call states like q, and sets of states then Q. So the notations for the ϵ-closure of a set Q of states is $close_\epsilon(Q)$ and Q_a represent the a-successors of Q.

Remark 9 (ϵ-closure)**.** *Louden [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 write 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 also be worth remarking: later, when it comes to parsing, we will encounter the phenomenon again: some steps done treating symbols from a context-free grammar will be done "eating symbols" (for parsing, those symbols will be called "terminals" or "terminal symbols" and correspond to tokens). Consequently, in the context of parsing and "parsing automata" (which are supposed to be deterministic as well), we will likewise encounter the notion of an ϵ-closure which is analogous to the concept here.

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

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

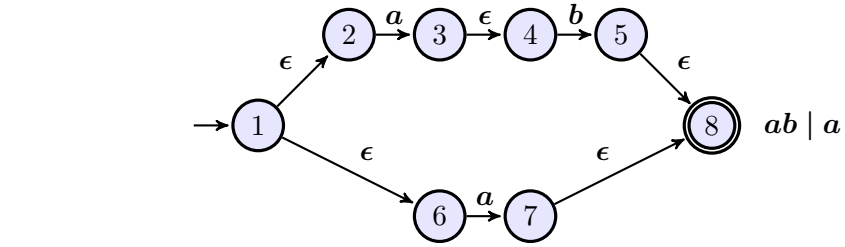
3. repeat step 2 for all states in $\overline{\mathcal{A}}$ and all $a \in \Sigma$, until no more states are being added
4. the *accepting* states in $\overline{\mathcal{A}}$: those containing *at least* one accepting state of $\mathcal{A}$

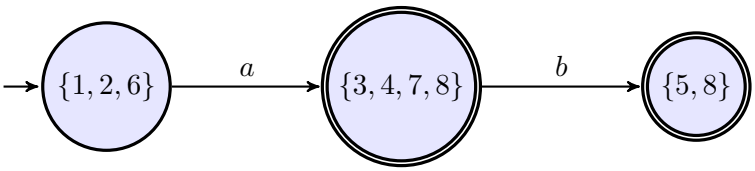
Note: the book Cooper and Torczon [1] uses a slightly more "concrete" formulation using a work-list. We will encounter work-list algos also elsewhere in the lecture (for instance, later in this chapter in minimization of automata, and also for liveness analysis at the end of the lecture, though we don't go into details. Very abstractly, a work-list is a data structure (like a list, more generally a collection), where one keeps "work still to be done". One works though the list, removing one piece after the other. It's characteristic for work-list algorithms, that one not only removes pieces of work, but doing some work may add other work, or re-adds a pieces of work done already. Often that connected to the traversal of graphs (which may contain cycles). A piece of work is "treating" a node of the graph (or an edge, depending on how it's organized). The node then is removed as

"done". However, perhaps the neighboring nodes are added, to be done. And since there are cycles, the node we just removed may be readded later. So, when connected to graphs like that, worklist algorithm are connected with traversals of the graph. And the "list" may be actually be a stack or a queue, or some other priorities, and that influences the traversal strategy (depth-first vs. breadth-first vs. other strategies). But as said, we don't dig deeper into that kind of algos, neither here nor in the examples later.

Next we show a few examples. More are covered by the exercises. In the figures, we show the resulting deterministic automata. However, we don't show the *complete* or total version, i.e., the extra state sometimes needed to obtain a total successor function is not shown. The state can be seen as being "marked" with the empty set {}

Example $ab \mid a$

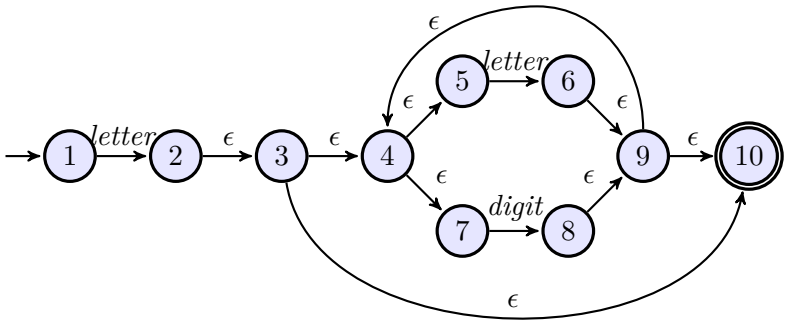




$ab \mid a$

Example: identifiers

Remember: regexpr for identifies from equation (2.13)



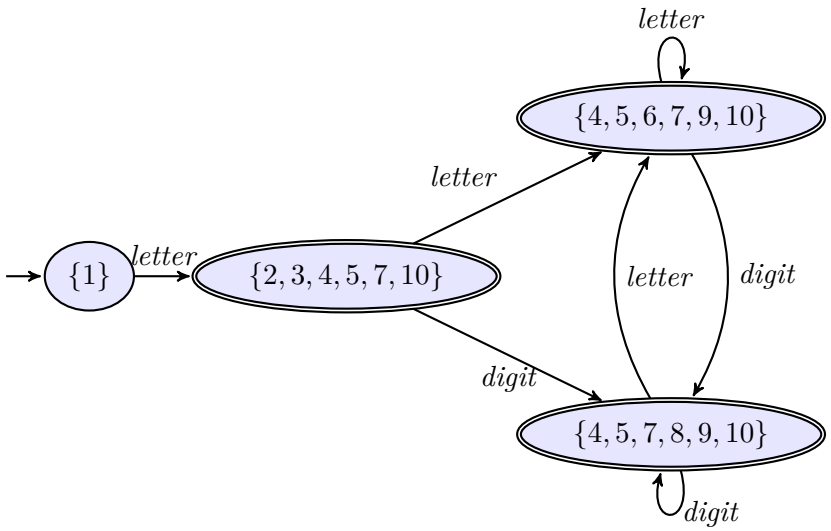


Figure 2.2: DFA for identifiers

Identifiers: DFA

2.8 Minimization

This then is the last stage of the construction, minimizing a DFA. It should be clear why that is useful: less states means more compact representation (but perhaps not necessarily speed-up in lexing). Minimal means, with the least amount of states. It's clear that there exists an automaton with the least amount of states. But what perhaps more surprising is: there exists *exactly one* automaton with the minimal number of states. A priori, there might be two different automata with a minimal amount of states, but that is not the case. Of course, being the same automaton means *up-to isorphism*. Isomorphic means "structually identical" basically it means, the "names" of the states don't matter, but otherwise the automata are the same; there is a one-to-one correspondance between states of both automata, that is honored by the transitions (and also the initial and final state sets).

We learn the algorithm that systematically calculates the minimal DFA from a given DFA. Previously, we downplayed the question, whether a DFA is complete or not, because if not complete, one can easily think of it as complete, assuming an extra error state. In the minimization here, it's important to indeed have a *complete* deterministic automaton, all states participate in the construction, including an extra error state that may need to be added to complete the DFA.

The one presented here is only one of different ways to achieve the goal. It's known as Hopcrofts's partitioning refinement algo.

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

Remarks

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

The algorithm will be explained to some extent in the following. The slides stated that it works "the other way around". I would expect, when tasked with the problem of minimizing a given DFA, most would try to approach the problem the following way. One would look at an the automaton and would find situations when there save some state. That's a natural way of thinking, also when one try concrete examples on pen and paper. For instance, one could look at the DFA for identifiers from Figure 2.2. It has three accepting states, but it does not take long to realize that only one is enough. One can "merge" the two states because the "do the same". By "doing the same" it's meant that both accept the same language, when one starts in them for accepting words. That language can be described by the regular expression (*letter* | *digit*)*. Collapsing the two (or three) states into one makes the automaton smaller without changing the accepted language. That could be a core step for an algorithm: hunt for a pair of equivalent states, collapse them, then then hunt for another pair and continue until only non-equivalent states remain, and then stop.

That's a valid idea. One would check some aspects before being sure it works. Termination is obvious. Another issue would be: is it important in which order to do the collapsing. It is a priori clear whether the algo would be independent from the strategy which pairs to collapse first. It could be that by choosing "wrong", one gets a smaller automaton, but somehow get stuck before reaching the really minimal one. That would be an unpleasant property of the approach (and would lead to backtracking). One also would have to solve the problem of checking when are two states equivalent (and that might be computationally complex).

But, as said, it's a valid idea (and I am rather sure, the approach would indepedent from the order of collapsing). The algorithm we describe below works the other way around in that it's not based on merging equivalent states, but by starting out with a "collapsed automaton", where all states are collapsed, and the splits them repeatedly (based on a criterion described later). The algo is not very obvious. In the merge-based, naive one, one starts with a DFA and in each steps, it gets smaller, but the algo maintains as invariant that all the intermediately constructed DFAs accept the same language as the original. Thereby it's clear that the result is likewise equivalent. And since we stop, when there are no more non-equivalent states, it's also plausible, that the result is minimal.

How is the situation in Hopcroft's algo, which works the other way around? Well, we start by some collapsed automaton. Being basically fully collapsed (with 2 states only), it will be generally *not* equivalent to the DFA, it accepts a differen (larger) language. It will also be *smaller* than the minimal one. The algorithm proceeds by splitting collapsed states as long as the splitting criterion is fulfilled. Note that all the intermediate DFA are non-equivalent to the targeted DFA and all of them are smaller than the minimal one. Once the splitting-criterion is no longer satisifed, one stops, and one has reach the *first* automaton in this process which, surprise, surprise, *is* equivalent to the targeted one, and at the same time is the minimal one.

That is the high-level idea of the algorithm. We have not intiutively explained the splitting-criterion. We will do that after we had a look at a "pseudo-code" description of the partitioning-refinement algo.

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)

The initialization, as mentioned before, starts with an (almost completely) collapsed automaton. It's not totally collapsed to a one-state representation, but consists of 2 states, no matter how big the original automaton is.

The algo speaks about *partitions* and operates by refining them. A partition is a technical term about sets, is splitting up a set into different (non-empty) subsets in such a way, that each element of the original set is in exactly *one* of the subsets, and the the union of all subsets is the original set. Alternatively (and equivalently), a partition on a set can be seens as equivalence relation on the set (an equivalence relation being a binary relation which is reflexive, transitive, and symmetric). We won't dig into mathematical depth here, so let's just illustrate it in a very examples. Assume a five-element set

$$A = \{1, 2, 3, 4, 5\} \ .$$

We can partition it into two subsets

$$\{\{1, 2, 3\}, \{4, 5\}\}$$

Let's call the two subsets A_1 and A_2.

Equivalently, one can see that partition as considering 1, 2, and 3 and "equivalent" and likewise 4 and 5. In other words, the partition corresponds to an equivalence relation. If one likes to spell the equivalence relation $\sim \subset A \times A$ in full detail as set of pairs, it would be

$$\sim = \{(1,1), (1,2), (2,1), (2,2), (1,3), (3,1), (2,3), (3,2), (3,3), (4,4), (4,5), (5,4), (5,5)\}$$

which corresponds to $A_1^2 + A_2^2$. Both views are interchangable. Seen as equivalence relation, one can also view the algorithm as refining a equivalence relation instead of a partitioning. Remember when discussing the naive "merging approach", we merged "non-equivalent" states. So, also there, we were working with an equivalence relation, what was meant there was semantical language equivalence. To states are equivalent, if they accept the same language, when starting acceptance runs from there.

Of course here, during the run of the automaton, the equivalence relation that corresponds to the current partition is *not* yet semantic language equivalence, it's a more coarse-grained equivalence relation, considering states currently as equivalent (grouped together in the same subset of the partition), when in fact, semantically, they are not equivalent. When the algo stops, though, the equivalence relation coincides with the intended langugue equivalence.

Here, we are working with partitons of the set of states of the given DFA, and we start with a partition, consisting of to subsets: the set of states is split into two parts: the accepting states and the non-accepting states. The algorithm works in one direction only: namely by taking a subset, i.e., one element in the current partition of Q, and splits that, if needed. The partition gets more finegrained with each iteration, until no more splitting can be done.

If one looks at some partition during the run of the algo, one can conceptually *interpret* the partition as an automaton: Each subset of the partition forms some "meta-state" consisting of sets of states, and there are transitions between those meta-states in the obvious. In this way, the algo not just steps through a sequence of partitions it refines, but at least conceptually, to a sequence of automata. This is a way of "thinking" about the run of the algo, the algo itself does not explicitly construct sequences of automata, it works on a sequence of partitions that gets more and more finegrained during the run.

However, thinking in terms of intermediate automata helps to interpret the splitting-condition: when (and how) should the algo split a meta-state, and when can it stop. As mentioned earlier, starting from the initial 2-state automaton, the intermediate automata are generally smaller than the minimal one, and they are accept a language different from the one of the target automaton (a larger language, actually). There is a third aspect, not mentioned so far: at and intermediate stage, the automaton with the meta-states is generally *non-deterministic*. It's clear that if one takes a DFA and collapses some states into one meta-state, the result will no longer be deterministic. That is also the splitting-condition. The algo looks at meta-states (i.e., a subset in the current partition) and if that meta-state violates the requirement that it should be *determistic*, then it splits it. Actually, the algo checks whether or not a meta-state is determistic per *symbol*, i.e., the algo checks where some meta-state Q and a symbol a behave deterministically or not.

If the meta-state behaves non-deterministically, we have to repair that, and that's by splitting the that meta-state, so the the resulting split behaves deterministically (with respect to that symbol a). Howerver, we split only as much as we need to repair the non-determistic violation, but not more. For instance, one does not simply "atomize" the meta-states into its individual original states. Those would surely behave deterministic as the starting point had been DFA, but this way, we won't get the minimal automaton in general, as we would do more splits than actually necessary.

So far so good. Of course, one need to treat more than a, it may be necessary, after splitting a meta-state wrt. a, that one need to split the result further wrt. b. That's clear, and let's not talk about that, let's focus on one symbol. More interesting is the question: after having split one meta-state in the way described, making the fragments deterministic, am I done with the fragments, or will I have to split them further? The answer is: doing it one time may not be enough. The reason is as follows. Splitting a meta-state in the way described may have a *rippling effect* on other meta-states. For instance, if one has a stituation like

$$Q \xrightarrow{a} Q'$$

and the meta-state Q' happens to be split in, say, two refined meta-states Q'_1 and Q'_2, then the predecessor state Q suddenly has 2 outgoing a-transitions even if we assume that sometimes earlier, Q was the result of some splitting step, making it determistic at that earlier point. That mean, splitting a state may affect that other states have to be split again, that is the mentioned rippling effect.

A good way to organize the splitting task is to put all the current meta-states that have not been checked wether they need a split or not into a *work list*. It may not technically be a list, but could be a queue or stack, or in general a collection data type, but the

algo would still be called worklist algorithm. Anyway, with this data structure, one can remove one piece of work, a current meta state out from the work-list, splits it, if necessary, removes the piece of work, but (re-)adds predecessor states, as they need to be rechecked and re-treated.

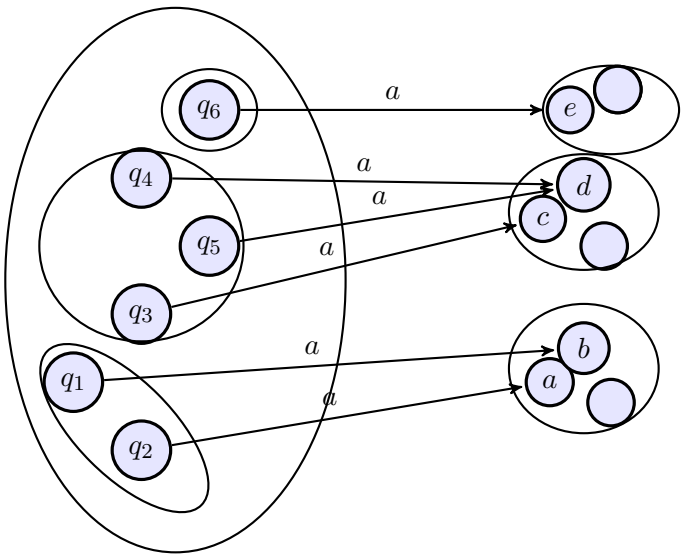
Partition refinement vs. merging equivalent states We started earlier by claiming that a naive approach would probably try to *merge* equivalent states starting from the given DFA (with would be a "partition coarsening"), as that seems more obvious. Now, why is the partition refinement algo intuitvely a better idea (without going into algorithmic complexity considerations)?

In a way, the two approaches (refinement vs. coarseing) look pretty similar. One merges states resp. split states, until no more merging resp. splitting is neccessary, and then stops. It's also not easy to say, which is the shorter route, i.e., which approach needs on average the least amount of iterations (perhaps in the special case where the automaton comes via Thompson's construction and determinzation).

There *is* a significant difference, though, that that's the condition to decide when to stop (resp. if still merging resp. splitting is necessary). In Hopcrof's refinement approach, the check is *local*. The condition concerns *the next single edges originating in a (meta)-state*. If they violate the determinism-requirement: then split, otherwise not.

The condition on the merging approach is *not-local*. They require to check wether to states accept the same language. That cannot be checked by the looking one step ahead, checking the outgoing edges. That involves checking all reachable states, and is a much more complicated condition. Perhaps some memoization (remembering and caching (partial) earlier checks) can help a bit, but Hopcroft's partitioning refinment seems not only more clever, it looks also superior.

Split in partition refinement: basic step



- before the split $\{q_1, q_2, \ldots, q_6\}$

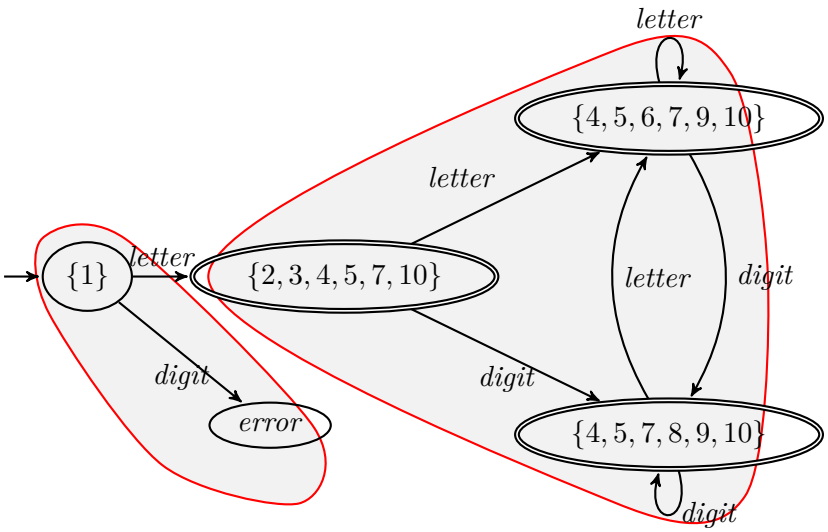
- after the split on a: $\{q_1, q_2\}, \{q_3, q_4, q_5\}, \{q_6\}$

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

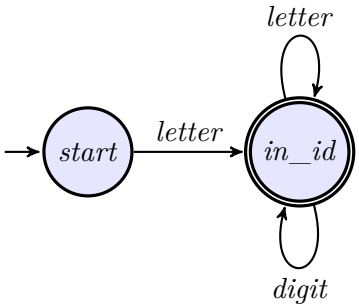
Examples

The following examples are shown in overlays in the slides. They unfolging of the overlays is not done for the script version here.

Completed automaton

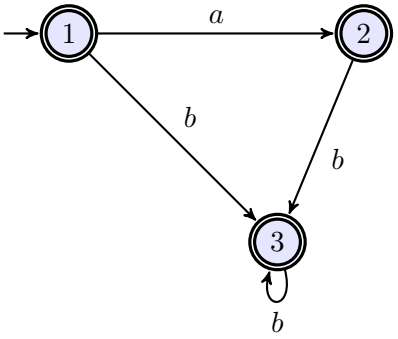


Minimized automaton (error state omitted)



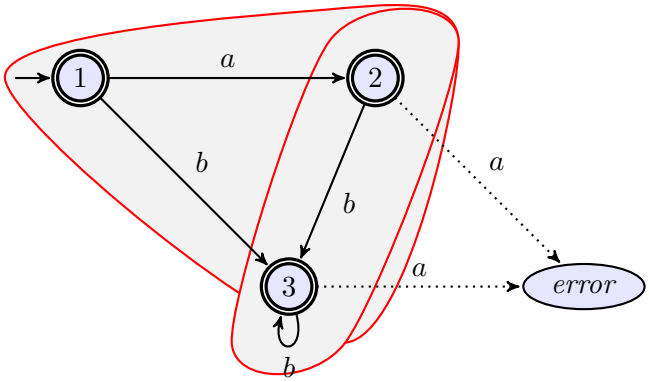
Another example: partition refinement & error state

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

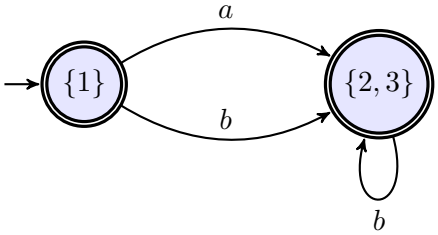


Partition refinement

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.

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 (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)

Tokens and actions of a parser will be covered later. For example, identifiers and digits as described by the regular 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)

```
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.

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

Index