Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Talen en Compilers

2009/2010, periode 2

Andres Löh

Department of Information and Computing Sciences
Utrecht University

December 17, 2009

10. Regular languages

This lecture

Regular languages

- Regular languages
- Finite state automata
- NFAs vs. DFAs
- Regular grammars
- Regular grammars vs. finite state automata

10.1 Regular languages

Context-free languages

The languages we dealt with until now were mostly **context-free** languages:

- can be described using a context-free grammar,
- can be parsed relatively easily (for instance, using parser combinators),
- resulting parsers need polynomial time and space (often not much worse than linear).

Context-free languages

The languages we dealt with until now were mostly **context-free** languages:

- can be described using a context-free grammar,
- can be parsed relatively easily (for instance, using parser combinators),
- resulting parsers need polynomial time and space (often not much worse than linear).

The rest of the course: classes of languages and/or grammars that allow more efficient parsing.

Regular languages

A proper subset of the context-free languages:

- can be described using finite state automata,
- can be described using regular grammars,
- can be described using regular expressions,
- can be parsed very easily, in linear time and constant space.

Regular languages

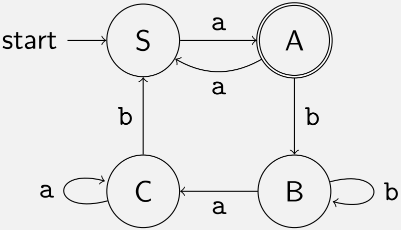
A proper subset of the context-free languages:

- can be described using finite state automata,
- can be described using regular grammars,
- can be described using regular expressions,
- can be parsed very easily, in linear time and constant space.

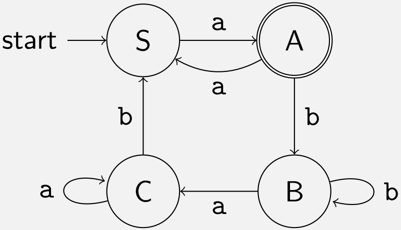
We will look at the different formalisms, their respective advantages and disadvantages, and show their equivalence.

10.2 Finite state automata

Deterministic finite state automata (DFA)

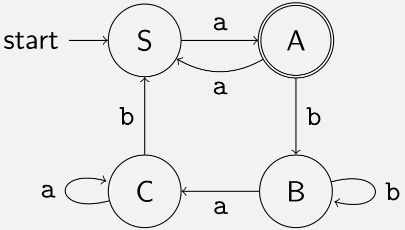


Deterministic finite state automata (DFA)



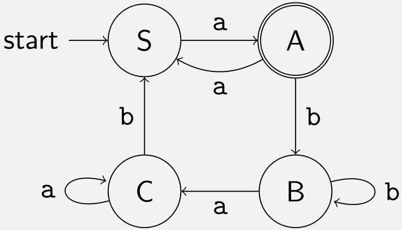
- Input alphabet:
  $X = \{\mathtt{a}, \mathtt{b}\}$

Deterministic finite state automata (DFA)



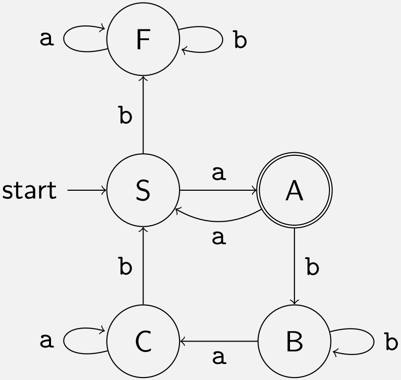
- Input alphabet: $X = \{\mathtt{a}, \mathtt{b}\}$
- States: $Q = \{S, A, B, C\}$

Deterministic finite state automata (DFA)



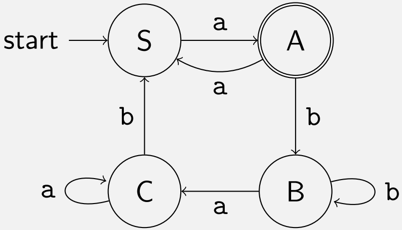
- Input alphabet:
  $X = \{\mathtt{a}, \mathtt{b}\}$
- States:
  $Q = \{S, A, B, C\}$
- Transitions:
  $d :: Q \rightarrow X \rightarrow Q$

Deterministic finite state automata (DFA)



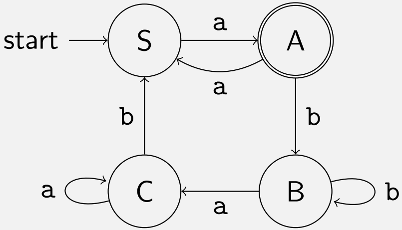
- Input alphabet:
  $X = \{a, b\}$
- States:
  $Q = \{S, A, B, C\}$
- Transitions:
  $d :: Q \to X \to Q$

Deterministic finite state automata (DFA)



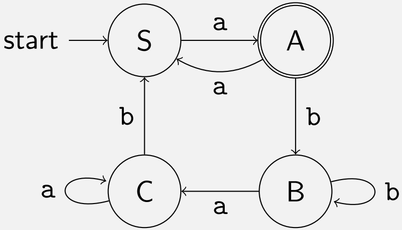
- Input alphabet: $X = \{\mathtt{a}, \mathtt{b}\}$
- States: $Q = \{S, A, B, C\}$
- Transitions: $d :: Q \to X \to Q$

Deterministic finite state automata (DFA)



- Input alphabet:
  $X = \{\mathtt{a}, \mathtt{b}\}$
- States:
  $Q = \{S, A, B, C\}$
- Transitions:
  $d :: Q \to X \to Q$
- Start state:
  S (where $S \in Q$)

Deterministic finite state automata (DFA)



- Input alphabet:
  $X = \{\mathtt{a}, \mathtt{b}\}$
- States:
  $Q = \{S, A, B, C\}$
- Transitions:
  $d :: Q \to X \to Q$
- Start state:
  S (where $S \in Q$)
- Accepting states:
  $F = \{A\}$
  (where $F \subseteq Q$)

Definition of a DFA

A DFA is given by

- an input alphabet X,
- a set of states Q,
- a transition function d of type $Q \to X \to Q$,
- a start state $S \in Q$,
- a set of accepting states $F \subseteq Q$.

Definition of a DFA

A DFA is given by

- an input alphabet X,
- a set of states Q,
- a transition function d of type $Q \to X \to Q$,
- a start state $S \in Q$,
- a set of accepting states $F \subseteq Q$.

Often, a DFA is simply written as a tuple (X, Q, d, S, F).

Definition of a DFA

A DFA is given by

- an input alphabet X,
- a set of states Q,
- a transition function d of type $Q \to X \to Q$,
- a start state $S \in Q$,
- a set of accepting states $F \subseteq Q$.

Often, a DFA is simply written as a tuple (X, Q, d, S, F).

Sometimes, when X and Q are clear from the context, (d, S, F) is sufficient to specify a DFA.

Running a DFA

$$
\begin{aligned}
&\mathsf{dfa} :: (\mathsf{Q} \to \mathsf{X} \to \mathsf{Q}) \to \mathsf{Q} \to [\mathsf{X}] \to \mathsf{Q}\\
&\mathsf{dfa}\ \mathsf{d}\ \mathsf{q}\ []\qquad\qquad = \mathsf{q}\\
&\mathsf{dfa}\ \mathsf{d}\ \mathsf{q}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{dfa}\ \mathsf{d}\ (\mathsf{d}\ \mathsf{q}\ \mathsf{x})\ \mathsf{xs}
\end{aligned}
$$

Running a DFA

$$
\begin{aligned}
&\mathsf{dfa} :: (\mathsf{Q} \to \mathsf{X} \to \mathsf{Q}) \to \mathsf{Q} \to [\mathsf{X}] \to \mathsf{Q}\\
&\mathsf{dfa\ d\ q}\ []\qquad\quad\;\; = \mathsf{q}\\
&\mathsf{dfa\ d\ q}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{dfa\ d}\ (\mathsf{d\ q\ x})\ \mathsf{xs}
\end{aligned}
$$

Question

Does this function look familiar?

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Running a DFA

$$
\begin{aligned}
&\mathsf{dfa} :: (Q \to X \to Q) \to Q \to [X] \to Q\\
&\mathsf{dfa}\ d\ q\ [\,] \qquad\quad = q\\
&\mathsf{dfa}\ d\ q\ (x : xs) = \mathsf{dfa}\ d\ (d\ q\ x)\ xs
\end{aligned}
$$

Question

Does this function look familiar?

$$\mathsf{dfa} = \mathsf{foldl}$$

Acceptance by a DFA

A word xs is **accepted** by a DFA if running the DFA on the word, starting in the start state S, yields an accepting state.

Acceptance by a DFA

A word xs is **accepted** by a DFA if running the DFA on the word, starting in the start state S, yields an accepting state.

$$\mathsf{dfaAccept} :: [\mathsf{X}] \rightarrow (\mathsf{Q} \rightarrow \mathsf{X} \rightarrow \mathsf{Q}, \mathsf{Q}, \mathsf{Set}\ \mathsf{Q}) \rightarrow \mathsf{Bool}$$
$$\mathsf{dfaAccept}\ \mathsf{xs}\ (\mathsf{d}, \mathsf{s}, \mathsf{fs}) = \mathsf{dfa}\ \mathsf{d}\ \mathsf{s}\ \mathsf{xs}\ \text{`member`}\ \mathsf{fs}$$

Language of a DFA

All words that are accepted by the DFA (d, S, F).

$$\{ w \in [X] \mid \text{dfaAccept } w \ (d, S, F) \}$$

Language of a DFA

All words that are accepted by the DFA (d, S, F).

$$\{w \in [X] \mid \text{dfaAccept } w\ (d, S, F)\}$$

One language can in general be described by multiple automata.

Exercise

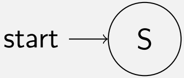
Question

Can the empty language be described by a DFA?

Exercise

Question

Can the empty language be described by a DFA?



Any automaton without accepting states is possible.

Exercise

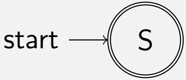
Question

Can the language $\{\varepsilon\}$ be described by a DFA?

Exercise

Question

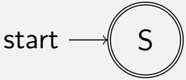
Can the language $\{\varepsilon\}$ be described by a DFA?



Exercise

Question

Can the language $\{\varepsilon\}$ be described by a DFA?



In general, any automaton where the starting state is accepting will accept the empty word (and possibly other words).

Observation

Running a DFA is clearly possible in linear time and constant space.

Nondeterministic finite state automata (NFA)

Similar to DFA, but:

- Potentially multiple start states.
- Potentially multiple transitions for the same terminal from a given state.

Nondeterministic finite state automata (NFA)

Similar to DFA, but:

- Potentially multiple start states.
- Potentially multiple transitions for the same terminal from a given state.

Formally:

- an input alphabet X,
- a set of states Q,
- a transition function d of type $Q \to X \to \text{Set } Q$,
- a **set of** start states $S \subseteq Q$,
- a set of accepting states $F \subseteq Q$.

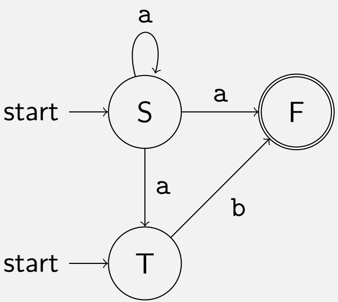
Interpretation using choices

- We can choose a start state.
- When processing a terminal, we can choose one of the possible transitions for that terminal at that state and thereby end up with a new state.
- A word is accepted if there is a sequence of choices that gets us to an accepting state.

Interpretation using a set of all possible choices

- At any time, a set of states in the NFA is active. We start with the set of start states.
- When we process a terminal, we take all possible actions from all current states and thereby end up with a new set of states.
- A word is accepted if the set of states that is active after processing the word contains at least one accepting state.

NFA example



Running an NFA

$$\begin{aligned}
&\mathrm{nfa} :: (Q \to X \to \mathrm{Set}\ Q) \to \mathrm{Set}\ Q \to [X] \to \mathrm{Set}\ Q\\
&\mathrm{nfa}\ d\ qs\ [\,] \qquad\quad = qs\\
&\mathrm{nfa}\ d\ qs\ (x : xs) = \mathrm{nfa}\ d\ (\mathrm{join}\ (\mathrm{map}\ (\mathrm{flip}\ d\ x)\ qs))\ xs
\end{aligned}$$

where join is the concat for sets and computes the union of a set of sets:

$$\mathrm{join} :: \mathrm{Set}\ (\mathrm{Set}\ Q) \to \mathrm{Set}\ Q$$

Acceptance by an NFA

$$\mathsf{nfaAccept} :: [\mathsf{X}] \rightarrow (\mathsf{Q} \rightarrow \mathsf{X} \rightarrow \mathsf{Set}\ \mathsf{Q}, \mathsf{Set}\ \mathsf{Q}, \mathsf{Set}\ \mathsf{Q}) \rightarrow \mathsf{Bool}$$
$$\mathsf{nfaAccept}\ \mathsf{xs}\ (\mathsf{d}, \mathsf{ss}, \mathsf{fs}) = \mathsf{not}\ (\mathsf{null}\ (\mathsf{nfa}\ \mathsf{d}\ \mathsf{ss}\ \mathsf{xs}\ \text{`}\mathsf{intersect}\text{`}\ \mathsf{fs}))$$

10.3 NFAs vs. DFAs

From DFA to NFA

Every DFA (d, S, F) is trivially an NFA.

From DFA to NFA

Every DFA (d, S, F) is trivially an NFA.

The start state S becomes the one-element set of start states $\{S\}$.

From DFA to NFA

Every DFA (d, S, F) is trivially an NFA.

The start state S becomes the one-element set of start states $\{S\}$.

The transition function is changed such that it returns singleton sets:

$$d' :: Q \rightarrow X \rightarrow \text{Set } Q$$
$$d' \ q \ x = \text{singleton } (d \ x)$$

From DFA to NFA

Every DFA (d, S, F) is trivially an NFA.

The start state S becomes the one-element set of start states $\{S\}$.

The transition function is changed such that it returns singleton sets:

$$\begin{aligned} & d' :: Q \to X \to \text{Set } Q \\ & d' \; q \; x = \text{singleton } (d \; x) \end{aligned}$$

It is quite easy to show that the resulting NFA accepts the same language.

From NFA to DFA

We can also make a DFA from an NFA.

Question

How?

From NFA to DFA

We can also make a DFA from an NFA.

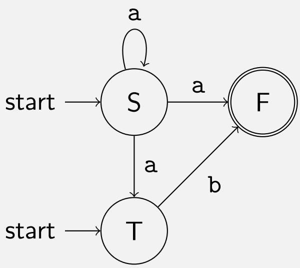
Question

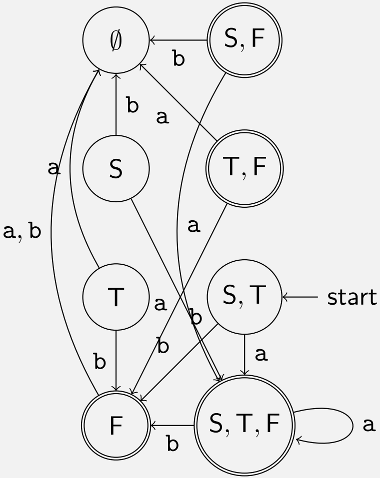
How?

The construction is called the **powerset construction**:

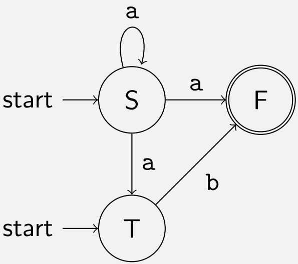
- For each **set of states** in the NFA, we get **one state** in the DFA.
- The set of start states in the NFA thus corresponds to a single state in the DFA.
- Since the transition function for the NFA takes sets of states to sets of states, we can then reuse it for the DFA.
- All states that contain an accepting state of the NFA become accepting states in the DFA.

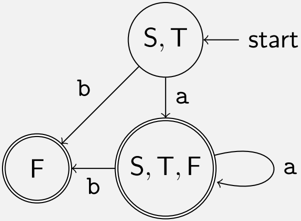
From NFA to DFA – example





From NFA to DFA – example





10.4 Regular grammars

Regular grammar

A context-free grammar G is called **regular** if all productions are of one of the following two forms:

> $A \rightarrow xB$
> $A \rightarrow x$

where x is a (possibly empty) sequence of terminals, and A and B are nonterminals.

Regular grammar

A context-free grammar G is called **regular** if all productions are of one of the following two forms:

$$A \to xB$$
$$A \to x$$

where x is a (possibly empty) sequence of terminals, and A and B are nonterminals.

In other words: Every right hand side has at most one nonterminal that must occur in the end.

Regular language

A language is called **regular** if it can be described by a regular grammar.

Regular vs. context-free

From the definition, it is immediately clear that every regular language is context-free.

Regular vs. context-free

From the definition, it is immediately clear that every regular language is context-free.

Question

Is every context-free language regular?

Regular vs. context-free

From the definition, it is immediately clear that every regular language is context-free.

Question

Is every context-free language regular?

No. The standard example is the language $\{\, \mathtt{a}^n \mathtt{b}^n \mid n \in \mathbb{N} \,\}$.

Regular vs. context-free

From the definition, it is immediately clear that every regular language is context-free.

Question

Is every context-free language regular?

No. The standard example is the language $\{ \mathtt{a}^n \mathtt{b}^n \mid n \in \mathbb{N} \}$.

We will investigate later how such a negative statement (not belonging to the class of regular languages) can be proved.

Closure properties

As context-free languages, regular languages are closed under

- union (corresponds to the $<|>$ combinator),
- sequencing (corresponds to the $<*>$ combinator),
- the star operator (corresponds to the many combinator).

Closure properties

As context-free languages, regular languages are closed under

- union (corresponds to the <|> combinator),
- sequencing (corresponds to the <*> combinator),
- the star operator (corresponds to the many combinator).

While context-free languages are not closed under intersection, regular languages are (werkcollege).

Simplifying regular grammars

For every regular language, there is a regular grammar that has no productions of the form

> $A \rightarrow B$
> $C \rightarrow \varepsilon$

where A, B, and C are nonterminals, except that for the start symbol there may be a production

> $S \rightarrow \varepsilon$

Simplifying regular grammars – contd.

The grammar transformation works in two phases:

- first all productions of the form $A \rightarrow B$ are removed;
- then all epsilon-productions are removed.

Simplifying regular grammars – contd.

Consider all pairs of nonterminals Y and Z.

If $Y \Rightarrow^* Z$:

- for every production $Z \rightarrow z$ (with z a sequence of symbols, but not a single nonterminal), add a production $Y \rightarrow z$.

If $Y \rightarrow Z$ is in the grammar, remove it.

Simplifying regular grammars – contd.

Consider all pairs of nonterminals Y and Z.

If $Y \Rightarrow^* Z$:

- for every production $Z \rightarrow z$ (with z a sequence of symbols, but not a single nonterminal), add a production $Y \rightarrow z$.

If $Y \rightarrow Z$ is in the grammar, remove it.

The only problematic productions left are now epsilon-productions.

Simplifying regular grammars – contd.

For each production $Y \rightarrow \varepsilon$, consider all productions $Z \rightarrow xY$ (where x now can consist only of terminals) and add a production $Z \rightarrow x$.

Then remove all epsilon-productions but $S \rightarrow \varepsilon$ if it exists.

Simplifying regular grammars – contd.

We can simplify a regular grammar even further and require that all productions are of one of the following two forms

$$\mathsf{Y} \rightarrow \mathtt{x}\mathsf{Z}$$
$$\mathsf{Y} \rightarrow \mathtt{x}$$

where $\mathtt{x}$ is thus a single terminal, except for the start symbol S, for which a production of $\mathsf{S} \rightarrow \varepsilon$ is allowed.

Simplifying regular grammars – contd.

We can simplify a regular grammar even further and require that all productions are of one of the following two forms

$$\mathsf{Y} \rightarrow \mathtt{x}\mathsf{Z}$$
$$\mathsf{Y} \rightarrow \mathtt{x}$$

where $\mathtt{x}$ is thus a single terminal, except for the start symbol S, for which a production of $\mathsf{S} \rightarrow \varepsilon$ is allowed.

The transformation works by introducing new nonterminals. For example

$$\mathsf{A} \rightarrow \mathtt{xy}\mathsf{C}$$

is transformed into

$$\mathsf{A} \rightarrow \mathtt{x}\mathsf{B}$$
$$\mathsf{B} \rightarrow \mathtt{y}\mathsf{C}$$

10.5 Regular grammars vs. finite state automata

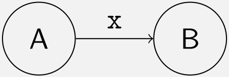
From NFA to regular grammars

For each NFA, there exists a regular grammar that describes the same language.

From NFA to regular grammars

For each NFA, there exists a regular grammar that describes the same language.

- the states become nonterminals,
- the start state becomes the start symbol,
- for each transition



we introduce a production

> $A \rightarrow xB$

- for each accepting state F we introduce a production

> $A \rightarrow \varepsilon$

From regular grammars to an NFA

We can also produce an automaton for every regular grammar.

From regular grammars to an NFA

We can also produce an automaton for every regular grammar.

We first simplify the grammar. Then, all the hard work is done.