Compiler Construction

Christian Rinderknecht

31 October 2008

Why study compiler construction?

Few professionals design and write compilers.

So why teach how to make compilers?

- A good software/telecom engineer understands the high-level **languages** as well as the **hardware**.

  A compiler links these two aspects.

- That is why understanding the compiling techniques is understanding the interaction between the programming languages and the computers.

- Many applications embed small languages for configuration purposes or make their control versatile (think of macros, scripts, data description etc.)

Why study compiler construction? (cont)

The techniques of compilation are necessary for implementing such languages.

Data formats are also formal languages (languages to specify data), like HTML, XML, ASN.1 etc.

The compiling techniques are mandatory for reading, treating and writing data but also to port (migrate) applications (re-engineering). This is a common task in companies.

Anyway, compilers are excellent examples of complex software systems

- which can be rigorously specified,

- which only can be implemented by combining theory and practice.

Function of a compiler

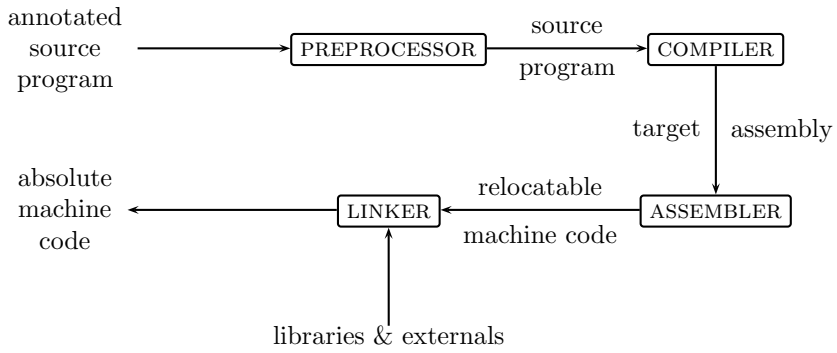
The function of a compiler is to **translate** texts written in a **source language** into texts written in a **target language**.

Usually, the source language is a **programming language**, and the corresponding texts are **programs**. The target language is often an **assembly language**, i.e. a language closer to the machine language (it is the language understood by the processor) than the source language.

Some programming languages are compiled into a **byte-code language** instead of assembly. Byte-code is usually not close to any assembly language. Byte-code is **interpreted** by another program, called **virtual machine** (**VM**), instead of being translated to machine language (which is directly executed by the machine processor): the VM processes the instructions of the byte-code.

Compilation chain

From an engineering point of view, the compiler is one link in a chain of tools:



Compilation chain (cont)

Let us consider the example of the **C language**. A famous free compiler is GNU GCC.

In reality, GCC includes the complete compilation chain, not just a C compiler:

- to only preprocess the sources: `gcc -E prog.c` (standard output) Annotations are introduced by `#`, like `#define x 6`
- to preprocess and compile: `gcc -S prog.c` (output `prog.s`)
- to preprocess, compile and assemble: `gcc -c prog.c` (output `prog.o`)
- to preprocess, compile, assemble and link: `gcc -o prog prog.c` (output `prog`) Linkage can be directly called using `ld`.

The analysis-synthesis model of compilation

In this class we shall detail only the compilation stage itself.

There are two parts to compilation: **analysis** and **synthesis**.

1. The analysis part breaks up the source program into constituent pieces of an **intermediary representation** of the program.
2. The synthesis part constructs the target program from this intermediary representation.

In this class we shall restrict ourselves to the analysis part.

Analysis

The analysis can itself be divided into three successive stages:

1. **linear analysis,** in which the stream of characters making up the source program is read and grouped into **lexemes** that are sequences of characters having a collective meaning; sets of lexemes with a common interpretation are called **tokens**;
2. **hierarchical analysis,** in which tokens are grouped hierarchically into nested collections (**trees**) with a collective meaning;
3. **semantic analysis,** in which certain checks are performed to ensure the components of a program fit together meaningfully.

In this class we shall focus on linear and hierarchical analysis.

Lexical analysis

In a compiler, linear analysis is called **lexical analysis** or **scanning**.

During lexical analysis, the characters in the assignment statement

```
position := initial+rate*60
```

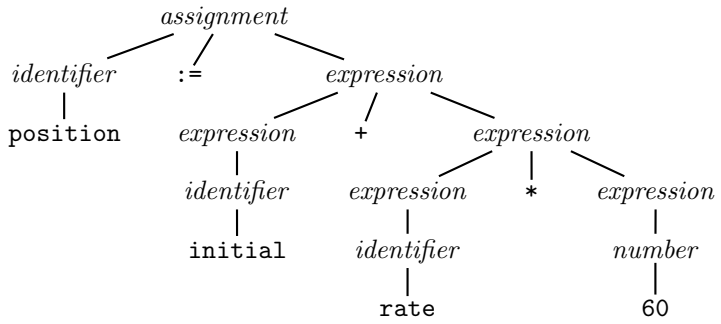
would be grouped into the following lexemes and tokens (see facing table).

The blanks separating the characters of these tokens are normally eliminated.

Token	Lexeme
identifier	`position`
assignment symbol	`:=`
identifier	`initial`
plus sign	`+`
identifier	`rate`
multiplication sign	`*`
number	`60`

Syntax analysis

Hierarchical analysis is called **parsing** or **syntax analysis**. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source are represented by a **parse tree** such as:



Syntax analysis (cont)

In the expression

```
initial + rate * 60
```

the phrase

```
rate * 60
```

is a logical unit because the usual conventions of arithmetic expressions tell us that multiplication is performed prior to addition.

Thus, because the expression

```
initial + rate
```

is followed by a `*`, it is **not** grouped into the same subtree.

Syntax analysis (cont)

The hierarchical structure of a program is usually expressed by **recursive rules**. For instance, an expression can be defined by a set of cases:

1. Any *identifier* is an expression.
2. Any *number* is an expression.
3. If *expression*$_1$ and *expression*$_2$ are expressions, then so are
   (a) *expression*$_1$ `+` *expression*$_2$
   (b) *expression*$_1$ `*` *expression*$_2$
   (c) `(` *expression*$_1$ `)`

Syntax analysis (cont)

Rule 1 and 2 are non-recursive base rules, while the others define expressions in terms of operators applied to other expressions.

`initial` and `rate` are identifiers.

Therefore, by rule 1, `initial` and `rate` are expressions.

`60` is a number.

Thus, by rule 2, we infer that `60` is an expression.

Then, by rule 3b, we infer that `rate * 60` is an expression.

Thus, by rule 3a, we conclude that `initial + rate * 60` is an expression

Syntax analysis (cont)

Similarly, many programming languages define statements recursively by rules such as

1. If *identifier* is an identifier and *expression* is an expression, then

   identifier `:=` *expression*

   is a statement.
2. If *expression* is an expression and *statement* is a statement, then

   `while (`*expression*`) do` *statement* `if (`*expression*`) then` *statement*

   are statements.

Syntax analysis (cont)

The division between lexical and syntactic analysis is somewhat arbitrary.

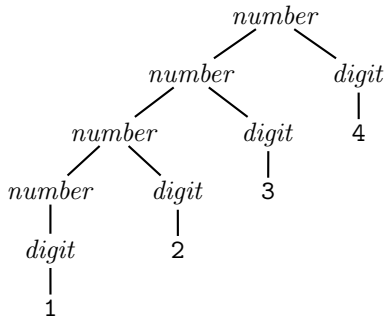
For instance, we could define the integer numbers by means of recursive rules:

1. a *digit* is a *number* (base rule),
2. a *number* followed by a *digit* is a *number* (recursive rule).

Imagine now that the lexer does **not** recognise numbers, just digits. The parser therefore uses the previous recursive rules to group in a parse tree the digits which form a number.

Syntax analysis (cont)

For instance, the parse tree for the number `1234`, following these rules, would be



But notice how this tree actually is almost a list.

The structure, i.e. the embedding of trees, is indeed not meaningful here.

For example, there is no obvious meaning to the separation of `12` (same subtree at the leftmost part) in the number `1234`.

Syntax analysis (cont)

Therefore, pragmatically, the best division between the lexer and the parser is the one that simplifies the overall task of analysis.

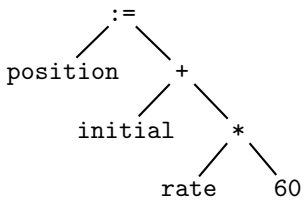
One factor in determining the division is whether a source language construct is inherently recursive or not: lexical constructs do not require recursion, while syntactic construct often do.

For example, recursion is not necessary to recognise identifiers, which are typically strings of letters and digits beginning with a letter: we can read the input stream until a character that is neither digit nor letter is found, then these read characters are grouped into an identifier token.

On the other hand, this kind of linear scan is not powerful enough to analyse expressions or statements, like matching parentheses in expressions or `{` and `}` in block statements: a nesting structure is needed.

Syntax analysis (cont)

The parse tree page 5 describes the syntactic structure of the input. A more common *internal* representation of this syntactic structure is given by



An **abstract syntax tree** (or just **syntax tree**) is a compressed version of the parse tree, where only the most important elements are retained for the semantic analysis.

Semantic analysis

The semantic analysis checks the syntax tree for meaningless constructs and completes it for the synthesis.

An important part of semantic analysis is devoted to **type checking**, i.e. checking properties on how the data in the program is combined.

For instance, many programming languages require an error to be issued if an array is indexed with a floating-point number (called *float*).

Some languages allow such floats and integers to be mixed in arithmetic expressions. Some do not (because internal representation of integers and floats is very different, as well as the cost of the corresponding arithmetic functions).

Semantic analysis (cont)

In our example, page 8, assume all identifiers were declared as floats.

The type-checking compares the type of `rate`, which is float, and of `60`, which is integer. Let us assume our language allows these two types of operands for the multiplication `*`.

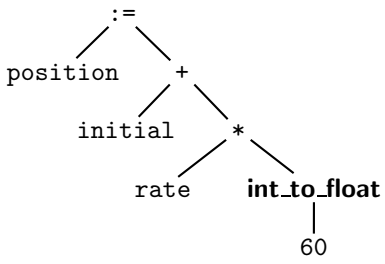
Then the analyser must insert a special node in the syntax tree which represents a **type cast** from integer to float for `60`.

At the level of the programming language, a type cast is the identity function (in mathematics: $x \mapsto x$), so the value is not changed, but the type of the result is different from the type of the argument.

This way the synthesis will know that the assembly code for such a conversion has to be output.

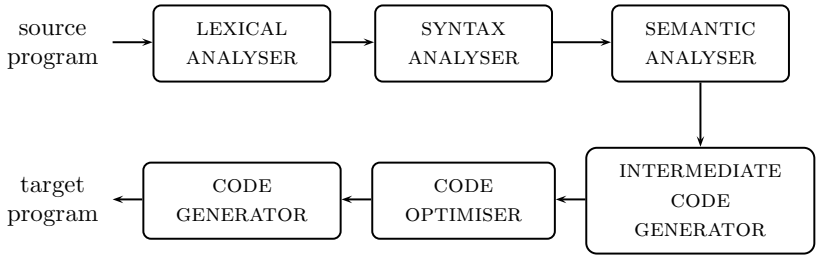
Semantic analysis (cont)

Hence the semantic analysis issues no error and produces the following **annotated syntax tree** for the synthesis:



Phases

Conceptually, a compiler operates in **phases**, each of which transforms the program from one representation to another. A typical decomposition of a compiler is as follows:



The first row makes up the analysis and the second the synthesis.

Phases/Symbol table

The previous figure did not depict another component which is connected to all the phases: the **symbol table manager**. A symbol table is a two-column table whose first column contains identifiers collected in the program and the second column contains any interesting information, called **attributes**, about their corresponding identifier. Example of identifier attributes are

- the allocated storage,
- the type,
- the **scope** (i.e. where in the program it is valid),
- in case of procedures names, the number and type of the parameters, the method of passing each argument (e.g., by reference) and the result type, if any.

Phases/Symbol table (cont)

When an identifier in the source program is detected by the lexical analyser (or simply called **lexer**), this identifier is entered into the symbol table.

However, some attributes of an identifier cannot normally be determined during lexical analysis (or simply called **lexing**). For example, in a Pascal declaration like

```
var position, initial, rate: real;
```

the type `real` is not known when `position`, `initial` and `rate` are recognised by the lexical analyser.

The remaining phases enter information about the identifiers into the symbol table and use this information. For example, the semantic analyser needs to know the type of the identifiers to generate intermediate code.

Phases/Error detection and reporting

Another compiler component that was omitted from picture page 9 because it is also connected to all the phases is the **error handler**.

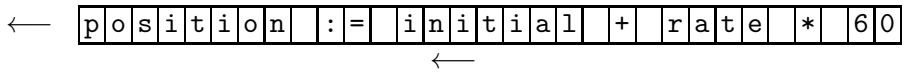
Indeed, each phase can encounter errors, so each phase must somehow deal with these errors. Here come some examples.

- Lexical analysis finds an error if a series of characters do not form a token.

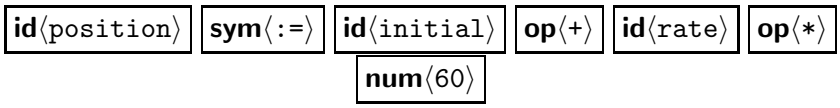
- Syntax analysis finds an error if the relative position of a group of tokens is not described by the grammar (syntax).
- Semantic analysis finds an error if the program contains the addition a an integer and an array.

Phases/The analysis phase/Lexing

Let us consider again the analysis phase and its sub-phases in more details, following a previous example. Consider the next character string



First, as we stated page 4, the lexical analysis recognises the tokens of this character string (which can be stored in a file). The output of the lexing is a stream of tokens like



where **id** (*identifier*), **sym** (*symbol*), **op** (*operator*) and **num** (*number*) are the token names and between brackets are the **lexemes**.

Phases/The analysis phase/Lexing (cont)

The lexer also outputs or updates a symbol table like[1]

Identifier	Attributes
`position`	...
`initial`	...
`rate`	...

The attributes often include the position of the corresponding identifier in the original string, like the position of the first character either counting from the start of the string or through the line and column numbers.

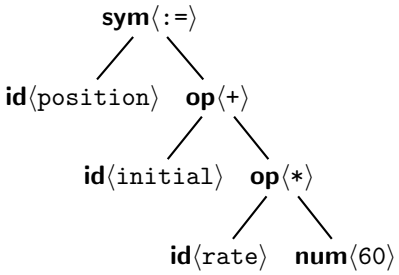
[1] Even if the table is named "symbol table" it actually contains information about identifiers only.

Phases/The analysis phase/Parsing

The parser takes this token stream and outputs the corresponding syntax tree and/or report errors.

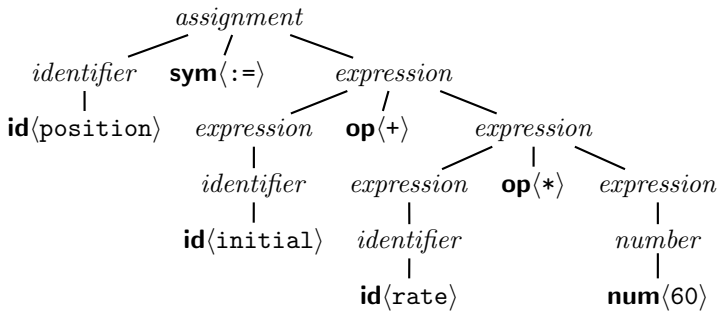
Page 8, we gave a simplified version of this syntax tree. A refined version is given in the facing column.

Also, if the language requires variable definitions, the syntax analyser can complete the symbol table with the type of the identifiers.



Phases/The analysis phase/Parsing (cont)

The parse tree can be considered as a **trace** of the syntax analysis process: it summarises all the recognition work done by the parser. It depends on the syntax rules (i.e. the grammar) and the input stream of tokens.



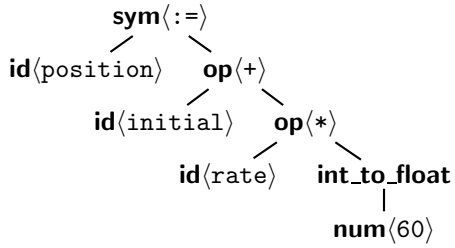
Phases/The analysis phase/Semantic analysis

The semantic analysis considers the syntax tree and checks certain properties depending on the language, typically it makes sure that the valid syntactic constructs also have a certain meaning (with respect to the rules of the language).

We saw page 9 that this phase can annotate or even add nodes to the syntax tree. It can as well update the symbol table with the information newly gathered in order to facilitate the code generation and/or optimisation.

Phases/The analysis phase/Semantic analysis (cont)

Assuming that our toy language accepts that an integer is mixed with floats in arithmetic operations, the semantic analysis can insert a type cast node. A new version of the annotated syntax tree would be:



Note that the new node is not a token, just a (semantic) tag for the code generator.

Phases/The synthesis phase

The purpose of the synthesis phase is to use all the information gathered by the analysis phase in order to produce the code in the target language.

Given the annotated syntax tree and the symbol table, the first sub-phase consists in producing a program in some artificial, intermediary, language.

Such a language should be independent of the target language, while containing features common to the *family* the target language belongs to.

For instance, if the target language is the PowerPC G4 microprocessor, the intermediary language should be like an assembly of the IBM RISC family.

Phases/The synthesis phase (cont)

If we want **to port a compiler** from one platform to another, i.e., make it generate code for a different OS or processor, such intermediary language comes handy: if the new platform share some features with the first one, we only have to rewrite the code generator component of the compiler — not the whole compiler.

It may be interesting to have the same intermediary language for different source languages, allowing the sharing of the synthesis.

We can think of an intermediary language as an assembly for an **abstract machine** (or processor). For instance, our example could lead to the code

```
temp1 := inttoreal(60)
temp2 := id_rate * temp1
temp3 := id_initial + temp2
id_position := temp3
```

Phases/The synthesis phase (cont)

Another point of view is to consider the intermediary code as a tiny subset of the source language, as it retains some high-level features from it, like, in our example, variables (instead of explicit storage information, like memory addresses or register numbers), operator names etc.

This point of view enables optimisations that otherwise would be harder to achieve (because too many aspects would depend closely on many details of the target architecture).

Phases/The synthesis phase (cont)

This kind of assembly is called **three-address code**. It has several properties:

- each instruction has at most one operator (in addition to the assignment);
- each instruction can have at most three operands;
- some instructions can have less than three operands (e.g. the first and last instruction);
- the result of an operation must be linked to a variable;

As a consequence, the compiler must order well the code for the sub-expressions, e.g. the second instruction must come before the third one because the multiplication has priority on addition.

Phases/The synthesis phase/Code optimisation

The code optimisation phase attempts to improve the intermediate code, so that faster-running target code will result.

The code optimisation produces intermediate code: the output language is the same as the input language.

For instance, this phase would find out that our little program would be more efficient this way:

```
temp1 := id_rate * 60.0
id_position := id_initial + temp1
```

This simple optimisation is based on the fact that type casting can be performed at compile-time instead of run-time, but it would be an unnecessary concern to integrate it in the code generation phase.

Phases/The synthesis phase/Code generation

The code generation is the last phase of a compiler. It consists in the generation of target code, usually relocatable assembly code, from the optimised intermediate code.

A crucial aspect is the assignment of variables to registers.

For example, the translation of code page 15 could be

```
MOVF id_rate, R2
MULF #60.0, R2
MOVF id_initial, R1
ADDF R2, R1
MOVF R1, id_position
```

The first and second operands specify respectively a source and a destination.

The `F` in each instruction tells us that the instruction is dealing with floating-point numbers.

Phases/The synthesis phase/Code generation (cont)

This code moves the contents of the address `id_rate` into register `2`, then multiplies it with the float 60.0.

The `#` signifies that `60.0` is a constant.

The third instruction moves `id_initial` into register `1` and adds to it the value previously computed in register `2`.

Finally, the value in register `1` is moved to the address of `id_position`.

Implementation of phases into passes

An implementation of the analysis is called a **front-end** and an implementation of the synthesis **back-end**.

A **pass** consists in reading an input file and writing an output file.

It is possible to group several phases into one pass in order to interleave their activity.

- On one hand, this can lead to a greater efficiency since interactions with the file system are much slower than with internal memory.
- On the other hand, this architecture leads to a greater complexity of the compiler — something the software engineer always fears.

Implementation of phases into passes (cont)

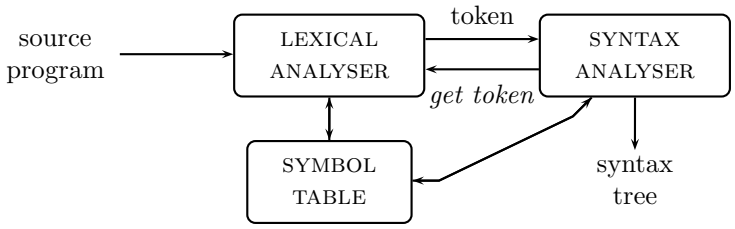
Sometimes it is difficult to group different phases into one pass.

For example, the interface between the lexer and the parser is often a single token. There is not a lot of activity to interleave: the parser requests a token to the lexer which computes it and gives it back to the parser. In the meantime, the parser had to wait.

Similarly, it is difficult to generate the target code if the intermediate code is not fully generated first. Indeed, some languages allow the use of variables without a prior declaration, so we cannot generate immediately the target code because this requires the knowledge of the variable type.

Lexer

The lexical analyser is the first phase of a compiler. Its main task is to read the input characters and produce a sequence of tokens that the syntax analyser uses.



Upon receiving a request for a token (*get token*) from the parser, the lexical analyser reads input characters until a lexeme is identified and returned to the parser together with the corresponding token.

Lexer (cont)

Usually, a lexical analyser is in charge of

- stripping out from the source program **comments** and **white spaces**, in the form of blank, tabulation and newline characters;
- keeping trace of the position of the lexemes in the source program, so the error handler can refer to exact positions in error messages.

Lexer/Tokens, lexemes, patterns

A **token** is a set of strings which are interpreted in the same way, for a given source language. For instance, **id** is a token denoting the set of all possible identifiers.

A **lexeme** is a string belonging to a token. For example, `5.4` is a lexeme of the token **num**.

The tokens are defined by means of **patterns**. A pattern is a kind of compact rule describing all the lexemes of a token. A pattern is said to *match* each lexeme in the token.

For example, in the Pascal statement

```
const pi = 3.14159;
```

the substring `pi` is a lexeme for the token **id** (*identifier*).

Lexer/Tokens, lexemes, patterns (cont)

Token	Sample lexemes	Informal pattern
id	`pi count D2 ...`	letter followed by letters and digits
relop	`< <= = >= >`	`<` or `<=` or `<` or `=` or `>=` or `>`
const	`const`	`const`
if	`if`	`if`
num	`3.14 4 .2E2 ...`	any numeric constant
literal	`"message" "" ...`	any characters between " and " except "

Lexer/Tokens, lexemes, patterns (cont)

Most recent programming languages distinguish a finite set of strings that match the identifiers but are not part of the identifier token: the **keywords**.

For example, in Ada, `function` is a keyword and, as such, is not a valid identifier.

In C, `int` is a keyword and, as such, cannot be used an identifier (e.g. to declare a variable).

Nevertheless, it is common **not** to create explicitly a **keyword** token and let each keyword lexeme be the only one of its own token, as displayed in the table page 18.

Specification of tokens

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for sets of strings.

Strings and formal languages

The term **alphabet** denotes any finite set of symbols. Typical examples of symbols are letters and digits. The set $\{0, 1\}$ is the *binary alphabet*. ASCII is an example of computer alphabet.

Specification of tokens (cont)

A **string** over some alphabet is a finite sequence of symbols drawn from that alphabet. The terms **sentence** and **word** are often used as synonyms.

The length of string s, usually noted $|s|$, is the number of occurrences of symbols in s. For example, `banana` is a string of length six. The **empty string**, denoted ε, is a special string of length zero.

Specification of tokens/Strings and formal languages (cont)

Term	Informal definition
prefix of s	A string obtained by removing zero or more trailing symbols of string s; e.g. `ban` is a prefix of `banana`.
suffix of s	A string formed by deleting zero or more of the leading symbols of s; e.g. `nana` is a suffix of `banana`.
substring of s	A string obtained by deleting a prefix and a suffix from s; e.g. `nan` is a substring a `banana`. Every prefix and every suffix of s is a substring s, but not every substring of s is a prefix or a suffix of s. For every string s, both s and ε are prefixes, suffixes and substrings of s.

Specification of tokens/Strings and formal languages (cont)

Term	Informal definition
proper prefix, suffix or substring of s	Any non-empty string x that is, respectively, a prefix, suffix, substring of s such that $s \neq x$; e.g. ε and `banana` are **not** proper prefixes of `banana`.
subsequence of s	Any string formed by deleting zero or more not necessarily contiguous symbols from s; e.g. `baaa` is a subsequence of `banana`.

Specification of tokens/Strings and formal languages (cont)

The term **language** denotes any set of strings over some fixed alphabet.

The **empty set**, noted $\varnothing$, or $\{\varepsilon\}$, the set containing only the empty word are languages. The set of valid C programs is an infinite language.

If x and y are strings, then the **concatenation** of x and y, written xy or $x \cdot y$, is the string formed by appending y to x.

For example, if $x =$ `dog` and $y =$ `house`, then $xy =$ `doghouse`.

The empty string is the identity element under concatenation: $s\varepsilon = \varepsilon s = s$.

Specification of tokens/Strings and formal languages (cont)

If we think of concatenation as a product, we can define string exponentiation as follows.

- $s^0 = \varepsilon$
- $s^n = s^{n-1}s$, if $n > 0$.

Since $\varepsilon s = s$, $s^1 = s$, then $s^2 = ss$, $s^3 = sss$ etc.

Specification of tokens/Strings and formal languages (cont)

We can now revisit the definitions we gave in table page 19 and 19 using a formal notation. Let L be the language under consideration.

Term	Formal definition
x is a *prefix* of s	$\exists y \in L.s = xy$
x is a *suffix* of s	$\exists y \in L.s = yx$
x is a *substring* of s	$\exists u, v \in L.s = uxv$
x is a *proper prefix* of s	$\exists y \in L.y \neq \varepsilon$ and $s = xy$
x is a *proper suffix* of s	$\exists y \in L.y \neq \varepsilon$ and $s = yx$
x is a *proper substring* of s	$\exists u, v \in L.uv \neq \varepsilon$ and $s = uxv$

Specification of tokens/Operations on languages

It is possible to define operation in languages. For lexical analysis, we are interested mainly in **union**, **concatenation** and **closure**. Let L and M be two languages.

Operation	Formal definition
union of L and M	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
concatenation of L and M	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$ where $L^0 = \{\varepsilon\}$
positive closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

In other words, L^* means "zero or more concatenations of L", and L^+ means "one or more concatenations of L."

Specification of tokens/Operations on languages/Examples

Let $L = \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$ and $D = \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}\}$.

1. L is the alphabet consisting of the set of upper and lower case letters and D is the alphabet of the decimal digits.

2. Since a symbol is a string of length one, the sets L and D are finite languages too.

These two ways of considering L and D and the operations on languages allow us to create new languages from other languages defined by their alphabet.

Here are some examples of new languages created from L and D:

- $L \cup D$ is the language of letters and digits.
- LD is the language whose words consist of a letter followed by a digit.

Specification of tokens/Operations on languages/Examples (cont)

- L^4 is the language whose words are four-letter strings.
- L^* is the language made up on the alphabet L, i.e. the set of all strings of letters, including the empty string ε.
- $L(L \cup D)^*$ is the language whose words consist of letters and digits and beginning with a letter.
- D^+ is the language whose words are made of one or more digits, i.e. the set of all decimal integers.

Regular expressions

In Pascal, an identifier is a letter followed by zero or more letters or digits, that is, and identifier is a member of the set defined by $L(L \cup D)^*$.

The notation we introduced so far is comfortable for mathematics but not for computers. Let us introduce another notation, called **regular expressions**, for describing the same languages and define its meaning in terms of the mathematical notation.

With this notation, we might define Pascal identifiers as

$$\textbf{letter (letter | digit)}^\star$$

where the vertical bar means "or", the parentheses group subexpressions, the star means "zero or more instances of" the previous expression and juxtaposition means concatenation.

Regular expressions (continued)

A regular expression r is built up out of simpler regular expressions using a set of rules, as follows. Let Σ be an alphabet and $L(r)$ the language denoted by r.

1. ϵ is a regular expression that denotes $\{\varepsilon\}$.
2. If $a \in \Sigma$, then a is a regular expression that denotes $\{a\}$. This is ambiguous: a can denote a language, a word or a letter — it depends on the context.
3. Assume r and s denote the languages $L(r)$ and $L(s)$; a denotes a letter. Then
   (a) $r \mid s$ is a regular expression denoting $L(r) \cup L(s)$.
   (b) rs is a regular expression denoting $L(r)L(s)$.
   (c) $r^\star$ is a regular expression denoting $(L(r))^*$.
   (d) $\overline{a}$ is a regular expression denoting $\Sigma \backslash \{a\}$.

Regular expressions (continued)

A language described by a regular expression is a **regular language**.

Rules 1 and 2 form the base of the definition. Rule 3 provides the inductive step.

Unnecessary parentheses can be avoided in regular expressions if

- the unary operator * has the highest precedence and is left associative,
- concatenation has the second highest precedence and is left associative,
- $\mid$ has the lowest precedence and is left associative.

Under those conventions, $(a) \mid ((b)^\star(c))$ is equivalent to $a \mid b^\star c$.

Both expressions denote the language containing either the string a or zero or more b's followed by one c: $\{a, c, bc, bbc, bbbc, \dots\}$.

Regular expressions/Examples

- The regular expression $a \mid b$ denotes the set $\{a, b\}$.
- The regular expression $(a \mid b)(a \mid b)$ denotes $\{aa, ab, ba, bb\}$, the set of all strings of a's and b's of length two. Another regular expression for the set is $aa \mid ab \mid ba \mid bb$.
- The regular expression $a^\star$ denotes the set of all strings of zero or more a's, i.e. $\{\varepsilon, a, aa, aaa, \dots\}$.
- The regular expression $(a \mid b)^\star$ denotes the set of all strings containing zero of more instances of an a or b, that is the language of all words made of a's and b's. Another expression is $(a^\star b^\star)^\star$.

Regular expressions/Algebraic laws

If two regular expressions r and s denote the same language, we say r and s are **equivalent** and write $r = s$.

Law	Description
$r \mid s = s \mid r$	$\mid$ is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$	concatenation distributes over $\mid$
$\epsilon r = r$ $r\epsilon = r$	ϵ is the identity element for the concatenation

Regular expressions/Algebraic laws (cont)

Law	Description
$r^{\star\star} = r^{\star}$	Kleene closure is idempotent
$r^{\star} = r^{+} \mid \epsilon$	Kleene closure and positive closure
$r^{+} = rr^{\star}$	are closely linked

Regular definitions

It is convenient to give names to regular expressions and define new regular expressions using these names as if they were symbols.

If Σ is an alphabet, then a **regular definition** is a series of definitions of the form

$$\begin{aligned} d_1 &\to r_1 \\ d_2 &\to r_2 \\ &\cdots \\ d_n &\to r_n \end{aligned}$$

where each d_i is a distinct name and each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$, i.e. the basic symbols and the previously defined names. The restriction to d_j such $j < i$ allows to construct a regular expression over Σ only by repeatedly replacing all the names in it.

Regular definitions/Examples

As we have stated, the set of Pascal identifiers can be defined by the regular definitions

$$\begin{aligned} \textbf{letter} &\to \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\ \textbf{digit} &\to \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\ \textbf{id} &\to \textbf{letter}\ (\textbf{letter} \mid \textbf{digit})^{\star} \end{aligned}$$

Unsigned numbers in Pascal are strings like `5280`, `39.37`, `6.336E4` or `1.894E-4`.

$$\begin{aligned} \textbf{digit} &\to \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\ \textbf{digits} &\to \textbf{digit}\ \textbf{digit}^{\star} \\ \textbf{optional\_fraction} &\to \texttt{.}\ \ \textbf{digits} \mid \epsilon \\ \textbf{optional\_exponent} &\to (\texttt{E}\ (\texttt{+} \mid \texttt{-} \mid \epsilon)\ \textbf{digits}) \mid \epsilon \\ \textbf{num} &\to \textbf{digits optional\_fraction optional\_exponent} \end{aligned}$$

Regular definitions/Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

Zero or one instance. The unary operator `?` means "zero or one instance of." Formally, by definition, if r is a regular expression then $r?= r \mid \epsilon$. In other words, $(r)?$ denotes the language $L(r) \cup \{\varepsilon\}$.

digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
digits → **digit**$^+$
optional_fraction → (`.` **digits**)?
optional_exponent → (`E` (`+` | `-`)? **digits**)?
num → **digits optional_fraction optional_exponent**

Regular definitions/Shorthands (cont)

It is also possible to write:

digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
digits → **digit**$^+$
fraction → `.` **digits**
exponent → `E` (`+` | `-`)? **digits**
num → **digits fraction**? **exponent**?

Regular definitions/Shorthands (cont)

If we want to specify the characters `?`, `*`, `+`, `|`, we write them with a preceding backslash, e.g. `\?`, or between double-quotes, e.g. `"?"`. Then, of course, the character double-quote must have a backslash: `\"`

It is also sometimes useful to match against end of lines and end of files: `\n` stands for the control character "end of line" and **`$`** is for "end of file".

Non-regular languages

Some languages cannot be described by any regular expression.

For example, the language of balanced parentheses cannot be recognised by any regular expression: (), (()), ()(), (((()))()) etc.

Another example is the C programming language: it is not a regular language because it contains embedded blocs between `{` and `}`. Therefore, a lexer cannot recognise valid C programs: we need a parser.

Specifying lexers

Several tools have been built for constructing lexical analysers from special-purpose notations based on regular expressions.

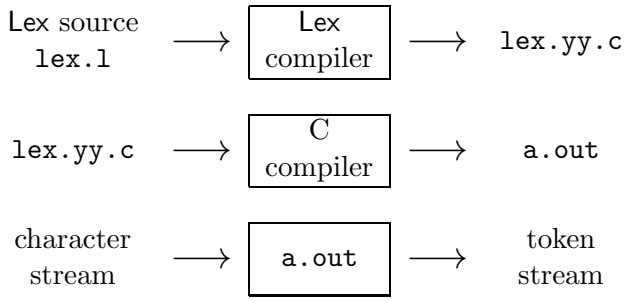
We shall now describe such a tool, named Lex, which is widely used in software projects developed in C.

Using this tool shows how the specification of patterns using regular expressions can be combined with actions, e.g., making entries into a symbol table, that a lexer may be required to perform.

We refer to the tool as the **Lex compiler** and to its input specification as the **Lex language**.

Specifying lexers (cont)

Lex is generally used in the following manner:



Specifying lexers/Lex specifications

A Lex specification (or source or program) consists of three parts:

```
declarations
%%
translation rules
%%
user code
```

The **declarations section** includes declarations of C variables, constants and regular definitions. The latter are used in the translation rules.

Specifying lexers/Lex specifications (cont)

The **translation rules** of a Lex program are statements of the form

p_1 {$action_1$}
p_2 {$action_2$}
... ...
p_n {$action_n$}

where each p_i is a regular expression and each $action_i$ is a C program fragment describing what action the lexer should take when pattern p_i matches a lexeme.

The third section holds whatever **user code** (auxiliary procedures) are needed by the actions.

Specifying lexers/Lex specifications (cont)

A lexer created by Lex interacts with a parser in the following way:

1. the parser calls the lexer;
2. the lexer starts reading its current input characters;
3. when the longest prefix of the input matches a regular expression p_i, the corresponding $action_i$ is executed;
4. finally, two cases occur whether $action_i$ returns control to the parser or not:
   (a) if so, the lexer returns the recognised token and lexeme;
   (b) if not, the lexer forgets about the recognised word and go to step 2.

Specifying lexers/Lex declarations section

```
%{ /* definitions of constants
     LT, LE, EQ, GT, GE, IF, THEN, ELSE, ID, NUM, RELOP */
%}

/* regular definitions */
ws       [ \t\n]+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter}|{digit})*
num      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%
```

Specifying lexers/Lex declarations section (cont)

First, we see a place for the declaration of the tokens. Depending on the parser, if any, used with Lex, these token may be declared by the parser. In this case, they are not declared here.

These declarations are surrounded by `%{` and `%}`. Anything between these brackets is copied verbatim in `lex.yy.c`.

Second, we see a series of regular definitions. Each definition consists of a name and a regular expression denoted by that name.

For instance, `delim` stands for the **character class** `[ \t\n]`, that is, any of the three characters: blank, tabulation (`\t`) or newline (`\n`).

Specifying lexers/Lex declarations section (cont)

Character classes. If we want to denote a set of letters or digits, it is often long to enumerate all the elements, like the **digit** regular expression.

So, instead of `4 | 1 | 2` we would shortly write `[142]`.

If the characters are consequently ordered, we can use **intervals**, called in Lex *character classes*.

For instance we write `[a-c]` instead of `a | b | c`.

Or `[0-9]` instead of `0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`.

We can now describe identifiers in a very compact way:

$$[\texttt{A-Za-z}][\texttt{A-Za-z0-9}]^{\star}$$

Specifying lexers/Lex declarations section (cont)

It is possible to have `]` and `-` in a character range: the character `]` must be first and `-` must be first or last.

The second definition is of white space, denote by the name `ws`. Note that we must write `{delim}` for **delim**, with braces inside regular expressions in order to distinguish it from the pattern made of the five letters `delim`.

The definitions of `letter` and `digit` illustrate the use of character classes (interval of (ordered) characters).

The definition of `id` shows the use of some Lex special symbols (or **meta-symbols**): parentheses and vertical bar.

Specifying lexers/Lex declarations section (cont)

The definition of `num` introduces a few more features.

There is another metasymbol "`?`" with the obvious meaning.

We notice the use of a backslash to make a character mean itself instead of being interpreted as a metasymbol: `\.` means "the dot character", while `.` (metasymbol) means "any character." This works with any metasymbol.

Note finally that we wrote `[+\-]` because, in this context, the character "`-`" has the meaning of "range", as in `[0-9]`, so we must add a backslash. This action is called **to escape** (a character).

Another way of escaping a character is to use double-quotes around it, like `"."`

Specifying lexers/Lex translation rules

```
%%
{ws}     { /* no action and no return */ }
if       { return IF; }
then     { return THEN; }
else     { return ELSE; }
{id}     { yylval = lexeme(); return ID; }
{number} { yylval = lexeme(); return NUM; }
"<"      { return LT; }
"<="     { return LE; }
"="      { return EQ; }
"<>"     { return NE; }
">"      { return GT; }
">="     { return GE; }
```

Specifying lexers/Lex translation rules (cont)

The translation rules follow the first `%%`.

The first rule says that if the regular expression denoted by the name `ws` maximally matches the input, we take no action. In particular, we do not return to the parser. Therefore, by default, this implies that the lexer will start again to recognise a token after skipping white spaces.

The second rule says that if the letters `if` are seen, return the token `IF`.

In the rule for `{id}`, we see two statements in the action. First, the Lex predefined variable `yylval` is set to the lexeme and the token `ID` is returned to the parser. The variable `yylval` is shared with the parser (it is defined in `lex.yy.c`) and is used to pass attributes about the token.

Specifying lexers/User code

Contrary to our previous presentation, the procedure `lexeme` takes here no argument. This is because the input buffer is directly and globally accessed in Lex through the pointer `yytext`, which corresponds to the first character in the buffer when the analysis started for the last time.

The length of the lexeme is given via the variable `yyleng`.

We do not show the details of the auxiliary procedures but the trailing section should look like

```
%%
char* lexeme () {
  /* returns a copy of the matched string
     between yytext[0] and yytext[yyleng-1] */
}
```

Specifying lexers/Lex longest-prefix match

If several regular expressions match the input, Lex chooses the rule which matches the most text. This is why the input `if123` is matched (recognised) as an identifier and not as the two tokens keyword (`if`) and number (`123`).

If Lex finds two or more matches of the same length, the rule listed *first* in the Lex input file is chosen.

That is why we listed the patterns `if`, `then` and `else` before `{id}`. For example, the input `if` is matched by `if` and `{id}`, so the first rule is chosen, and since we want the token keyword **if**, its regular expression is written *before* `{id}`.

Specifying lexers/Example

It is possible to use Lex alone. For instance, let `count.l` be the Lex specification

```
%{
int char_count=1, line_count=1;
%}
```

```
%%
.  {char_count++;}
\n {line_count++; char_count++;}
%%
int main () {
  yylex();  /* Calls the lexer */
  printf("There were %d characters in %d lines.\n",
          char_count,line_count);
  return 0;
}
```

Specifying lexers/Example (cont)

We have to compile the Lex specification into C code, then compile this C code and link the object code against a special library named `l`:

```
lex -t count.l > count.c
gcc -c -o count.o count.c
gcc -o counter count.o -ll
```

We can also use the C compiler `cc` with the same options instead of `gcc`. The result is a binary `counter` that we can apply on `count.l` itself:

```
cat count.l | counter
There were 210 characters in 12 lines.
```

Specifying lexers/Example (cont)

We can extend the previous specification to count words as well. For this, we need to define a regular expression for letters and bind it to a name, at the end of the declarations.

```
%{
int char_count=1, line_count=1, word_count=0;
%}
letter [A-Za-z]
%%
{letter}+ { word_count++; char_count += yyleng;
            printf ("[%s]\n",yytext); }
.         { char_count++; }
\n        { line_count++; char_count++; }
%%
...
```

Specifying lexers/Example (cont)

We can also use more regular expressions with names.

```
letter [A-Za-z]
digit  [0-9]
alpha  ({letter}|{digit})      /* No space inside! */
id     {letter}([_]*{alpha})*  /* No space inside! */
%%
{id} { word_count++; char_count += yyleng;
       printf ("word=[%s]\n",yytext); }
.    { char_count++; }
\n   { line_count++; char_count++; }
```

Specifying lexers/Example (cont)

By default, if there is no parser and no explicit `main` procedure, Lex will add one in the produced C code as if it were given in the user code section (at the end of the specification) as

```
int main ()
{
 yylex();
 return 0;
}
```

Recognition of tokens

Until now we showed how to specify tokens. Now we show how to recognise them, i.e., realise lexical analysis. Let us consider the following token definition:

$$
\begin{aligned}
\textbf{if} &\rightarrow \texttt{if} \\
\textbf{then} &\rightarrow \texttt{then} \\
\textbf{else} &\rightarrow \texttt{else} \\
\textbf{relop} &\rightarrow \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{>} \mid \texttt{>=} \\
\textbf{digit} &\rightarrow [\texttt{0-9}] \\
\textbf{letter} &\rightarrow [\texttt{A-Za-z}] \\
\textbf{id} &\rightarrow \textbf{letter}\ (\textbf{letter} \mid \textbf{digit})^{\star} \\
\textbf{num} &\rightarrow \textbf{digit}^{+}\ (\texttt{.}\ \textbf{digit}^{+})?\ (\texttt{E}\ (\texttt{+} \mid \texttt{-})?\ \textbf{digit}^{+})?
\end{aligned}
$$

Recognition of tokens/Reserved identifiers and white space

It is common to consider keywords as **reserved identifiers**, i.e.,, in this case, a valid identifier cannot be any token **if**, **then** or **else**.

This is usually not specified but instead programmed.

In addition, assume lexemes are separated by white spaces, consisting of non-null sequences of blanks, tabulations and newline characters. The lexer usually strips out those white spaces by comparing them to the regular definition **white_space**:

$$
\begin{aligned}
\textbf{delim} &\rightarrow \textbf{blank} \mid \textbf{tab} \mid \textbf{newline} \\
\textbf{white\_space} &\rightarrow \textbf{delim}^{+}
\end{aligned}
$$

If a match for **white_space** is found, the lexer does **not** return a token to the parser. Rather, it proceeds to find a token following the white space and return it to the parser.

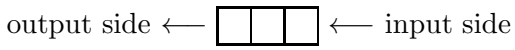
Recognition of tokens/Input buffer

The stream of characters that provides the input to the lexer comes usually from a file.

For efficiency reasons, when this file is opened, a **buffer** is associated to it, so the lexer actually reads its characters from this buffer in memory.

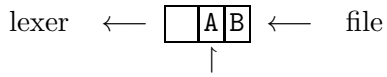
A buffer is like a **queue**, or **FIFO** (*First in, First out*), i.e., a list whose one end is used to put elements in and whose other end is used to get elements out, one at a time. The only difference is that a buffer has a **fixed size** (hence a buffer can be full).

An empty buffer of size three is depicted as



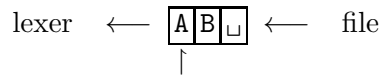
Recognition of tokens/Input buffer (cont)

If we input characters `A` then `B` in this buffer, we draw



The symbol ↑ is a pointer to the next character available for output.

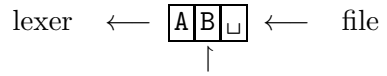
Beware! The blank character will now be noted ␣, in order to avoid confusion with an empty cell in a buffer. So, if we input now a blank in our buffer from the file, we get the full buffer



and no more inputs are possible until at least one output is done.

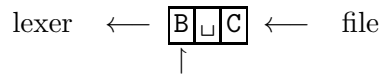
Recognition of tokens/Input buffer/Full buffer

Be careful: a buffer is full if and only if ↑ points to the leftmost character. For example,



is **not** a full buffer: there is still room for one character.

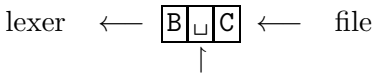
If we input `C`, it becomes:



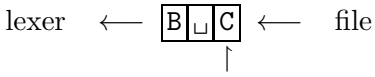
which is now a full buffer. The overflowing character `A` has been discarded.

Recognition of tokens/Input buffer (cont)

Now if we output a character (i.e.,, equivalently, the lexer inputs a character) we get



Let us output another character:

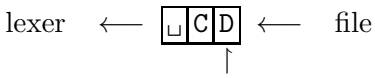


Now, if the lexer needs a character, `C` is output and some routine automatically reads some more characters from the disk and fill them in order into the buffer.

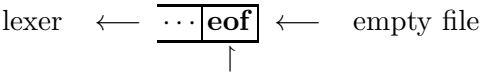
This happens when we output the rightmost character.

Recognition of tokens/Input buffer (cont)

Assuming the next character in the file is `D`, after outputting `C` we get



If the buffer only contains the **end-of-file** (noted here **eof**) character, it means that no more characters are available from the file. So if we have the situation



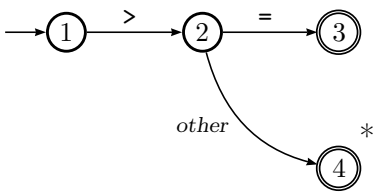
in which the lexer requests a character, it would get **eof** and subsequent requests would fail, because both the buffer and the file would be empty.

Recognition of tokens/Transition diagrams

As an intermediary step in the construction of a lexical analyser, we introduce another concept, called **transition diagram**

Transition diagrams depict the actions that take place when a lexer is called by a parser to get the next token.]

States in a transition diagram are drawn as circles. Some states have double circles, with or without a *. States are connected by arrows, called **edges**, each one carrying an input character as **label**, or the special label *other*.



Recognition of tokens/Transition diagrams (cont)

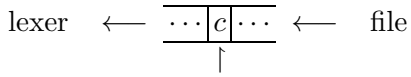
Double-circled states are called **final states**. The special arrow which do not connect two states points to the **initial state**.

A state in the transition diagram corresponds to the state of the input buffer, i.e., its contents and the output pointer at a given moment.

At the initial state, the buffer contains at least one character.

If the only one remaining character is **eof**, the lexer returns a special token **$** to the parser and stops.

Assume the character c is pointed by $\uparrow$ in the input buffer and that c is not **eof**:



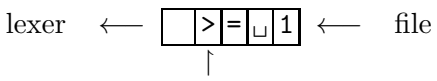
Recognition of tokens/Transition diagrams and buffering

When the parser requests a token, if an edge to state s has a label with character c, then the current state in the transition diagram becomes s and c is removed from the buffer.

This is repeated until a final state is reached or we get stuck.

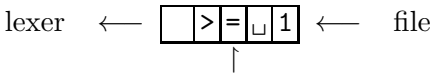
If a final state is reached, it means the lexer recognised a token — which is in turn returned to the parser. Otherwise a lexical error occurred.

Let us consider again the diagram page 35. Assume the initial input buffer is

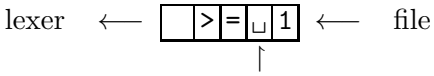


Recognition of tokens/Transition diagrams and buffering (cont)

From the initial state 1 to state number 2 there is an arrow with the label >. Because this label is present at the output position of the buffer, we can change the diagram state to 2 and remove < from the buffer, which becomes



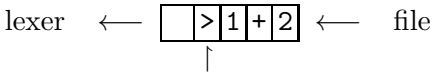
From state 2 to state 3 there is an arrow with label =, so we remove it:



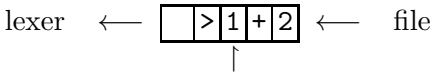
and we move to state 3. Since state 3 is a final state, we are done: we recognised the token **relop**⟨>=⟩.

Recognition of tokens/Transition diagrams and buffering (cont)

Imagine now the input buffer is



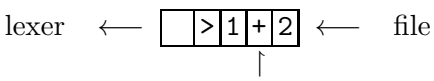
In this case, we will move from the initial state to state 2:



We cannot use the edge with label =. But we can use the one with "*other*". Indeed, *the "other" label refers to any character that is not indicated by any of the edges leaving the state.*

Recognition of tokens/Transition diagrams and buffering (cont)

So we move to state 4, the input buffer becomes

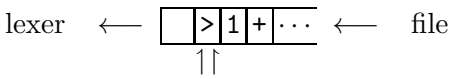


and the lexer emits the token **relop**⟨>⟩.

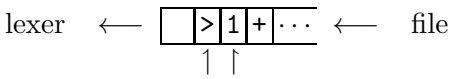
But there is a problem here: if the parser requests another token, we have to start again with this buffer but we already skipped the character 1 and we forgot where the recognised lexeme starts...

Recognition of tokens/Transition diagrams and buffering (cont)

The idea is to use another arrow to mark the starting position when we try to recognise a token. Let ↿ be this new pointer. Then the initial buffer of our previous example would be depicted as



When the lexer reads the next available character, the pointer ↾ is shifted to the right of one position.

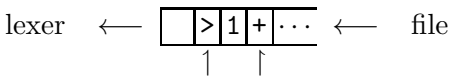


We are now at state 2 and the current character, i.e., pointed by ↾, is `1`.

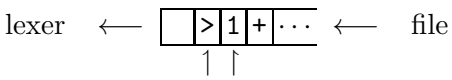
Recognition of tokens/Transition diagrams and buffering (cont)

The only way to continue is to go to state 4, using the special label *other*.

We shift the pointer of the secondary buffer to the right and, since it points to the last position, we input one character from the primary buffer:



State 4 is a final state a bit special: it is marked with *. This means that before emitting the recognised lexeme we have to shift the current pointer by one position *to the left*:



Recognition of tokens/Transition diagrams and buffering (cont)

This allows to recover the character `1` *as current character.*

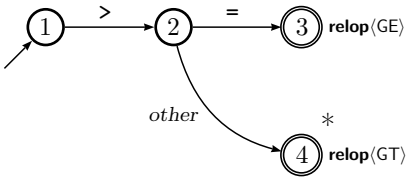
Moreover, the recognised lexeme now always starts at the ↿ pointer and ends one position before the ↾. So, here, the lexer outputs the lexeme `>`.

Recognition of tokens/Transition diagrams (resumed)

Actually, we can complete our token specification by adding some extra information that are useful for the recognition process (as we just described).

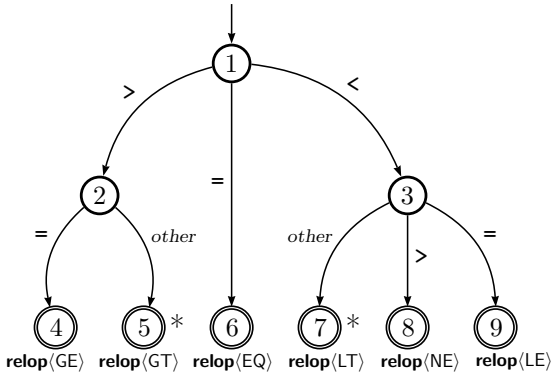
First, it is convenient for some tokens, like **relop** not to carry the lexeme verbatim, but a symbolic name instead, which is independent of the actual size of the lexeme. For instance, we shall write **relop**⟨GT⟩ instead of **relop**⟨>⟩.

Second, it is useful to write the recognised token and the lexeme close to the final state in the transition diagram itself. Consider



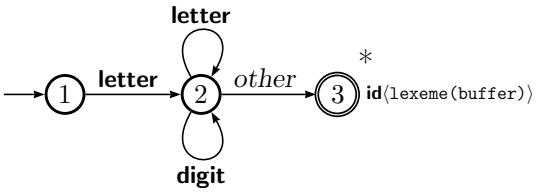
Recognition of tokens/Transition diagrams (cont)

Now let us give the transition diagram for recognising the token **relop** completely. Note that the previous diagram is a part of this one.



Recognition of tokens/Identifiers and longest prefix match

A transition diagram for specifying identifiers is



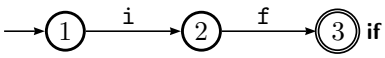
`lexeme` is a function call which returns the recognised lexeme (as found in the `buffer`)

The *other* label on the last step to final state force the identifier to be of *maximal length*. For instance, given `counter+1`, the lexer will recognise `counter` as identifier and not just `count`. This is called **the longest prefix** property.

Recognition of tokens/Keywords

Since keywords are sequences of letters, they are exceptions to the rule that a sequence of letters and digits starting with a letter is an identifier.

One solution for specifying keywords is to use dedicated transition diagrams, one for each keyword. For example, the **if** keyword is simply specified as



If one keyword diagram succeeds, i.e., the lexer reaches a final state, then the corresponding keyword is transmitted to the parser; otherwise, another keyword diagram is tried after shifting the current pointer $\uparrow$ in the input buffer back to the starting position, i.e pointed by $\restriction$.

Recognition of tokens/Keywords (cont)

There is a problem, though. Consider the Objective Caml language, where there are two keywords **fun** and **function**.

If the diagram of **fun** is tried successfully on the input `function` and then the diagram for identifiers, the lexer outputs the lexemes **fun** and **id**⟨`ction`⟩ instead of one keyword **function**...

As for identifiers, we want the longest prefix property to hold for keywords too and this is simply achieved by *ordering the transition diagrams*. For example, the diagram of **function** must be tried before the one for **fun** because **fun** is a prefix of **function**.

This strategy implies that the diagram for the identifiers (given page 39) must appear *after* the diagrams for the keywords.

Recognition of tokens/Keywords (cont)

There are still several drawbacks with this technique, though.

The first problem is that if we indeed have the longest prefix property among keywords, it does not hold with respect to the identifiers.

For instance, `iff` would lead to the keyword **if** and the identifier `f`, instead of the (longest and sole) identifier `iff`.

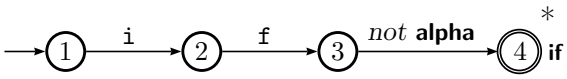
This can be remedied by forcing the keyword diagram to recognise a keyword and not an identifier. This is done by failing if the keyword is followed by a letter or a digit (remember we try the longest keywords first, otherwise we would miss some keywords — the ones which have prefix keywords).

Recognition of tokens/Keywords (cont)

The way to specify this is to use a special label *not* such as *not c* denotes the set of characters which are *not c*.

Actually, the special label *other* can always be represented using this *not* label because *other* means "not the others labels."

Therefore, the completed **if** transition diagram would be



where **alpha** (which stands for "alpha-numerical") is defined by the following regular definition:

$$\textbf{alpha} \rightarrow \textbf{letter} \mid \textbf{digit}$$

Recognition of tokens/Keywords (cont)

The second problem with this approach is that we have to create a transition diagram for each keyword and a state for each of their letters.

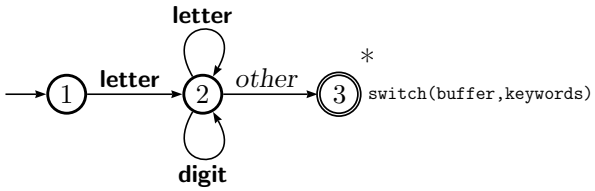
In real programming languages, this means that we get hundreds of states only for the keywords...

This problem can be avoided if we change our technique and give up the specification of keywords with transition diagrams.

Recognition of tokens/Keywords (cont)

Since keywords are a strict subset of identifiers, let us use only the identifier diagram but *we change the action at the final state*, i.e., instead of always returning a **id** token, we make some computations first to decide whether it is either a keyword or an identifier.

Let us call `switch` the function which makes this decision based on the buffer (equivalently, the current diagram state) and a **table of keywords**. We specify



Recognition of tokens/Keywords (cont)

The table of keywords is a two-column table whose first column (the entry) contains the keyword lexemes and the second column the corresponding token:

Keywords	
Lexeme	Token
`if`	**if**
`then`	**then**
`else`	**else**

Recognition of tokens/Keywords (cont)

Let us write the code for `switch` in the following pseudo-language:

SWITCH(*buffer*, *keywords*)
str ← LEXEME(*buffer*)
if *str* ∈ $\mathcal{D}$(*keywords*)
then SWITCH ← *keywords*[*str*]
else SWITCH ← **id**⟨*str*⟩

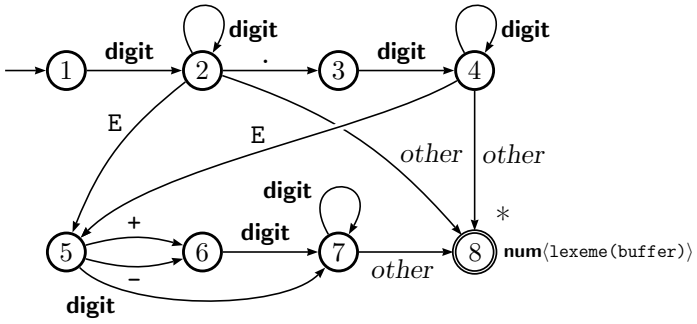
Function names are in uppercase, like LEXEME. Writing $x \leftarrow a$ means that we **assign** the value of expression a to the variable x. Then the value of x is the value of a. The value $\mathcal{D}(t)$ is the first column of table t. The value $t[e]$ is the value corresponding to e in table t. SWITCH is also used as a special variable whose value becomes the result of the function SWITCH when it finishes.

Recognition of tokens/Numbers

Let us consider now the numbers as specified by the regular definition

$$\textbf{num} \rightarrow \textbf{digit}^+ \ (\texttt{.}\ \textbf{digit}^+)?\ (\texttt{E}\ (\texttt{+} \mid \texttt{-})?\ \textbf{digit}^+)?$$

and propose a transition diagram as an intermediary step to their recognition:

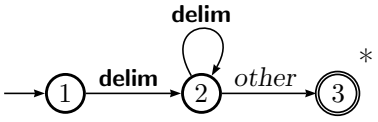


Recognition of tokens/White spaces

The only remaining issue concerns white spaces as specified by the regular definition

$$\textbf{white\_space} \rightarrow \textbf{delim}^+$$

which is equivalent to the transition diagram



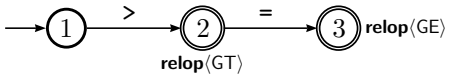
The specificity of this diagram is that there is no action associated to the final state: no token is emitted.

Recognition of tokens/Simplified

There is a simple away to reduce the size of the diagrams used to specify the tokens while retaining the longest prefix property: allow to pass through several final states.

This way, we can actually also get rid of the * marker on final states.

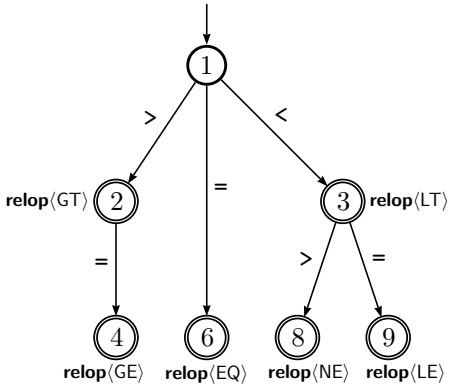
Coming back to the first example page 39, we would simply write:



But we have to change the recognition process a little bit here in order to keep the longest prefix match: we do not want to stop at state 2 if we could recognise >=.

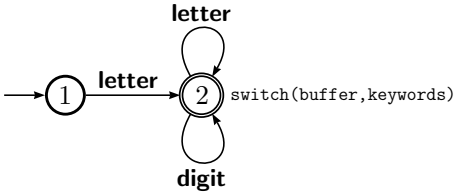
Recognition of tokens/Simplified/Comparisons

The simplified complete version with respect to the one given page 39 is



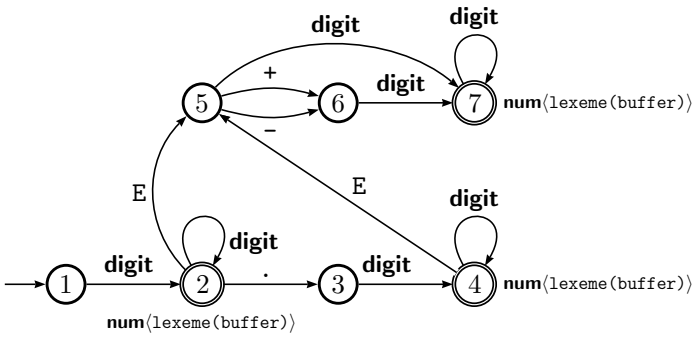
Recognition of tokens/Simplified/Identifiers

The transition diagram for specifying identifiers *and* keywords looks now like



Recognition of tokens/Simplified/Numbers

The transition diagram for specifying numbers is simpler now:



Recognition of tokens/Simplified/Interpretation

How does we interpret these new transition diagrams, where the final states may have out-going edges (and the initial state have incoming edges)?

For example, let us consider the recognition of a number:

lexer ⟵ | a | = | 1 | 5 | 3 | + | 6 | ··· | ⟵ file

↑ ↑

As usual, if there is a label of an edge going out of the current state which matches the current character in the buffer, the ↑ pointer is shifted to the right of one position.

Recognition of tokens/Simplified/Interpretation (cont)

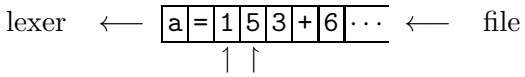
The new feature here is about final states. When the current state is final

1. the current position in the buffer is pointed to with a new pointer ⇑;
2. if there is an out-going edge which carries a matching character, we try to recognise a longer lexeme;
   (a) if we fail, i.e., if we cannot go further in the diagram and the current state is not final, then we shift back the current pointer ↑ to the position pointed by ⇑
   (b) and return the then-recognised token and lexeme.
3. if not, we return the recognised token and lexeme associated to the current final state.

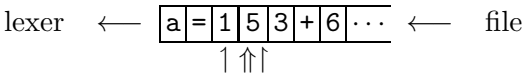
Recognition of tokens/Simplified/Example

Following our example of number recognition:

- The label **digit** matches the current character in the buffer, i.e., the one pointed by $\restriction$, so we move to state 2 and we shift right by one the pointer $\restriction$.

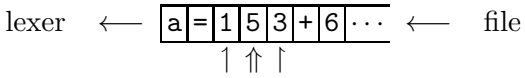


- The state 2 is final, so we set the $\Uparrow$ pointer to the current position in the buffer

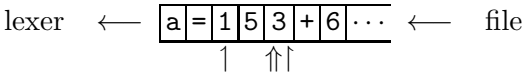


Recognition of tokens/Simplified/Example (cont)

- We shift right by one the current pointer and stay in state 2 because the matching edge is a loop (notice that we did not stop here).

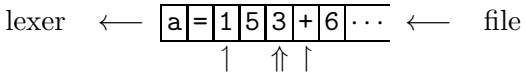


- The state 2 is final so we set the $\Uparrow$ to point to the current position:

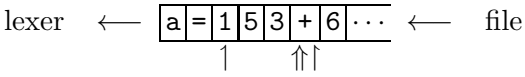


Recognition of tokens/Simplified/Example (cont)

- The **digit** label of the loop matches again the current character (here `3`), so we shift right by one the current pointer.



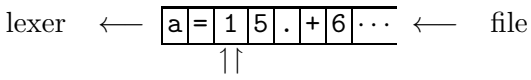
- Because state 2 is final we set the $\Uparrow$ to the current pointer $\restriction$:



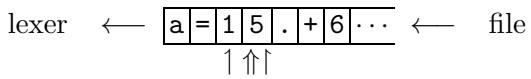
- State 2 is a final state, so it means that we succeeded in recognising the token associated with state 2: **num**$\langle$`lexeme(buffer)`$\rangle$, whose lexeme is between $\upharpoonright$ included and $\restriction$ excluded, i.e., `153`.

Recognition of tokens/Simplified/Example (cont)

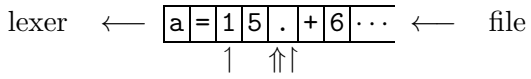
Let us consider the following initial buffer:



Character `1` is read and we arrive at state 2 with the following situation:

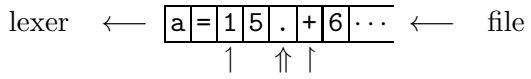


Then `5` is read and we arrive again at state 2 but with a different situation:

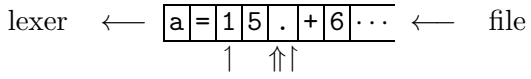


Recognition of tokens/Simplified/Example (cont)

The label on the edge from state 2 to 3 matches `.` so we move to state 3, shift by one the current pointer in the buffer:



Now we are stuck at state 3. Because this is not a final state, we should fail, i.e.,report a lexical error, but because the ⇑ has been set (i.e., we met a final state), we shift the current pointer back to the position of ⇑ and return the corresponding lexeme `15`:



Deterministic finite automata

Transition diagrams are useful *graphical* representations of instances of the mathematical concept of **deterministic finite automaton** (**DFA**).

Formally, a DFA $\mathcal{D}$ is a 5-tuple $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ where

1. a finite set of *states*, often noted Q;
2. an *initial state* $q_0 \in Q$;
3. a set of *final (*or *accepting) states* $F \subseteq Q$;
4. a finite set of *input symbols*, often noted Σ;
5. a *transition function* δ that takes a state and an input symbol and returns a state: if q is a state with an edge labeled a, the edge leads to state $\delta(q, a)$.

DFA/Recognised words

Independently of the interpretation of the states, we can define how a given word is accepted (or recognised) or rejected by a given DFA.

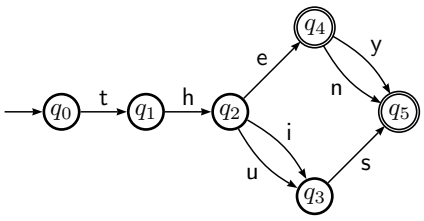
The word $a_1 a_2 \cdots a_n$, with $a_i \in \Sigma$, is recognised by the DFA $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ if

- for all $0 \leqslant i \leqslant n - 1$
- there is a sequence of states $q_i \in Q$ such as
- $\delta(q_i, a_{i+1}) = q_{i+1}$
- and $q_n \in F$.

The language recognised by $\mathcal{D}$, noted $L(\mathcal{D})$ is the set of words recognised by $\mathcal{D}$.

DFA/Recognised words/Example

For example, consider the following DFA:



The word "then" is recognised because there is a sequence of states $(q_0, q_1, q_2, q_4, q_5)$ connected by edges which satisfies

$$\begin{aligned}\delta(q_0, \mathsf{t}) &= q_1\\ \delta(q_1, \mathsf{h}) &= q_2\\ \delta(q_2, \mathsf{e}) &= q_4\\ \delta(q_4, \mathsf{n}) &= q_5\end{aligned}$$

and $q_5 \in F$, i.e. q_5 is a final state.

DFA/Recognised language

It is easy to define formally $L(\mathcal{D})$.

Let $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$.

First, let us extend δ to words and let us call this extension $\hat{\delta}$:

- for all state $q \in Q$, let $\hat{\delta}(q, \varepsilon) = q$, where ε is the empty string;
- for all state $q \in Q$, all word $w \in \Sigma^*$, all input $a \in \Sigma$, let $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$.

Then the word w is recognised by $\mathcal{D}$ if $\hat{\delta}(q_0, w) \in F$.

The language $L(\mathcal{D})$ recognised by $\mathcal{D}$ is defined as

$$L(\mathcal{D}) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

DFA/Recognised language/Example

For example, in our last example:

$$\begin{aligned}\hat{\delta}(q_0, \epsilon) &= q_0\\ \hat{\delta}(q_0, \mathsf{t}) &= \delta(\hat{\delta}(q_0, \epsilon), \mathsf{t}) = \delta(q_0, \mathsf{t}) = q_1\\ \hat{\delta}(q_0, \mathsf{th}) &= \delta(\hat{\delta}(q_0, \mathsf{t}), \mathsf{h}) = \delta(q_1, \mathsf{h}) = q_2\\ \hat{\delta}(q_0, \mathsf{the}) &= \delta(\hat{\delta}(q_0, \mathsf{th}), \mathsf{e}) = \delta(q_2, \mathsf{e}) = q_4\\ \hat{\delta}(q_0, \mathsf{then}) &= \delta(\hat{\delta}(q_0, \mathsf{the}), \mathsf{n}) = \delta(q_4, \mathsf{n}) = q_5 \in F\end{aligned}$$

DFA/Transition diagrams

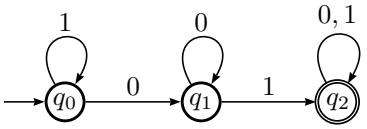
We can also redefine transition diagrams in terms of the concept of DFA.

A transition diagram for a DFA $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

1. for each state q in Q there is a **node**, i.e. a single circle with q inside;
2. for each state $q \in Q$ and each input symbol $a \in \Sigma$, if $\delta(q, a)$ exists, then there is an **edge**, i.e. an arrow, from the node denoting q to the node denoting $\delta(q, a)$ labeled by a; multiple edges can be merged into one and the labels are then separated by commas;
3. there is an edge coming to the node denoting q_0 without origin;
4. nodes corresponding to final states (i.e. in F) are double-circled.

DFA/Transition diagram/Example

Here is a transition diagram for the language over alphabet $\{0, 1\}$, called **binary alphabet**, which contains the string 01:



DFA/Transition table

There is a compact textual way to represent the transition function of a DFA: a **transition table**.

The rows of the table correspond to the states and the columns correspond to the inputs (symbols). In other words, the entry for the row corresponding to state q and the column corresponding to input a is the state $\delta(q, a)$:

δ	...	a	...
$\vdots$			
q		$\delta(q, a)$	
$\vdots$			

DFA/Transition table/Example

The transition table corresponding to the function δ of our last example is

$\mathcal{D}$	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
$\#q_2$	q_2	q_2

Actually, we added some extra information in the table: the initial state is marked with $\rightarrow$ and the final states are marked with $\#$.

Therefore, it is not only δ which is defined by means of the transition table here, but the whole DFA $\mathcal{D}$.

DFA/Example

We want to define formally a DFA which recognises the language L whose words contain an even number of 0's and an even number of 1's (the alphabet is binary).

We should understand that the role of the states here is to **not** to count the exact number of 0's and 1's that have been recognised before but this number **modulo 2**.

Therefore, there are four states because there are four cases:

1. there has been an even number of 0's and 1's (state q_0);
2. there has been an even number of 0's and an odd number of 1's (state q_1);
3. there has been an odd number of 0's and an even number of 1's (state q_2);
4. there has been an odd number of 0's and 1's (state q_3).

DFA/Example (cont)

What about the initial and final states?

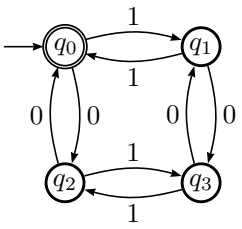
- State q_0 is the initial state because before considering any input, the number of 0's and 1's is zero and zero is even.
- State q_0 is the lone final state because its definition matches exactly the characteristic of L and no other state matches.

We know now almost how to specify the DFA for language L. It is

$$\mathcal{D} = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

where the transition function δ is described by the following transition diagram.

DFA/Example (cont)



Notice how each input 0 causes the state to cross the horizontal line.

Thus, after seeing an even number of 0's we are always above the horizontal line, in state q_0 or q_1, and after seeing an odd number of 0's we are always below this line, in state q_2 or q_3.

There is a vertically symmetric situation for transitions on 1.

DFA/Example (cont)

We can also represent this DFA by a transition table:

$\mathcal{D}$	0	1
$\# \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

We can use this table to illustrate the construction of $\hat{\delta}$ from *delta*. Suppose the input is 110101. Since this string has even numbers of 0's and 1's, it belongs to L, i.e. we expect $\hat{\delta}(q_0, 110101) = q_0$, since q_0 is the sole final state.

DFA/Example (cont)

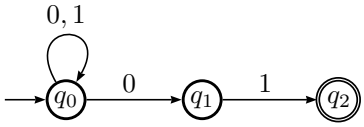
We can check this by computing step by step $\hat{\delta}(q_0, \texttt{110101})$, from the shortest prefix to the longest (which is the word `110101` itself):

$$
\begin{aligned}
\hat{\delta}(q_0, \varepsilon) &= q_0 \\
\hat{\delta}(q_0, \texttt{1}) &= \delta(\hat{\delta}(q_0, \varepsilon), \texttt{1}) && = \delta(q_0, \texttt{1}) = q_1 \\
\hat{\delta}(q_0, \texttt{11}) &= \delta(\hat{\delta}(q_0, \texttt{1}), \texttt{1}) && = \delta(q_1, \texttt{1}) = q_0 \\
\hat{\delta}(q_0, \texttt{110}) &= \delta(\hat{\delta}(q_0, \texttt{11}), \texttt{0}) && = \delta(q_0, \texttt{0}) = q_2 \\
\hat{\delta}(q_0, \texttt{1101}) &= \delta(\hat{\delta}(q_0, \texttt{110}), \texttt{1}) && = \delta(q_2, \texttt{1}) = q_3 \\
\hat{\delta}(q_0, \texttt{11010}) &= \delta(\hat{\delta}(q_0, \texttt{1101}), \texttt{0}) && = \delta(q_3, \texttt{0}) = q_1 \\
\hat{\delta}(q_0, \texttt{110101}) &= \delta(\hat{\delta}(q_0, \texttt{11010}), \texttt{1}) && = \delta(q_1, \texttt{1}) = q_0 \in F
\end{aligned}
$$

Non-deterministic finite automata

A **non-deterministic finite automaton** (**NFA**) has the same definition as a DFA except that δ returns a set of states instead of one state.

Consider



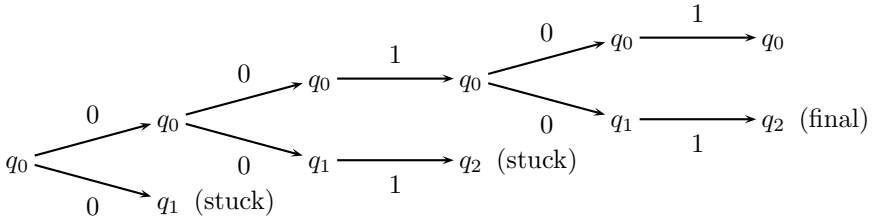
There are two out-going edges from state q_0 which are labeled 0, hence two states can be reached when 0 is input: q_0 (loop) and q_1.

This NFA recognises the language of words on the binary alphabet whose suffix is 01.

Non-deterministic finite automata (cont)

Before describing formally what is a recognisable language by a NFA, let us consider as an example the previous NFA and the input 00101.

Let us represent each transition for this input by an edge in a tree where nodes are states of the NFA.



NFA/Formal definitions

A NFA is represented essentially like a DFA: $\mathcal{N} = (Q_N, \Sigma, \delta_N, q_0, F_N)$ where the names have the same interpretation as for DFA, except δ_N which returns a subset of Q — not an element of Q.

For example, the NFA whose transition diagram is page 54 can be specified formally as

$$\mathcal{N} = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta_N, q_0, \{q_2\})$$

where the transition function δ_N is given by the transition table:

$\mathcal{N}$	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$\#q_2$	$\varnothing$	$\varnothing$

NFA/Formal definitions (cont)

Note that, in the transition table of a NFA, all the cells are filled: there is no transition between two states if and only if the corresponding cell contains $\varnothing$.

In case of a DFA, the cell would remain empty.

It is common also to set that in case of the empty word input, ε, both for the DFA and NFA, the state remains the same:

- for DFA: $\forall q \in Q.\delta_D(q, \varepsilon) = q$
- for NFA: $\forall q \in Q.\delta_N(q, \varepsilon) = \{q\}$

NFA/Formal definitions (cont)

As we did for the DFAs, we can *extend the transition function* δ_N to accept words and not just letters (labels). The extended function is noted $\hat{\delta}_N$ and defined as

- for all state $q \in Q$, let $\hat{\delta}_N(q, \varepsilon) = \{q\}$
- for all state $q \ \in Q$, all words $w \in \Sigma^*$, all input $a \in \Sigma$, let

$$\hat{\delta}_N(q, wa) = \bigcup_{q' \in \hat{\delta}_N(q,w)} \delta_N(q', a)$$

The language $L(\mathcal{N})$ recognised by a NFA $\mathcal{N}$ is defined as

$$L(\mathcal{N}) = \{w \in \Sigma^* \mid \hat{\delta}_N(q_0, w) \cap F \neq \varnothing\}$$

which means that the processing of the input stops successfully as soon as **at least one current state belongs to** F.

NFA/Example

Let us use $\hat{\delta}_N$ to describe the processing of the input `00101` by the NFA page 54:

1. $\hat{\delta}_N(q_0, \varepsilon) = q_0$
2. $\hat{\delta}_N(q_0, \mathtt{0}) = \delta_N(q_0, \mathtt{0}) = \{q_0, q_1\}$
3. $\hat{\delta}_N(q_0, \mathtt{00}) = \delta_N(q_0, \mathtt{0}) \cup \delta_N(q_1, \mathtt{0}) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$

4. $\hat{\delta}_N(q_0, \texttt{001}) = \delta_N(q_0, \texttt{1}) \cup \delta_N(q_1, \texttt{1}) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
5. $\hat{\delta}_N(q_0, \texttt{0010}) = \delta_N(q_0, \texttt{0}) \cup \delta_N(q_2, \texttt{0}) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$
6. $\hat{\delta}_N(q_0, \texttt{00101}) = \delta_N(q_0, \texttt{1}) \cup \delta_N(q_1, \texttt{1}) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\} \ni q_2$

Because q_2 is a final state, actually $F = \{q_2\}$, we get $\hat{\delta}_N(q_0, \texttt{00101}) \cap F \neq \varnothing$ thus the string `00101` is recognised by the NFA.

Equivalence of DFAs and NFAs

NFA are easier to build than DFA because one does not have to worry, for any state, of having out-going edges carrying a unique label.

The surprising thing is that NFA and DFA actually have the same expressiveness, i.e. all that can be defined by means of a NFA can also be defined with a DFA (the converse is trivial since a DFA is already a NFA).

More precisely, there is a procedure, called **the subset construction**, which converts any NFA to a DFA.

Subset construction

Consider that, in a NFA, from a state q with several out-going edges with the same label a, the transition function δ_N leads, in general, to *several* states.

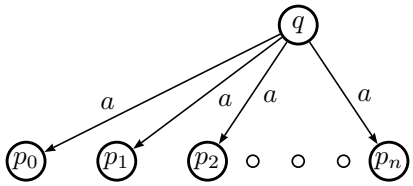
The idea of the subset construction is to create a new automaton where these edges are merged.

So we create a state p which corresponds to the set of states $\delta_N(q, a)$ in the NFA. Accordingly, we create a state r which corresponds to the set $\{q\}$ in the NFA. We create an edge labeled a between r and p. The important point is that *this edge is unique*.

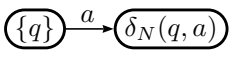
This is the first step to create a DFA from a NFA.

Subset construction (cont)

Graphically, instead of the non-determinism



where $\delta_N(q, a) = \{p_0, p_1, \ldots, p_n\}$, we get the determinism



Subset construction (cont)

Now, let us present the complete algorithm for the subset construction. Let us start from a NFA $\mathcal{N} = (Q_N, \Sigma, \delta_N, q_0, F_N)$. The goal is to construct a DFA $\mathcal{D} = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(\mathcal{D}) = L(\mathcal{N})$.

Notice that the input alphabet of the two automata are the same and the initial state of $\mathcal{D}$ if the set containing only the initial state of $\mathcal{N}$.

The other components of $\mathcal{D}$ are constructed as follows.

- Q_D is the set of subsets of Q_N; i.e. Q_D is the **power set** of Q_N. Thus, if Q_D has n states, Q_D has 2^n states. Fortunately, often not all these states are **accessible** from the initial state of Q_D, so these inaccessible states can be discarded.

Subset construction (cont)

Why is 2^n the number of subsets of a finite set of cardinal n?

Let us order the n elements and represent each subset by an n-bit string where bit i corresponds to the i-th element: it is 1 if the i-th element is present in the subset and 0 if not.

This way, we counted all the subsets and not more (a bit cannot always be 0 since all elements are used to form subsets and cannot always be 1 if there is more than one element).

There are 2 possibilities, 0 or 1, for the first bit; 2 possibilities for the second bit etc. Since the choices are independent, we multiply all: $\underbrace{2 \times 2 \times \cdots \times 2}_{n \text{ times}} = 2^n$.

Hence the number of subsets of an n-element set is also 2^n.

Subset construction (cont)

Resuming the definition of DFA $\mathcal{D}$, the other components are defined as follows.

- F_D is the set of subsets S of Q_N such as $S \cap F_N \neq \varnothing$. That is, F_D is all sets of N's states that include at least one final state of $\mathcal{N}$.
- for each set $S \subseteq Q_N$ and for each input $a \in \Sigma$,

$$\delta_D(S, a) = \bigcup_{q \in S} \delta_N(q, a)$$

In other words, to compute $\delta_D(S, a)$ we look at all the states q in S, see what states of $\mathcal{N}$ are reached from q on input a and take the union of all those states to make the next state of $\mathcal{D}$.

Subset construction/Example/Transition table

Let us consider the NFA given by its transition table page 54:

NFA $\mathcal{N}$	0	1
$\to q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$\#q_2$	$\varnothing$	$\varnothing$

and let us create an equivalent DFA.

First, we form all the subsets of the sets of the NFA and put them in the first column:

DFA $\mathcal{D}$	0	1
$\varnothing$		
$\{q_0\}$		
$\{q_1\}$		
$\{q_2\}$		
$\{q_0, q_1\}$		
$\{q_0, q_2\}$		
$\{q_1, q_2\}$		
$\{q_0, q_1, q_2\}$		

Subset construction/Example/Transition table (cont)

Then we annotate in this first column the states with $\to$ if and only if they contain the initial state of the NFA, here q_0, and we add a # if and only if the states contain at least a final state of the NFA, here q_2.

DFA $\mathcal{D}$	0	1
$\varnothing$		
$\to\{q_0\}$		
$\{q_1\}$		
$\#\{q_2\}$		
$\{q_0, q_1\}$		
$\#\{q_0, q_2\}$		
$\#\{q_1, q_2\}$		
$\#\{q_0, q_1, q_2\}$		

Subset construction/Example/Transition table (cont)

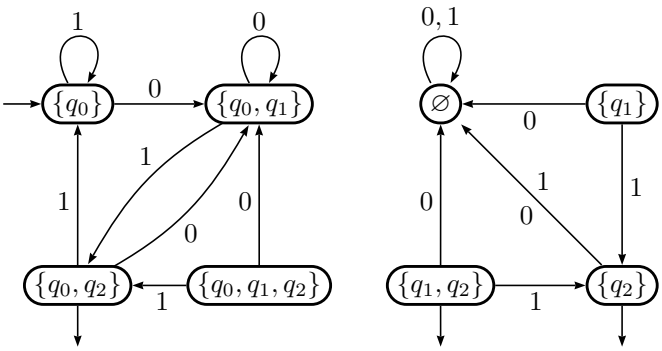
DFA $\mathcal{D}$	0	1
$\varnothing$	$\varnothing$	$\varnothing$
$\to\{q_0\}$	$\delta_N(q_0,0)$	$\delta_N(q_0,1)$
$\{q_1\}$	$\delta_N(q_1,0)$	$\delta_N(q_1,1)$
$\#\{q_2\}$	$\delta_N(q_2,0)$	$\delta_N(q_2,1)$
$\{q_0,q_1\}$	$\delta_N(q_0,0)\cup\delta_N(q_1,0)$	$\delta_N(q_0,1)\cup\delta_N(q_1,1)$
$\#\{q_0,q_2\}$	$\delta_N(q_0,0)\cup\delta_N(q_2,0)$	$\delta_N(q_0,1)\cup\delta_N(q_2,1)$
$\#\{q_1,q_2\}$	$\delta_N(q_1,0)\cup\delta_N(q_2,0)$	$\delta_N(q_1,1)\cup\delta_N(q_2,1)$
$\#\{q_0,q_1,q_2\}$	$\delta_N(q_0,0)\cup\delta_N(q_1,0)\cup\delta_N(q_2,0)$	$\delta_N(q_0,1)\cup\delta_N(q_1,1)\cup\delta_N(q_2,1)$

Subset construction/Example/Transition table (cont)

DFA $\mathcal{D}$	0	1
$\varnothing$	$\varnothing$	$\varnothing$
$\to\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_1\}$	$\varnothing$	$\{q_2\}$
$\#\{q_2\}$	$\varnothing$	$\varnothing$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$\#\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\#\{q_1,q_2\}$	$\varnothing$	$\{q_2\}$
$\#\{q_0,q_1,q_2\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$

Subset construction/Example/Transition diagram

The transition diagram of the DFA $\mathcal{D}$ is then

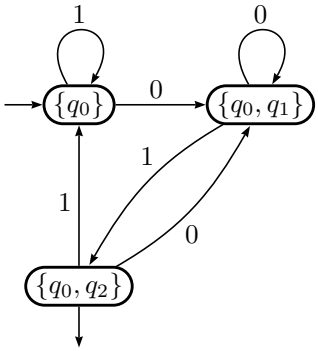


where states with out-going edges which have no end are final states.

Subset construction/Example/Transition diagram (cont)

If we look carefully at the transition diagram, we see that the DFA is actually made of two parts which are disconnected. i.e. not joined by and edge.

In particular, since we have only one initial state, this means that one part is not accessible, i.e. some states are never used to recognise or reject an input word, and we can remove this part.



Subset construction/Example/Transition diagram (cont)

It is important to understand that the states of the DFA are subsets of the NFA states.

This is due to the construction and, when finished, it is possible to hide this by **renaming the states**. For example, we can rename the states of the previous DFA in the following manner: $\{q_0\}$ into A, $\{q_0, q_1\}$ in B and $\{q_0, q_2\}$ in C.

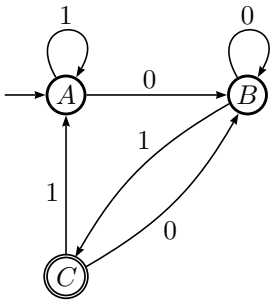
So the transition table changes:

DFA $\mathcal{D}$	0	1
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\#\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

DFA $\mathcal{D}$	0	1
$\rightarrow A$	B	A
B	B	C
$\#C$	B	A

Subset construction/Example/Transition diagram (cont)

So, finally, the DFA is simply



Subset construction/Optimisation

Even if in the worst case the resulting DFA has an exponential number of states of the corresponding NFA, it is in practice often possible to avoid the construction of inaccessible states.

- The singleton containing the initial state (in our example, $\{q_0\}$) is accessible.
- Assume we have a set S of accessible states; then for each input symbol a, we compute $\delta_D(S, a)$: this new set is also accessible.
- Repeat the last step, starting with $\{q_0\}$, until no new (accessible) sets are found.

Subset construction/Optimisation/Example

Let us consider the NFA given by its transition table page 54:

NFA $\mathcal{N}$	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$\#q_2$	$\varnothing$	$\varnothing$

Initially, the sole subset of accessible states is $\{q_0\}$:

DFA $\mathcal{D}$	0	1
$\rightarrow\{q_0\}$	$\delta_N(q_0, 0)$	$\delta_N(q_0, 1)$

that is

DFA $\mathcal{D}$	0	1
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$

Subset construction/Optimisation/Example (cont)

Therefore $\{q_0, q_1\}$ and $\{q_0\}$ are accessible sets. But $\{q_0\}$ is not a new set, so we only add to the table entries $\{q_0, q_1\}$ and compute the transitions from it:

DFA $\mathcal{D}$	0	1
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

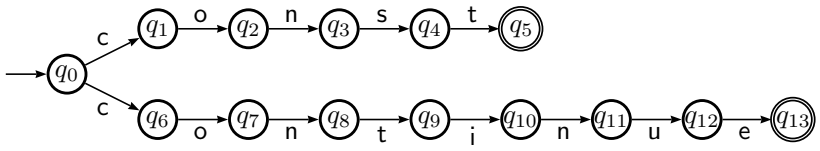
This step uncovered a new set of accessible states, $\{q_0, q_2\}$, which we add to the table and repeat the procedure, and mark it as final state since $q_2 \in \{q_0, q_2\}$:

DFA $\mathcal{D}$	0	1
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\#\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

We are done since there is no more new accessible sets.

Subset construction/Tries

Lexical analysis tries to recognise a prefix of the input character stream (in other words, the first lexeme of the given program). Consider the C keywords **const** and **continue**:

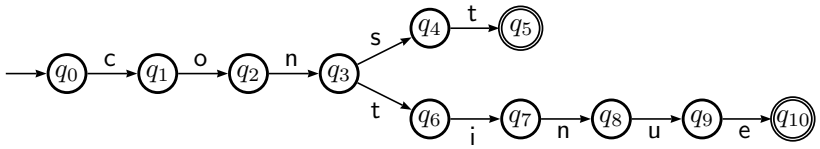


This example shows that a NFA is much more comfortable than a DFA for specifying tokens for lexical analysis: we design *separately* the automata for each token and then merge their initial states into one, leading to one (possibly big) NFA.

It is possible to apply the subset construction to this NFA.

Subset construction/Tries (cont)

After forming the corresponding NFA as in the previous example, it is actually easy to construct an equivalent DFA by **sharing their prefixes**, hence obtaining a tree-like automaton called **trie** (pronounced as the word 'try'):

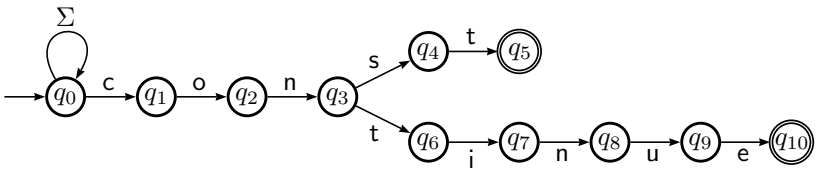


Note that this construction only works for a list of constant words, like keywords.

Subset construction/Text searching

This technique can easily be generalized for searching constant strings (like keywords) in a text, i.e. not only as a prefix of a text, but *at any position*.

It suffices to add a loop on the initial state for each possible input symbol. If we note Σ the language alphabet, we get



Subset construction/Text searching (cont)

It is possible to apply the subset construction to this NFA or to use it directly for searching the two keywords at any position in a text.

In case of direct use, the difference between this NFA and the trie page 63 is that there is no need here to "restart" by hand the recognition process once a keyword has been recognised: we just continue.

This works because of the loop on the initial state, which always allows a new start.

Try for instance the input `constantcontinue`.

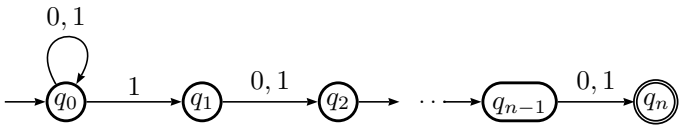
Subset construction/Bad case

The subset construction can lead, in the worst case, to a number of states which is the total number of state subsets of the NFA.

In other words, if the NFA has n states, the equivalent DFA by subset construction can have 2^n states (see page 58 for the count of all the subsets of a finite set).

Subset construction/Bad case (cont)

Consider the following NFA, which recognises all binary strings which have 1 at the n-th position from the end:



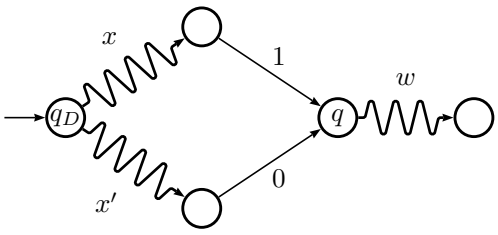
The language recognised by this NFA is $\Sigma^* 1 \Sigma^{n-1}$, where $\Sigma = \{0, 1\}$, that is: all words of length greater or equal to n are accepted as long as the n-th bit from the **end** is 1.

Therefore, in any equivalent DFA, all the prefixes of length n should not lead to a stuck state, because the automaton must wait until the **end** of the word to accept or reject it.

Subset construction/Bad case (cont)

If the states reached by these prefixes are all different, then there are at least 2^n states in the DFA.

Equivalently (by contraposition), if there are less than 2^n states, then some states can be reached by several strings of length n:



where words $x1w$ and $x'0w$ have length n.

Subset construction/Bad case (cont)

Let us call the DFA $\mathcal{D} = (Q_D, \Sigma, \delta_D, q_D, F_D)$, where $q_D = \{q_0\}$.

The extended transition function is noted $\hat{\delta}_D$ as usual. The situation of the previous picture can be formally expressed as

$$\hat{\delta}_D(q_D, x1) = \hat{\delta}_D(q_D, x'0) = q \tag{1}$$

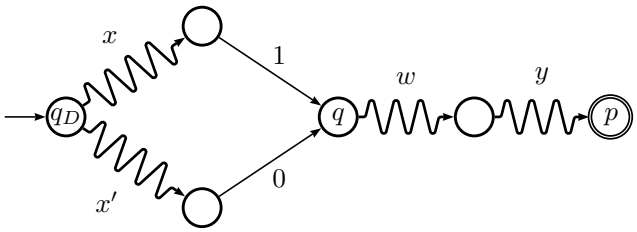
$$|x1w| = |x'0w| = n \tag{2}$$

where $|u|$ is the length of u.

Subset construction/Bad case (cont)

Let y be a any string of 0 and 1 such as $|wy| = n - 1$.

Then $\hat{\delta}_D(q_D, x1wy) \in F_D$ since there is a 1 at the n-th position from the end:



Also, $\hat{\delta}_D(q_D, x'0wy) \notin F_D$ because there is a 0 at the n-th position from the end.

Subset construction/Bad case (cont)

On the other hand, equation (1) implies

$$\hat{\delta}_D(q_D, x1wy) = \hat{\delta}_D(q_D, x'0wy) = p$$

So there is contradiction because a state (here, p) must be either final or not final, it cannot be both...

As a consequence, we must reject our initial assumption: there are at least 2^n states in the equivalent DFA.

This is a very bad case, even if it is not the worst case (2^{n+1} states).

NFA with ϵ-transitions (ϵ-NFA)

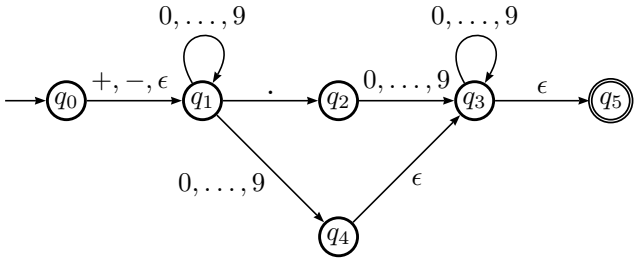
We shall now introduce another extension to NFA, called ϵ-NFA, which is a NFA whose labels can be the empty string, noted ϵ.

The interpretation of this new kind of transition, called ϵ-transition, is that the current state changes by following this transition *without reading any input*. This is sometimes referred as a **spontaneous transition**.

The rationale, i.e., the intuition behind that, is that $\epsilon a = a\epsilon = a$, so recognising ϵa or $a\epsilon$ is the same as recognising a. In other words, we do not need to read something more than a as input.

ϵ-NFA/Example

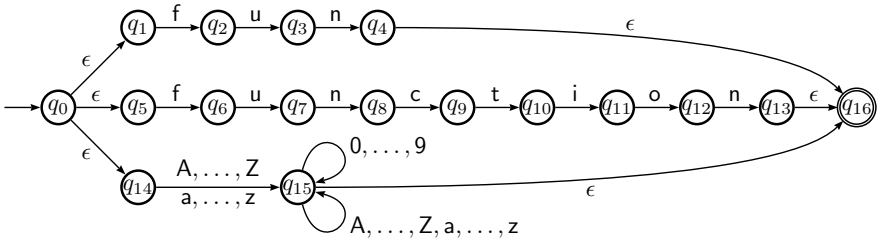
For example, we can specify signed natural and decimal numbers by means of the ϵ-NFA



This is not the simplest ϵ-NFA we can imagine for these numbers, but note the utility of the ϵ-transition between q_0 and q_1.

ϵ-NFA (cont)

In case of lexical analysers, ϵ-NFA allow to design separately a NFA for each token, then create an initial (respectively, final) state connected to all their initial (respectively, final) states with an ϵ-transition. For instance, for keywords **fun** and **function** and identifiers:



ϵ-NFA (cont)

In lexical analysis, once we have a single ϵ-NFA, we can

1. either remove all the ϵ-transitions and
   (a) either create a NFA and then maybe a DFA;
   (b) or create directly a DFA,
2. or use a formal definition of ϵ-NFA that directly leads to a recognition algorithm, just as we did for DFA and NFA.

Both methods assume that it is always possible to create an equivalent NFA, hence a DFA, from a given ϵ-NFA.

In other words, **DFA, NFA and ϵ-NFA have the same expressive power.**

ϵ-NFA (cont)

The first method constructs explicitly the NFA and maybe the DFA, while the second does not, at the possible cost of more computations at run-time.

Before entering into the details, we need to define formally an ϵ-NFA, as suggested by the second method.

The only difference between an NFA and an ϵ-NFA is that the transition function δ_E takes as second argument an element in $\Sigma \cup \{\epsilon\}$, with $\epsilon \notin \Sigma$, instead of Σ — but the alphabet still remains Σ.

ϵ-NFA/ϵ-closure

We need now a function called **ϵ-close**, which takes an ϵ-NFA $\mathcal{E}$, a state q of $\mathcal{E}$ and returns all the states which are accessible in $\mathcal{E}$ from q with label ϵ. The idea is to achieve a **depth-first traversal** of $\mathcal{E}$, starting from q and following only ϵ-transitions.

Let us call ϵ-DFS ("ϵ-Depth-First-Search") the function such as $\epsilon\text{-DFS}(q, Q)$ is the set of states reachable from q following ϵ-transitions and which is not included in Q, Q *being interpreted as the set of states already visited in the traversal*. The set Q ensures the termination of the algorithm even in presence of cycles in the automaton. Therefore, let

$$\epsilon\text{-close}(q) = \epsilon\text{-DFS}(q, \varnothing) \quad \text{if } q \in Q_E$$

where the ϵ-NFA is $\mathcal{E} = (Q_E, \Sigma, \delta_E, q_0, F_E)$.

ϵ-NFA/ϵ-closure (cont)

Now we define ϵ-DFS as follows:

$$\epsilon\text{-DFS}(q, Q) = \varnothing \qquad \text{if } q \in Q \qquad (3)$$

$$\epsilon\text{-DFS}(q, Q) = \{q\} \quad \cup \bigcup_{p \in \delta_E(q,\epsilon)} \epsilon\text{-DFS}(p, Q \cup \{q\}) \qquad \text{if } q \notin Q \qquad (4)$$

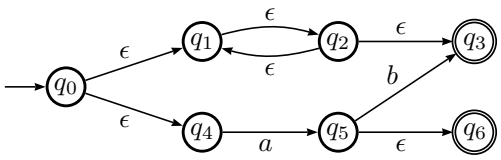
The ϵ-NFA page 67 leads to the following ϵ-closures:

$$\epsilon\text{-close}(q_1) = \{q_1\}$$
$$\epsilon\text{-close}(q_0) = \{q_0, q_1\}$$
$$\epsilon\text{-close}(q_5) = \{q_5\}$$

$$\epsilon\text{-close}(q_2) = \{q_2\}$$
$$\epsilon\text{-close}(q_3) = \{q_3, q_5\}$$
$$\epsilon\text{-close}(q_4) = \{q_4, q_3, q_5\}$$

ϵ-NFA/ϵ-closure (cont)

Consider, as a more difficult example, the following ϵ-NFA $\mathcal{E}$:



ϵ-close(q_0)

$$= \epsilon\text{-DFS}(q_0, \varnothing) \qquad \text{since } q_0 \in Q_E$$
$$= \{q_0\} \cup \epsilon\text{-DFS}(q_1, \{q_0\}) \cup \epsilon\text{-DFS}(q_4, \{q_0\}) \qquad \text{by eq. 4}$$
$$= \{q_0\} \cup \left(\{q_1\} \cup \bigcup_{p \in \delta_E(q_1,\epsilon)} \epsilon\text{-DFS}(p, \{q_0, q_1\}) \right) \qquad \text{by eq. 4}$$
$$\cup \left(\{q_4\} \cup \bigcup_{p \in \delta_E(q_4,\epsilon)} \epsilon\text{-DFS}(p, \{q_0, q_4\}) \right) \qquad \text{by eq. 4}$$

ϵ-NFA/ϵ-closure (cont)

$$\begin{aligned}
\epsilon\text{-close}(q_0) &= \{q_0\} \cup \left(\{q_1\} \cup \bigcup_{p \in \{q_2\}} \epsilon\text{-DFS}(p, \{q_0, q_1\}) \right) \\
&\qquad \cup \left(\{q_4\} \cup \bigcup_{p \in \varnothing} \epsilon\text{-DFS}(p, \{q_0, q_4\}) \right) \\
&= \{q_0\} \cup (\{q_1\} \cup \epsilon\text{-DFS}(q_2, \{q_0, q_1\})) \cup (\{q_4\} \cup \varnothing) \\
&= \{q_0, q_1, q_4\} \cup \epsilon\text{-DFS}(q_2, \{q_0, q_1\}) \\
&= \{q_0, q_1, q_4\} \cup \left(\{q_2\} \cup \bigcup_{p \in \delta_E(q_2, \epsilon)} \epsilon\text{-DFS}(p, \{q_0, q_1, q_2\}) \right)
\end{aligned}$$

ϵ-NFA/ϵ-closure (cont)

$$\begin{aligned}
\epsilon\text{-close}(q_0) &= \{q_0, q_1, q_4\} \cup \left(\{q_2\} \cup \bigcup_{p \in \{q_1, q_3\}} \epsilon\text{-DFS}(p, \{q_0, q_1, q_2\}) \right) \\
&= \{q_0, q_1, q_2, q_4\} \cup \epsilon\text{-DFS}(q_1, \{q_0, q_1, q_2\}) \\
&\qquad \cup \epsilon\text{-DFS}(q_3, \{q_0, q_1, q_2\}) \\
&= \{q_0, q_1, q_2, q_4\} \cup \varnothing \qquad \text{by eq. 3, since } q_1 \in \{q_0, q_1, q_2\} \\
&\quad \cup \left(\{q_3\} \cup \bigcup_{p \in \delta_E(q_3, \epsilon)} \epsilon\text{-DFS}(p, \{q_0, q_1, q_2, q_3\}) \right) \quad \text{by eq. 4} \\
&= \{q_0, q_1, q_2, q_3, q_4\} \cup \bigcup_{p \in \varnothing} \epsilon\text{-DFS}(p, \{q_0, q_1, q_2, q_3\}) \\
&= \{q_0, q_1, q_2, q_3, q_4\}
\end{aligned}$$

ϵ-NFA/ϵ-closure (cont)

It is useful to extend ϵ-close to sets of states, not just states.

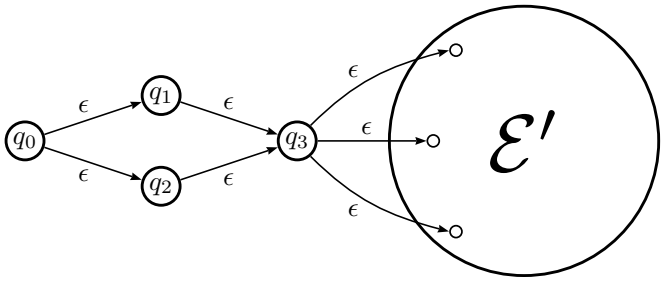
Let us note $\overline{\epsilon\text{-close}}$ this extension, which we can easily define as

$$\overline{\epsilon\text{-close}}(Q) = \bigcup_{q \in Q} \epsilon\text{-close}(q)$$

for any subset $Q \subseteq Q_E$ where the ϵ-NFA is $\mathcal{E} = (Q_E, \Sigma, \delta_E, q_E, F_E)$.

ϵ-NFA/ϵ-closure / Optimisation

Compute the ϵ-closure of q_0 in the following ϵ-NFA $\mathcal{E}$:



where the sub-ϵ-NFA $\mathcal{E}'$ contains only ϵ-transitions and all its Q' states are accessible from q_3.

ϵ-NFA/ϵ-closure/Optimisation (cont)

$$
\begin{aligned}
\epsilon\text{-close}(q_0) &= \epsilon\text{-DFS}(q_0, \varnothing) \\
&= \{q_0\} \cup \epsilon\text{-DFS}(q_1, \{q_0\}) \cup \epsilon\text{-DFS}(q_2, \{q_0\}) \\
&= \{q_0\} \cup (\{q_1\} \cup \epsilon\text{-DFS}(q3, \{q_0, q_1\})) \\
&\qquad \cup (\{q_2\} \cup \epsilon\text{-DFS}(q_3, \{q_0, q_2\})) \\
&= \{q_0, q_1, q_2\} \cup \epsilon\text{-DFS}(q3, \{q_0, q_1\}) \cup \epsilon\text{-DFS}(q_3, \{q_0, q_2\}) \\
&= \{q_0, q_1, q_2, q_3, \} \cup (\{q_3\} \cup Q') \cup (\{q_3\} \cup Q') \\
&= \{q_0, q_1, q_2, q_3, \} \cup Q'
\end{aligned}
$$

We compute $\{q_3\} \cup Q'$ two times, that is, we traverse two times q_3 and all the states of $\mathcal{E}'$, which can be inefficient if Q' is big.

ϵ-NFA/ϵ-closure/Optimisation (cont)

The way to avoid duplicating traversals is to change the definitions of ϵ-close and $\overline{\epsilon\text{-close}}$.

Dually, we need a new definition of ϵ-DFS and create function $\overline{\epsilon\text{-DFS}}$ which is similar to ϵ-DFS except that it applies to set of states instead of one state:

$$
\begin{aligned}
\epsilon\text{-close}(q) &= \epsilon\text{-DFS}(q, \varnothing) && \text{if } q \in Q_E \\
\overline{\epsilon\text{-close}}(Q) &= \overline{\epsilon\text{-DFS}}(Q, \varnothing) && \text{if } Q \subseteq Q_E
\end{aligned}
$$

We interpret Q' in ϵ-DFS(q, Q') and $\overline{\epsilon\text{-DFS}}(Q, Q')$ as the set of states that have already been visited in the depth-first search.

Variables q and Q denote, respectively, a state and a set of states that have to be explored.

ϵ-NFA/ϵ-closure/Optimisation (cont)

In the first definition we computed the *new reachable states*, but in the new one we compute the *currently reached states*. Then let us redefine ϵ-DFS this way:

$$\epsilon\text{-DFS}(q, Q') = Q' \qquad \text{if } q \in Q' \qquad (1')$$
$$\epsilon\text{-DFS}(q, Q') = \overline{\epsilon\text{-DFS}}(\delta_E(q, \epsilon), Q' \cup \{q\}) \qquad \text{if } q \notin Q' \qquad (2')$$

Contrast with the first definition

$$\epsilon\text{-DFS}(q, Q') = \varnothing \qquad \text{if } q \in Q' \qquad (1)$$
$$\epsilon\text{-DFS}(q, Q') = \{q\} \quad \cup \bigcup_{p \in \delta_E(q,\epsilon)} \epsilon\text{-DFS}(p, Q' \cup \{q\}) \qquad \text{if } q \notin Q' \qquad (2)$$

Hence, in (1) we return $\varnothing$ because there is no new state, i.e., none not already in Q', whereas in (1') we return Q' itself.

ϵ-NFA/ϵ-closure/Optimisation (cont)

The new definition of $\overline{\epsilon\text{-DFS}}$ is not more difficult than the first one:

$$\overline{\epsilon\text{-DFS}}(\varnothing, Q') = Q' \qquad (5)$$
$$\overline{\epsilon\text{-DFS}}(\{q\} \cup Q, Q') = \overline{\epsilon\text{-DFS}}(Q, \epsilon\text{-DFS}(q, Q')) \qquad \text{if } q \notin Q \qquad (6)$$

Notice that the definitions of ϵ-DFS and $\overline{\epsilon\text{-DFS}}$ are **mutually recursive**, i.e., they depend on each other.

In (2) we traverse states in *parallel* (consider the union operator), starting from each element in $\delta_E(q, \epsilon)$, whereas in (2') and (6), we traverse them *sequentially* so we can use the information collected (currently reached states) in the previous searches.

ϵ-NFA/ϵ-closure/Optimisation (cont)

Coming back to our example page 185, we find

$$\begin{aligned}
\epsilon\text{-close}(q_0) &= \epsilon\text{-DFS}(q_0, \varnothing) && q_0 \in Q_E \\
&= \overline{\epsilon\text{-DFS}}(\{q_1, q_2\}, \{q_0\}) && \text{by eq. (2')} \\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \epsilon\text{-DFS}(q_1, \{q_0\})) && \text{by eq. (4)} \\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \overline{\epsilon\text{-DFS}}(\{q_3\}, \{q_0, q_1\})) && \text{by eq. (2')} \\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_3, \{q_0, q_1\}))) && \text{by eq. (4)} \\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \epsilon\text{-DFS}(q_3, \{q_0, q_1\})) && \text{by eq. (3)} \\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \{q_0, q_1, q_3\} \cup Q') \\
&= \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_2, \{q_0, q_1, q_3\} \cup Q')) && \text{by eq. (4)}
\end{aligned}$$

ϵ-NFA/ϵ-closure/Optimisation (cont)

$$\begin{aligned}
\epsilon\text{-close}(q_0) &= \epsilon\text{-DFS}(q_2, \{q_0, q_1, q_3\} \cup Q') && \text{by eq. (3)} \\
&= \overline{\epsilon\text{-DFS}}(\{q_3\}, \{q_0, q_1, q_2, q_3\} \cup Q') && \text{by eq. (2')} \\
&= \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_3, \{q_0, q_1, q_2, q_3\} \cup Q')) && \text{by eq. (4)} \\
&= \epsilon\text{-DFS}(q_3, \{q_0, q_1, q_2, q_3\} \cup Q') && \text{by eq. (3)} \\
&= \{q_0, q_1, q_2, q_3\} \cup Q' && \text{by eq. (1')}
\end{aligned}$$

The important thing here is that we did not compute (traverse) several times Q'. Note that some equations can be used in a different order and q can be chosen arbitrarily in equation (4), but the result is always the same.

Extended transition functions for ϵ-NFAs

The ϵ-closure allows to explain how a ϵ-NFA recognises or rejects a given input. Let $\mathcal{E} = (Q_E, \Sigma, \delta_E, q_0, F_E)$.

We want $\hat{\delta}_E(q, w)$ be the set of states reachable from q along a path whose labels, when concatenated, for the string w. The difference here with NFA's is that several ϵ can be present along this path, despite not contributing to w. For all state $q \in Q_E$, let

$$\begin{aligned}
\hat{\delta}_E(q, \epsilon) &= \epsilon\text{-close}(q) \\
\hat{\delta}_E(q, wa) &= \overline{\epsilon\text{-close}}\left(\bigcup_{p \in \hat{\delta}_E(q,w)} \delta_N(p, a) \right) \qquad \text{for all } a \in \Sigma, w \in \Sigma^*
\end{aligned}$$

This definition is based on the regular identity $wa = ((w\epsilon^*)a)\epsilon^*$.

Extended transition functions for ϵ-NFAs/Example

Let us consider again the ϵ-NFA recognising natural and decimal numbers, at page 67, and compute the states reached on the input `5.6`:

$$\begin{aligned}
\hat{\delta}_E(q_0, \epsilon) &= \epsilon\text{-close}(q_0) = \{q_0, q_1\} \\
\hat{\delta}_E(q_0, \mathtt{5}) &= \overline{\epsilon\text{-close}}\left(\bigcup_{p \in \hat{\delta}_E(q_0,\epsilon)} \delta_N(p, \mathtt{5})\right) \\
&= \overline{\epsilon\text{-close}}(\delta_N(q_0, \mathtt{5}) \cup \delta_N(q_1, \mathtt{5})) = \overline{\epsilon\text{-close}}(\varnothing \cup \{q_1, q_4\}) \\
&= \{q_1, q_3, q_4, q_5\} \\
\hat{\delta}_E(q_0, \mathtt{5.}) &= \overline{\epsilon\text{-close}}\left(\bigcup_{p \in \hat{\delta}_N(q_0,\mathtt{5})} \delta_N(p, .)\right) \\
&= \overline{\epsilon\text{-close}}(\delta_N(q_1, .) \cup \delta_N(q_3, .) \cup \delta_N(q_4, .) \cup \delta_N(q_5, .))
\end{aligned}$$

Extended transition functions for ϵ-NFAs/Example (cont)

$$\begin{aligned}
\hat{\delta}_E(q_0, \mathtt{5.}) &= \overline{\epsilon\text{-close}}(\{q_2\} \cup \varnothing \cup \varnothing \cup \varnothing) = \{q_2\} \\
\hat{\delta}_N(q_0, \mathtt{5.6}) &= \overline{\epsilon\text{-close}}\left(\bigcup_{p \in \hat{\delta}_E(q_0,\mathtt{5.})} \delta_N(p, \mathtt{6})\right) \\
&= \overline{\epsilon\text{-close}}(\delta_N(q_2, \mathtt{6})) \\
&= \overline{\epsilon\text{-close}}(\{q_3\}) \\
&= \{q_3, q_5\} \ni q_5
\end{aligned}$$

Since q_5 is a final state, the string `5.6` is recognised as a number.

Subset construction for ϵ-NFAs

Let us present now how to construct a DFA from a ϵ-NFA such as both recognise the same language.

The method is a variation of the subset construction we presented for NFA: we must take into account the states reachable through ϵ-transitions, with help of ϵ-closures.

Subset construction for ϵ-NFAs (cont)

Assume that $\mathcal{E} = (Q, \Sigma, \delta, q_0, F)$ is an ϵ-NFA. Let us define as follows the equivalent DFA $\mathcal{D} = (Q_D, \Sigma, \delta_D, q_D, F_D)$.

1. Q_D is the set of subsets of Q_E. More precisely, all accessible states of $\mathcal{D}$ are ϵ-closed subsets of Q_E, i.e., sets $Q \subseteq Q_E$ such as $Q = \epsilon\text{-close}(Q)$.

2. $q_D = \epsilon\text{-close}(q_0)$, i.e., we get the start state of $\mathcal{D}$ by ϵ-closing the set made of only the start state of $\mathcal{E}$.

3. F_D is those sets of states that contain at least one final state of $\mathcal{E}$, that is to say $F_D = \{Q \,|\, Q \in Q_D \text{ and } Q \cap F_E \neq \varnothing\}$.

4. For all $a \in \Sigma$ and $Q \in Q_D$, let $\delta_D(Q, a) = \overline{\epsilon\text{-close}}\left(\bigcup_{q \in Q} \delta_E(q, a)\right)$

Subset construction for ϵ-NFAs/Example

Let us consider again the ϵ-NFA page 67. Its transition table is

$\mathcal{E}$	$+$	$-$	$0, \ldots, 9$	$.$	ϵ
$\to q_0$	$\{q_1\}$	$\{q_1\}$	$\varnothing$	$\varnothing$	$\{q_1\}$
q_1	$\varnothing$	$\varnothing$	$\{q_1, q_4\}$	$\{q_2\}$	$\varnothing$
q_2	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$	$\varnothing$
q_3	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$	$\{q_5\}$
q_4	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$
$\# q_5$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Subset construction for ϵ-NFAs/Example (cont)

By applying the subset construction to this ϵ-NFA, we get the table

$\mathcal{D}$	$+$	$-$	$0, \ldots, 9$	$.$
$\to \{q_0, q_1\}$	$\{q_1\}$	$\{q_1\}$	$\{q_1, q_3, q_4, q_5\}$	$\{q_2\}$
$\{q_1\}$	$\varnothing$	$\varnothing$	$\{q_1, q_3, q_4, q_5\}$	$\{q_2\}$
$\# \{q_1, q_3, q_4, q_5\}$	$\varnothing$	$\varnothing$	$\{q_1, q_3, q_4, q_5\}$	$\{q_2\}$
$\{q_2\}$	$\varnothing$	$\varnothing$	$\{q_3, q_5\}$	$\varnothing$
$\# \{q_3, q_5\}$	$\varnothing$	$\varnothing$	$\{q_3, q_5\}$	$\varnothing$

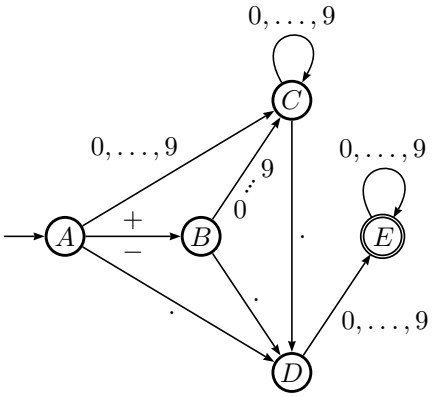
Subset construction for ϵ-NFAs/Example (cont)

Let us rename the states of $\mathcal{D}$ and get rid of the empty sets:

$\mathcal{D}$	$+$	$-$	$0,\ldots,9$	$.$
$\to A$	B	B	C	D
B			C	D
$\#C$			C	D
D			E	
$\#E$			E	

Subset construction for ϵ-NFAs/Example (cont)

The transition diagram of $\mathcal{D}$ is therefore



From regular expressions to ϵ-NFAs

We let behind the regular expressions when we introduced informally the transition diagrams for the token recognition.

Let us show now that regular expressions, used in lexers to specify tokens, can be converted to ϵ-NFAs, so to DFA. This proves that *regular languages are recognisable languages.*

Actually, it is possible to prove that any ϵ-NFA can be converted to a regular expression denoting the same language, but we will not do so.

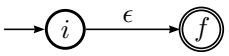
Therefore, keep in mind that **regular languages are recognisable languages**. In other words, using a regular expression or a finite automaton is only a matter of convenience.

From regular expressions to ϵ-NFAs (cont)

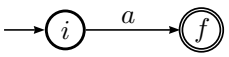
The construction we present here to build an ϵ-NFA from a regular expression is called **Thompson's construction**.

Let us first associate an ϵ-NFA to the basic regular expressions.

- For the expression ϵ, construct the following NFA, where i and f are **new** states



- For $a \in \Sigma$, construct the following NFA, where i and f are **new** states

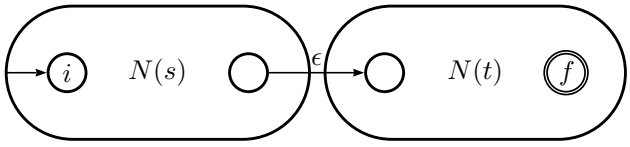


From regular expressions to ϵ-NFAs (cont)

Now let us associate NFAs to complex regular expressions.

Assume $N(s)$ and $N(t)$ are the NFAs for regular expressions s and t.

- For the regular expression st, construct the following NFA $N(st)$, where **no new state** is created:

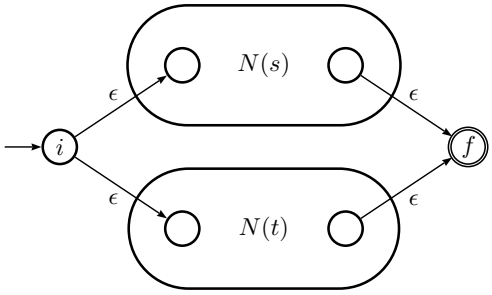


The final state of $N(s)$ becomes a normal state, as well as the initial state of $N(t)$.

This way only remains a unique initial state i and a unique final state f.

From regular expressions to ϵ-NFAs (cont)

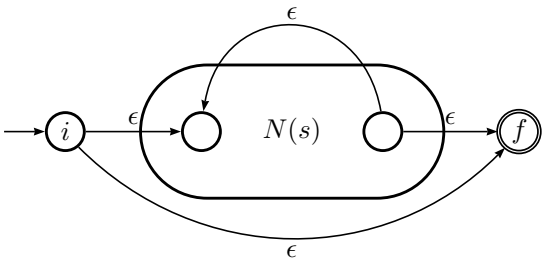
- For the regular expression $s \mid t$, construct the following NFA $N(s \mid t)$



where i and f are **new** states. Initial and final states of $N(s)$ and $N(t)$ become normal.

From regular expressions to ϵ-NFAs (cont)

- For the regular expression $s^\star$, construct the following NFA $N(s^\star)$, where i and f are **new** states:



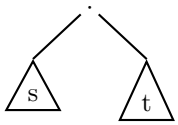
Note that we added two ϵ transitions and that the initial and final states of $N(s)$ become normal states.

From regular expressions to ϵ-NFAs (cont)

But how do we apply these simple rules when we have a complex regular expression, having many level of nested parentheses etc?

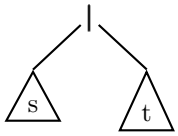
Actually, the **abstract syntax tree** of the regular expression direct, i.e., orders, the application of the rules.

If the syntax tree has the shape



then we construct first $N(s)$, $N(t)$ and finally $N(st)$.

If the syntax tree has the shape



then we construct first $N(s)$, $N(t)$ and finally $N(s \mid t)$.

From regular expressions to ϵ-NFAs (cont)

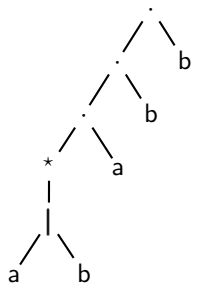
If the syntax tree has the shape

then we construct first $N(s)$ and finally $N(s^\star)$.

This pattern-matchings are applied first at the **root** of the abstract syntax tree of the regular expression.

From regular expressions to ϵ-NFAs/Exercise

Consider the regular expression $(\mathsf{a} \mid \mathsf{b})^\star\mathsf{abb}$ and its abstract syntax tree



Apply the previous rules to build the corresponding ϵ-NFA.