

Decision, Computation and Language

INT201 (*CSE203*)

Dr. Wenjin Lu

2021-2022

Department of Computer Science and Software Engineering
Xi'an Jiaotong-Liverpool University

Decision, Computation and Language

This module is about the theory of computation and answering the question: "**What are the fundamental capabilities and limitations of computers?**".

Given a set L , and an element x , the membership problem w.r.t. (with respect to) L is a decision problem: "Is x a member of L ?"

We say the membership problem can be solved algorithmically (computed) if there is an algorithm A^L , for any input x , if x is a member of L , then the output will be "Yes", otherwise the output is "No"

In this module, a formal language is a special set of strings over a alphabet (formal definition is given late)

The central theme: Are there languages, their membership problems can not be solved (computed) Algorithmically

Languages

Spoken languages and programming languages: in both cases we like to know whether a sequence of symbols belongs to the language.

Formal Languages have the property that there is a precise rule that governs what strings belong to the language.

Formal languages include programming languages, database query languages, various file formats. (so, in the world of computers, they are everywhere...)

By contrast, English, French etc. are not formal languages, although you can still try to write down rules that work most of the time.

How we usually describe or teach syntax

Informal descriptions:

“An arithmetic expression is constructed from variables and numbers, and infix operators `+`, `-`, `*`, `/`, and subexpressions may possibly be enclosed in parentheses, in which case every `(` must have a corresponding `)` ...”

“A comment begins with `/*` and ends with `*/`”

“Your password should have 4-8 characters and contain a non-alphabetic character.”

How we usually describe or teach syntax

By examples:

Let E denote the set of arithmetic expressions

$$E = \{x, 1 + (2 - x), x * y + ((z)), \dots\}$$

a42 is a valid variable name; 42a is not, because a variable name can't start with a number.

(The variable-name description is a combination of English and examples.) We need some notation to express these descriptions more precisely!

What the module is about

notations for representing *formal languages*. These give us

- ways to define them precisely
- ways to build compilers that recognise the languages
- ways to check whether
 - ▶ a string of symbols belongs to a language
 - ▶ whether two alternative descriptions of languages are actually the same language

What the module is about

Tools to analyse languages

These tools originate in analysis of natural languages (NL) as well as programming languages (PL).

NL: want to recognise valid sentence

PL: want to recognise valid program

Sentences can be arbitrarily long, so you can't list them. A list is infeasible even if you limit the length to some "reasonable" amount, also not very enlightening.

What the module is about

Some notations/methods for describing a language (other than explicitly listing it) include

- finite state automaton
- regular expressions
- context-free grammar
- pushdown Automation
- Turing machine

We will see that some of the above can describe more languages than others (More powerful than others).

¹but there are problems if we impose no restrictions on the programs...

Grammars

Sets of rules for generating syntactically correct programs/sentences.

A grammar for generating some English sentences:

$\langle S \rangle \rightarrow \langle S \rangle \text{ and } \langle S \rangle$

$\langle S \rangle \rightarrow \langle \text{subject phrase} \rangle \langle \text{Verb} \rangle \langle \text{object phrase} \rangle$

$\langle \text{subject phrase} \rangle \rightarrow \langle \text{subject pronoun} \rangle$

$\langle \text{subject phrase} \rangle \rightarrow \langle \text{Article} \rangle \langle \text{Noun} \rangle$

$\langle \text{object phrase} \rangle \rightarrow \langle \text{object pronoun} \rangle$

$\langle \text{object phrase} \rangle \rightarrow \langle \text{Article} \rangle \langle \text{Noun} \rangle$

$\langle \text{subject pronoun} \rangle \rightarrow \text{he} \mid \text{she}$

$\langle \text{object pronoun} \rangle \rightarrow \text{him} \mid \text{her} \mid \text{me}$

$\langle \text{Article} \rangle \rightarrow \text{a} \mid \text{the}$

$\langle \text{Noun} \rangle \rightarrow \text{dog} \mid \text{cat} \mid \text{mouse} \mid \text{house}$

$\langle \text{Verb} \rangle \rightarrow \text{sees} \mid \text{likes} \mid \text{finds} \mid \text{leaves}$

Observations

- The grammar can generate sentences like “the dog sees the cat and the mouse leaves the house”, which may be nonsense e.g. “the house sees me”.
- Arbitrarily long sentences can be generated (which you can't do by enumeration!)
- Semantics can be given with reference to grammar, e.g. logical conjunction of subsentences formed by word “and”
- The grammar could be extended to handle subordinate clauses, adverbs etc.
- You can't define natural language sentences completely this way, but you can for programming languages

Applications in programming languages

Stages of compilation:

- ① **lexical analysis**: divide sequence of characters into *tokens*, such as variable names, operators, labels. In a natural language tokens are strings of consecutive letters (easy to recognise!)
- ② **parsing**: identify relationships between tokens
- ③ code generation: generate object code
- ④ code optimisation

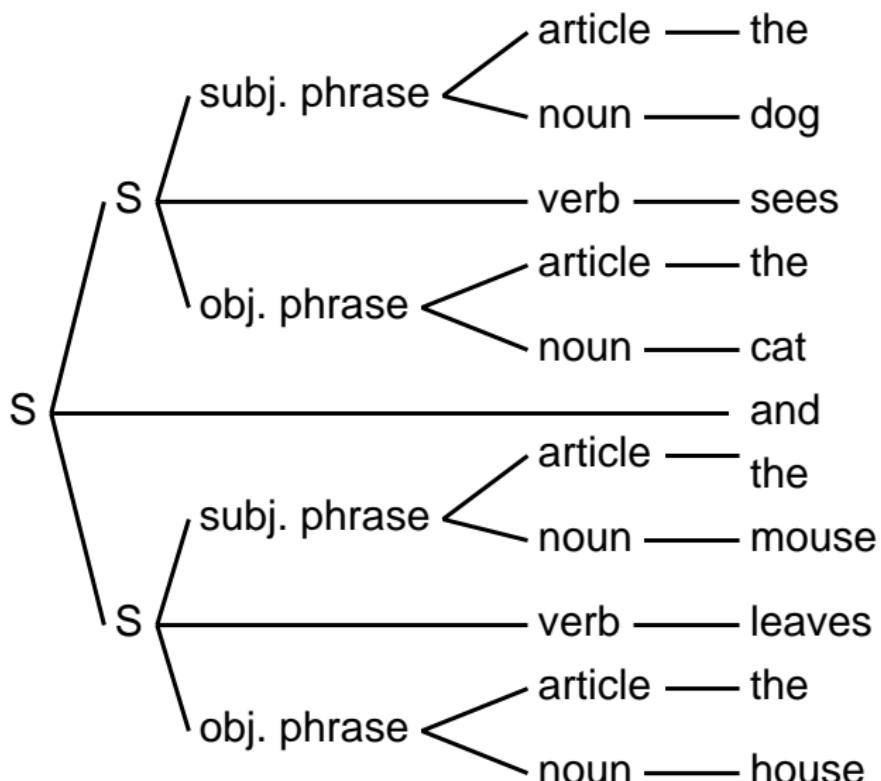
Lexical Analysis

```
pay=salary+(overtimerate*overtime);
```

Break into tokens as follows:

```
pay  
=  
salary  
+  
(  
overtimerate  
*  
overtime  
)  
;
```

Parse Tree



Definitions and notation

- An *alphabet* is a finite set of symbols.
- A *word* over alphabet A is a string of symbols belonging to A .
- The *empty word* will be denoted ϵ
- A^* denotes the set of *all* words over A
- A^+ denotes the set of all non-empty words over A

Definitions and notation

The *concatenation* (a.k.a.² "product") of two words is obtained by appending them together to form one long word.

Concatenation of words w_1 and w_2 can be written $w_1 w_2$.

For any word w , note that $w\epsilon = \epsilon w = w$.

Concatenation is associative.

w^n denotes the concatenation of n copies of w .

$|w|$ denotes the length (number of letters) of w .

$$|w_1 w_2| = |w_1| + |w_2|$$

²also known as

For $w \in A^*$, the *reverse* of w is denoted w^R and consists of w 's letters in reverse order.

A *palindrome* is a word w satisfying $w = w^R$.

If u, v, w are words and $w = uv$ then u is a *prefix* of w and v is a *suffix* of w . A *proper prefix* of w is a prefix that is not equal to ϵ or w .
(Similarly for proper suffix)

Languages

A *language* (or *formal language*) over alphabet A is a subset of A^* .

We can express new languages in terms of other languages using concatenation and closure.

$$L_1 L_2 = \{w_1 w_2 : w_1 \in L_1 \text{ and } w_2 \in L_2\}.$$

$$L^* = \{w_1 w_2 \dots w_n : n \geq 0 \text{ and } w_1, w_2, \dots, w_n \in L\}$$

Alphabet A ,

$A^* = \{\text{all strings consisting symbols from } A\}$

A subset L of A^* is called a language

Question:

For a language L , does exist an algorithm to check for any x , if x is in L ?

For example:

$A = \{0, 1\}$

A^* is the set of all 0, 1 strings including empty string ϵ

Some languages over A :

$L_1 = \{\}$

$L_2 = \{\epsilon\}$

$L_3 = \{a, b\}$

$L_4 = \{w : w \text{ ends with } b\}$

$L_5 = \{ab, aabb, aaabbb, aaaabbbb, \dots\}$

Deterministic Finite Automata

It is invented to recognise a special class of formal languages but can be used in

lexical analysis — scan program from beginning to end and divide it into tokens, specify tokens of programming languages.

“model checking”, reasoning about systems with objective of proving they satisfy useful properties.

statistical models for analysing biological and textual sequences.

Deterministic Finite Automata

Simple mechanism for scanning a string

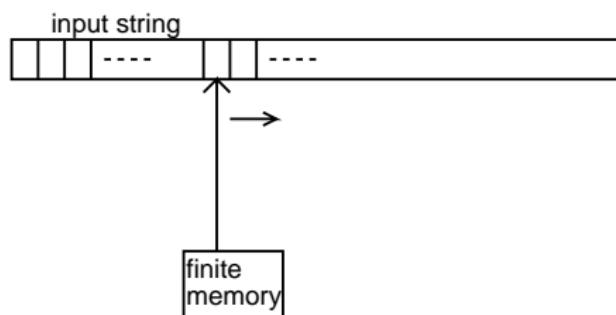


Figure: finite automaton

Gives method of recognising words belonging to certain languages
(must accept or fail to accept at end of string)

Deterministic Finite Automata

We shall see that finite automata can be used to describe

- any finite set of strings
- various infinite sets of strings, e.g.
 - ▶ strings having exactly 2 occurrences of the letter **a**
 - ▶ strings having more than 6 letters
 - ▶ strings in which letter **b** never comes before letter **a**

Deterministic Finite Automata

Furthermore we shall see that finite automata cannot be used to describe certain languages such as

- the set of strings containing more a 's than b 's
- all words that remain the same if you read them back to front
- well-formed arithmetic expressions, if there is no limit on nesting of parentheses.

A *deterministic finite automaton* (DFA) has 5 components:

- ① Q is a finite nonempty set whose members are called *states* of the automaton;
- ② A is a finite nonempty set called the *alphabet* of the automaton;
- ③ ϕ is a map from $Q \times A$ to Q called the *transition function* of the automaton;
- ④ i is a member of Q and is called the *initial state*;
- ⑤ T is a nonempty subset of Q whose members are called *terminal states* or *accepting states*.

“quintuple” — any DFA can be divided into these 5 components

state of a machine tells you something about the prefix that has been read so far. If the string is a member of the language of interest, the state reached when the whole string has been scanned will be an accepting state (a member of T).

There is only one empty string so there is only one initial state (denoted i)

Transition function ϕ tells you how state should change when an additional letter is read by the DFA

A DFA is often depicted as a labelled directed graph. (called *transition diagram*)

Initially the state is i and if the input word is $w = a_1 a_2 \dots a_n$ then, as each letter is read, the state changes and we get q_1, q_2, \dots, q_n defined by

$$\begin{aligned} q_1 &= \phi(i, a_1) \\ q_2 &= \phi(q_1, a_2) \\ q_3 &= \phi(q_2, a_3) \\ &\vdots \quad \vdots \quad \vdots \\ q_n &= \phi(q_{n-1}, a_n) \end{aligned}$$

Extend the definition of the transition function so that it tells us which state we reach after a word (not just a single letter) has been scanned:

In the above notation, extend the map $\phi : Q \times A \rightarrow Q$ to
 $\phi : Q \times A^* \rightarrow Q$ by defining:

$$\begin{aligned}\phi(q, \epsilon) &= q && \text{for all } q \in Q \\ \phi(q, wa) &= \phi(\phi(q, w), a) && \text{for all } q \in Q; w \in A^*; \\ &&& a \in A.\end{aligned}$$

It is easy to show by induction that this extended map satisfies

$$\phi(q, vw) = \phi(\phi(q, v), w) \text{ for all } q \in Q; v, w \in A^*.$$

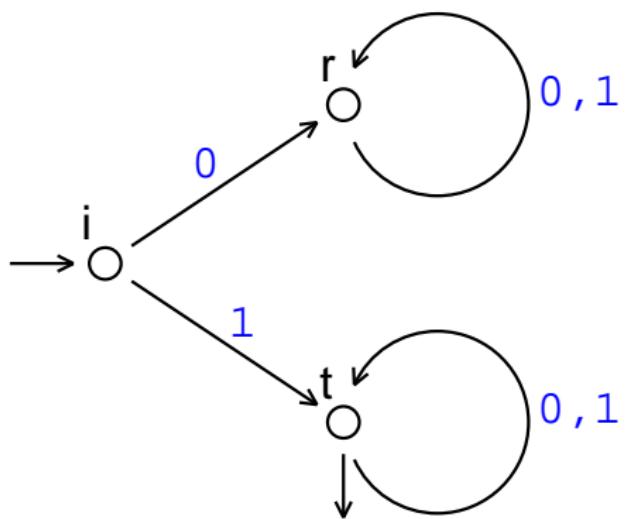
Language defined by a DFA

Suppose we have a DFA \mathcal{A} .

A word $w \in A^*$ is said to be *accepted* or *recognised* by \mathcal{A} if $\phi(i, w) \in T$, otherwise it is said to be rejected. The set of all words accepted by \mathcal{A} is called the *language accepted by \mathcal{A}* and will be denoted by $L(\mathcal{A})$. Thus

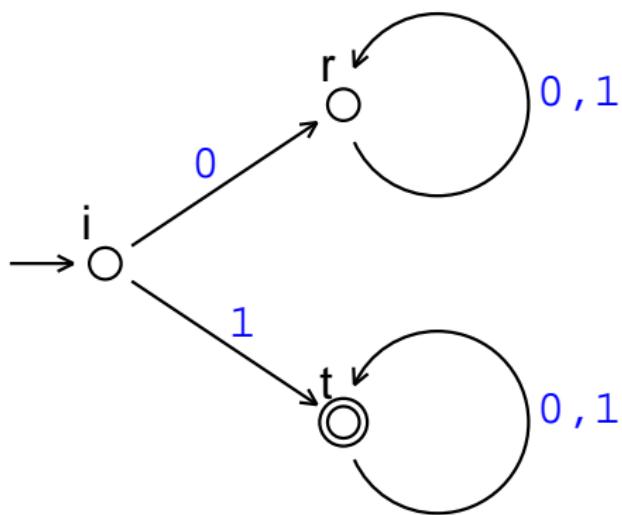
$$L(\mathcal{A}) = \{w \in A^* : \phi(i, w) \in T\}.$$

Example (transition diagram)



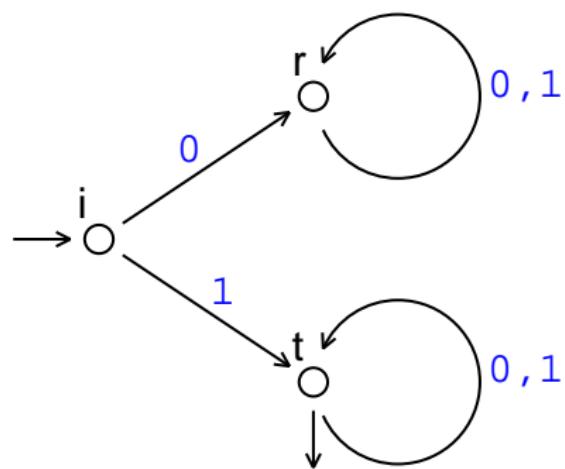
3 states, i , r and t . Accepting state t has outgoing arrow.

Example (transition diagram)



Alternative notation for accepting state is 2 concentric circles.

Example computation



Input word 110100

symbol	1	1	0	1	0	0
state	i	t	t	t	t	t

Symbolic description of the example DFA

Automaton $\mathcal{A} = (Q, A, \phi, i, T)$

Set of states $Q = \{i, t, r\}$, $A = \{\textcolor{blue}{0}, \textcolor{blue}{1}\}$, $T = \{t\}$ and the transition function ϕ is given by

$$\begin{array}{lll} \phi(i, \textcolor{blue}{0}) = r, & \phi(i, \textcolor{blue}{1}) = t, \\ \phi(t, \textcolor{blue}{0}) = t, & \phi(t, \textcolor{blue}{1}) = t, \\ \phi(r, \textcolor{blue}{0}) = r, & \phi(r, \textcolor{blue}{1}) = r. \end{array}$$

It is simpler to describe a transition function by a table of values. In this example we have:

	0	1
i	r	t
t	t	t
r	r	r

A simplification

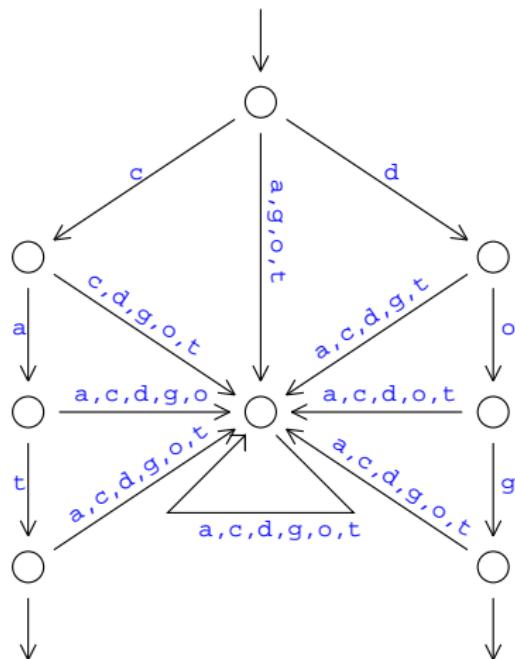
If ϕ is a partial function (not defined for some state/letter pairs), then the DFA rejects an input if it ever encounters such a pair.

This convention often simplifies the definition of a DFA. In the previous example we could use transition table

	0	1
i		t
t	t	t

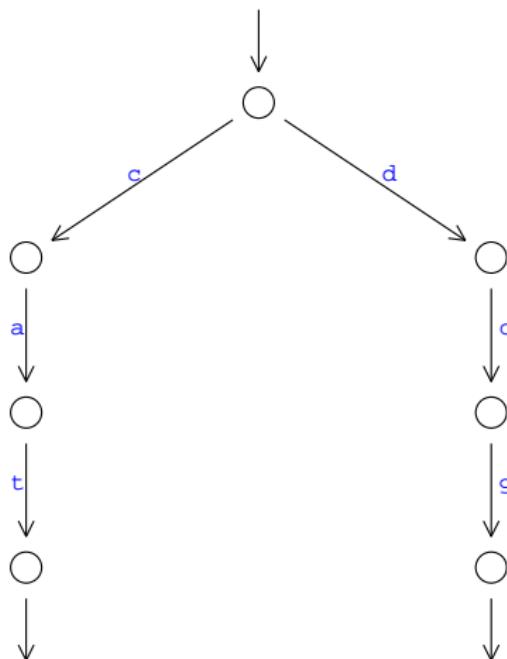
Example

Give a DFA that accepts the words “cat” and “dog” (using the 6-letter alphabet a, c, d, g, o, t).



Example

The DFA can be “stripped down” if we understand an undefined transition to mean: reject the whole string.



An important observation

Any DFA that uses the convention that an undefined transition leads to a rejection, can be converted to a DFA that uses a *total* transition function (that is, one that is defined for all combinations of input symbols and states).

The convention is useful, but it does not add extra *expressive power*.

Exercises

Draw a diagram for a DFA that only accepts the words:

fun, fair, unfair, funfair

Note. this language can be represented by simple enumeration: in set-theoretic notation

$$\{ \textit{fun}, \textit{fair}, \textit{unfair}, \textit{funfair} \}$$

How about the following infinite language. Can you give a DFA that accepts the words:

bad, baad, baaad, baaaad, ...?

Any finite language is accepted by some DFA

General rule: i is the initial state. For each prefix p of a word in the list, include state s_p with the idea that the machine should be in state s_p after p has been read.

If p is equal to one of the given words, make s_p an accepting state.

For prefixes p and q with $q = p\text{a}$ ($\text{a} \in A$), let $\phi(s_p, \text{a}) = s_q$

If $p\text{a}$ is not a prefix of any word, $\phi(s_p, \text{a}) = f$ (failure state)

For all $\text{a} \in A$, $\phi(i, \text{a}) = s_{\text{a}}$.

Try using this rule to construct a DFA that accepts
 $\text{a, cat, cats, dog, deer}$

General observations

Any finite language (and various infinite languages) have DFAs that recognise them.

Question: What sort of languages can be recognised by DFAs?

Note: although finite languages can be enumerated, it may be advantageous to describe them using a DFA (e.g. words of length 10)

Example

Construct a DFA that accepts words over the alphabet $\{a, b\}$ which contain an odd number of a 's and an even number of b 's.

State should keep track of parity of number of occurrences of each letter seen so far.

So: this suggests using 4 states, called E/E , E/O , O/E , O/O where $i = E/E$.

$$\phi(E/E, a) = O/E$$

$$\phi(E/E, b) = E/O$$

etc.

$$T = \{O/E\}$$

More examples

Write down DFAs which recognise the following languages over the alphabet $\{a, b\}$:

L_1 is set of all words containing exactly three occurrences of a

L_2 is set of all words containing at least three occurrences of a

L_3 is set of words containing the substring aaa

Alphabet A ,

$A^* = \{\text{all strings consisting symbols from } A\}$

A subset L of A^* is called a language

Question:

For a language L , does exist an algorithm to check
for any x , if x is in L ?

For example:

$A = \{0, 1\}$

A^* is the set of all 0, 1 strings including empty
string ϵ

Some languages over A :

$L_1 = \{\}$

$L_2 = \{\epsilon\}$

$L_3 = \{0, 1\}$

$L_4 = \{w: w \text{ ends with } 1\}$

$L_5 = \{01b, 0011, 000111, 00001111, \dots\}$

DFA $M = (Q, A, \delta, i, T)$

Where

$Q:$...

$A:$...

$\delta: Q \times A \rightarrow Q$

$i:$...

$T:$...

Extension:

$\delta: Q \times A^* \rightarrow Q$

Nondeterministic finite automata

Recall: we took original DFA definition and extended that definition to allow some transitions to be undefined. Nondeterministic Finite Automata (NFA) are a further extension...

Useful for

- describing a formal language (may be simpler than equivalent DFAs)
- conversion between finite automata and *regular expressions*

Nondeterministic finite automata

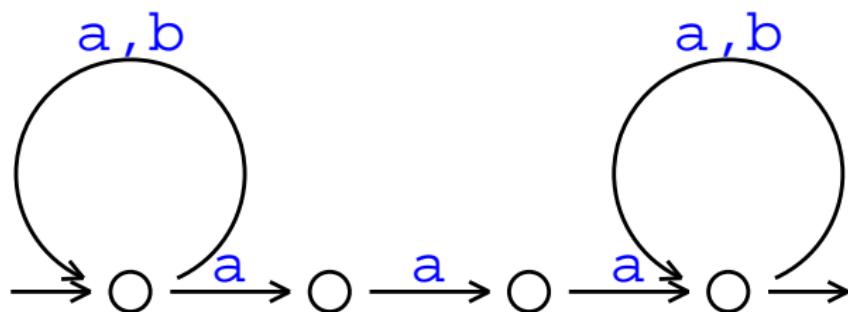
“nondeterministic” – an input string does not determine the final state.

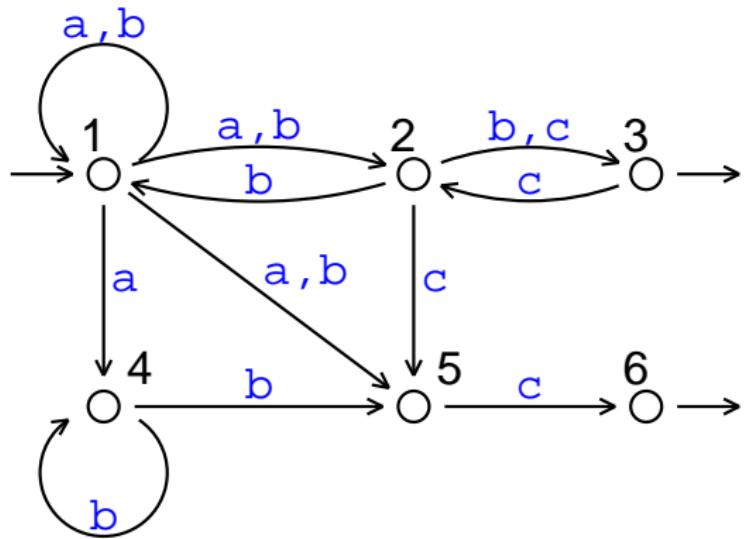
This is because given a current state and an input letter, we may specify a set of allowable new states, not just one.

Then we say that a NFA accepts an input word w if there exists a sequence of transitions labelled by symbols in w , starting from initial state and ending at some accepting state.

Example

Strings over alphabet $\{a, b\}$ containing aaa as a substring (i.e. three consecutive a's)





The NFA above accepts (amongst others) the words **ab**, **abc**, **aabbcc** (there's a choice of transition sequences for the second and third words...)

The formal definition

Let $\mathcal{P}(S)$ denote the set of all subsets of a set S . A *nondeterministic finite automaton* (or NFA) is a quintuple $\mathcal{A} = (Q, A, \phi, i, T)$ where

- ① Q is a finite nonempty set whose members are called *states* of the automaton;
- ② A is a finite nonempty set called the *alphabet* of the automaton;
- ③ ϕ is a map from $Q \times A$ to $\mathcal{P}(Q)$ called the *transition function* of the automaton;
- ④ i is a member of Q and is called the *initial state*;
- ⑤ T is a nonempty subset of Q whose members are called *terminal states* or *accepting states*.

Extend the map ϕ to a map $Q \times A^* \rightarrow \mathcal{P}(Q)$ by defining

$$\begin{aligned}\phi(q, \epsilon) &= \{q\} && \text{for all } q \in Q \\ \phi(q, wa) &= \bigcup_{p \in \phi(q, w)} \phi(p, a) && \text{for all } q \in Q; \\ &&& w \in A^*; a \in A.\end{aligned}$$

Thus $\phi(q, w)$ is the set of all possible states that can arise when the input w is received in the state q . w is accepted provided that $\phi(q, w)$ contains an accepting state.

Notation: accepting/rejecting paths

Suppose, in a DFA, we can get from state p to state q via transitions labelled by letters of a word w . Then we say that the states p and q are connected by a path with label w .

If $w = \text{abc}$ and the 2 intermediate states are r_1 and r_2 we could write this as

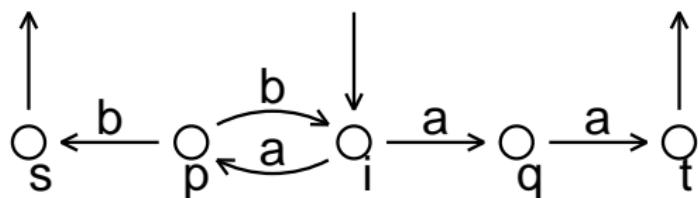
$$p \xrightarrow{a} r_1 \xrightarrow{b} r_2 \xrightarrow{c} q$$

In a NFA, if $\phi(p, a) = \{q, r\}$ we could write

$$\{p\} \xrightarrow{a} \{q, r\}$$

and this would be an accepting path if any state on the RHS is an accepting state, otherwise it would be rejecting path.

Example (in handout)



Input string abaa is accepted. Path:

$$\begin{aligned} \{i\} &\xrightarrow{a} \{p, q\} \xrightarrow{b} \{i, s\} \\ &\xrightarrow{a} \{p, q\} \xrightarrow{a} \{t\} \end{aligned}$$

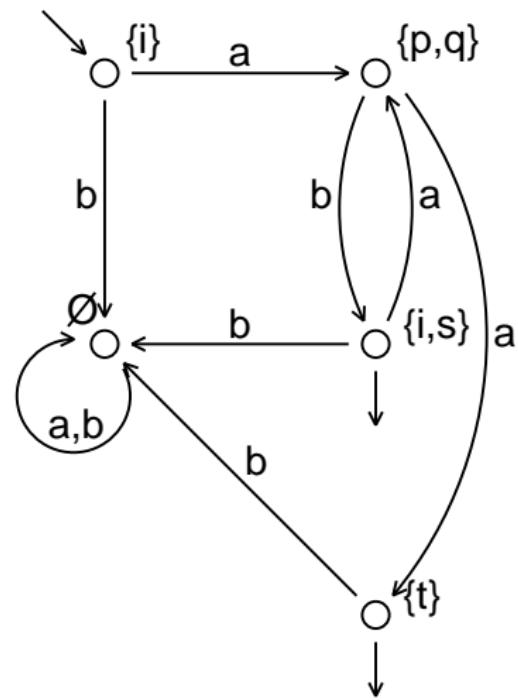
“correct choice” of new state:

$$i \xrightarrow{a} p \xrightarrow{b} i \xrightarrow{a} q \xrightarrow{a} t$$

How should you test whether a particular string should be accepted by some given NFA?

At each step, keep track of set of allowable states. This methods helps to explain how an NFA can be translated into an equivalent DFA (showing that the same kinds of languages are accepted).

Equivalent DFA



A DFA is a NFA...

...in the sense that a DFA is a restricted form of NFA. In a DFA, the set of all new states that can be reached in a single transition from any given state and any letter in the alphabet, is of size 0 or 1. (In a NFA, we allow the set of new states to be larger.)

Hence, any language accepted by a DFA is accepted by an NFA.

Next: prove that any language accepted by a NFA is accepted by some DFA.

Theorem. *A language which is accepted by a nondeterministic finite automaton is also accepted by some deterministic finite automaton.*

Proof. Let $\mathcal{A} = (Q, A, \phi, i, T)$ be a nondeterministic finite automaton. We define a deterministic finite automaton \mathcal{A}' as follows. The alphabet of \mathcal{A}' is the same as that of \mathcal{A} . The set of states of \mathcal{A}' is $\mathcal{P}(Q)$. The transition function of \mathcal{A}' is the map $\psi : \mathcal{P}(Q) \times A \rightarrow \mathcal{P}(Q)$ defined by

$$\psi(P, a) = \bigcup_{p \in P} \phi(p, a).$$

For the initial state of \mathcal{A}' we take $\{i\} \in \mathcal{P}(Q)$.

$$w \in L(\mathcal{A}) \iff \phi(i, w) \cap T \neq \emptyset \iff \psi(\{i\}, w) \cap T \neq \emptyset.$$

Hence defining T' by

$$T' = \{P \in \mathcal{P}(Q) : P \cap T \neq \emptyset\}$$

we have now specified a deterministic automaton

$$\mathcal{A}' = (\mathcal{P}(Q), A, \psi, \{i\}, T')$$
 and

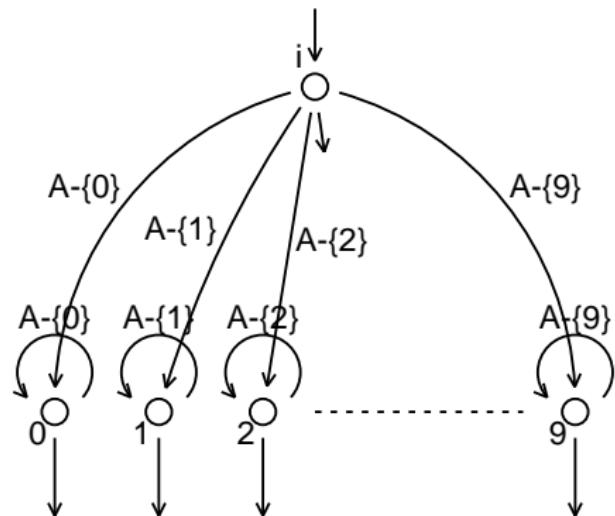
$$\begin{aligned} w \in L(\mathcal{A}') &\iff \psi(\{i\}, w) \in T' \\ &\iff \psi(\{i\}, w) \cap T \neq \emptyset \iff w \in L(\mathcal{A}). \end{aligned}$$

We conclude $L(\mathcal{A}) = L(\mathcal{A}')$.

A NFA may have many fewer states than the smallest equivalent DFA

alphabet $A = \{0, 1, 2, \dots, 9\}$

Let L denote the language over A of all words which do not contain at least one occurrence of each digit.



L denotes the language over A of all words which do not contain at least one occurrence of each digit.

Claim: a DFA $\mathcal{A} = (Q, A, \phi, i, T)$ that accepts L needs at least 2^{10} states.

Proof: Let $a \in \{0, 1, 2, \dots, 9\}$.

Suppose there are words w_1, w_2 where $a \in w_1$ but $a \notin w_2$, such that $\phi(i, w_1) = \phi(i, w_2)$.

We will prove by contradiction that no such w_1, w_2 exist.

Let w_3 denote any word consisting of the symbols in $A \setminus \{a\}$.

From definition of L , $w_1 w_3 \notin L$ but $w_2 w_3 \in L$.

But $\phi(i, w_1 w_3) = \phi(\phi(i, w_1), w_3) = \phi(\phi(i, w_2), w_3) = \phi(i, w_2 w_3)$, a contradiction.

Our assumption that $\phi(i, w_1) = \phi(i, w_2)$ was incorrect.

So if we have words w_1 and w_2 , and the sets of letters that can be found in each of w_1 and w_2 are different, then we must have $\phi(i, w_1) \neq \phi(i, w_2)$. There are 2^{10} distinct subsets of A , hence (at least) 2^{10} distinct states.

Regular expressions

A regular expression (r.e.) is a way of describing a language. It can consist of a finite set of words, like

$$\{ \textit{cat}, \textit{dog}, \textit{mouse}, \epsilon \}$$

which represents the 4 words that are listed.

Then, there are 3 operators that can appear in r.e's: they are

- Union
- concatenation
- closure

these operators allow you to glue together subexpressions to form larger expression.

Union and Concatenation

Union: Any r.e. represents a set of words (its language) and we may use \cup to connect 2 subexpressions into a larger regular expression, e.g.

$$\{cat, dog, mouse, \epsilon\} \cup \{cat, cats\}$$

which represents the 5 words ϵ , cat, cats, dog, mouse.

Concatenation

By joining two r.e's together, we denote the set of words you can make by taking a word from the first r.e. and concatenating it to some word from the second r.e., e.g.:

$$\{over, under\}\{cooked, state, rate\}$$

denotes the words overcooked, undercooked, overstate, understate, overrate, underrate.

Closure (also called "Kleene closure")

If we take a regular expression and add the superscript $*$, we get a new r.e. that represents the set of all words you can make by taking any sequence of words from the original r.e. and concatenating them together.

Example:

$$(\{cat\})^*$$

denotes the words ϵ , cat, catcat, catcatcat, catcatcatcat, ...

Notice that using closure, you can define an infinite language!

Examples

$$(\{cat, dog\})^*$$

denotes the words ϵ , cat, dog, catcat, catdog, dogcat, dogdog,
catcatcat, catcatdog, catdogcat, ...

$$a(\{a, b\})^*$$

is the set of all words over alphabet {a, b} that begin with the letter a.

We may write this as $a\{a, b\}^*$, using the convention that closure is applied to the shortest regular subexpression that precedes it.

Examples

$$\{a, b\}^* \{c, d\}^*$$

denotes words such as ababababcccdcccd,
abbbaabbaaaccccdcd, ...

Write down a r.e. for strings of a's and b's where all the a's must come before all the b's.

Write down an expression for the above language, assuming that the empty word is not allowed to belong to the language.

Examples

$\{a, b\}^*aaa\{a, b\}^*$ represents strings containing aaa as substring.

Assuming that alphabet $A = \{a, b\}$, the above can be written as

$$A^*aaaA^*.$$

$b^*ab^*ab^*ab^*$: strings with 3 a 's.

$A^*aA^*aA^*aA^*$: strings with at least 3 a 's.

How about strings with an even number of a 's and any number of b 's?
(there's a 2-state FA that accepts this language.)

Common extensions to the notation

Let E denote a regular expression.

$(E)^n$ denotes concatenations of n words generated by E . (Could be written $EEE \dots E$ (n times))

$(E)^+$ denotes concatenations of at least one word generated by E (Could be written EE^* or E^*E).

The above give no extra expressive power in terms of what languages can be described.

Lexical tokens

We can use regular expressions for short, precise definitions of lexical tokens:

“variable: string of letters/digits starting with a letter”

r.e.: $\{a,b,\dots,z,A,B,\dots,Z\} \{a,b,\dots,z,A,B,\dots,Z, 0,1,2,\dots,9\}^*$

number in exponential/scientific notation ("1.16121122E-03" denotes $1.16121122 \times 10^{-3}$, i.e. 0.00116121122)

$\{1,2,3,\dots,9\}.\{0,1,2,\dots,9\}^8 E \{00,$
 $\{\epsilon, -\}\{\{1,2,\dots,9\}\{0,1,2,\dots,9\}, \{0,1,2,\dots,9\}\{1,2,\dots,9\}\}}$

Regular Languages...

...are those languages that can be described using regular expressions. We will see that these are the same languages as those that can be accepted by finite automata (Kleene's theorem).

In terms of description length some languages are “better” expressed as DFAs and some as reg exprs

r.e’s usually give a easy-to-understand description, on the other hand it’s obvious how to run DFAs on input strings.

A language that is easier to describe using a regular expression

Alphabet $A = \{0, 1, 2, \dots, 9\}$

$L \subset A^*$ is defined to be words which contain an even number of occurrences of at least one of the digits $0, 1, 2, \dots, 9$

So the word 0123456789 (or any permutation of it) is the shortest word not in L (No occurrence is treated as even occurrence).

The regular expression

$\{ \{1, 2, \dots, 9\}^* 0 \{1, 2, \dots, 9\}^* 0 \{1, 2, \dots, 9\}^* \}^* \cup \{1, 2, \dots, 9\}^*$

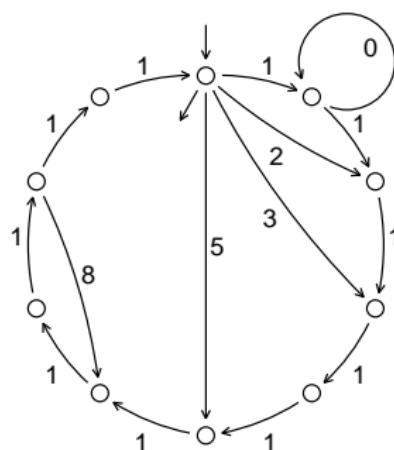
defines all words with an even number of 0 's.

Take union of that with 9 similar expressions (for the other digits) and you have a r.e. for L . A DFA would need 2^{10} states to keep track of whether each digit has appeared an odd or an even number of times so far.

A language that is easier to describe using a DFA

Alphabet $A = \{0, 1, 2, \dots, 9\}$

$L \subset A^*$ is defined to be numbers whose digits add up to a multiple of 10. e.g. 1234, 28, 2828.



(not all transitions are shown)

observations, exercise

Recall: any formal language is a well-defined set of words over some given alphabet.

A regular language is one that is given by some regular expression or accepted by some DFA (or NFA)

A regular language will have many different but equivalent DFAs or reg exprs that represent it.

Which of the following regular expressions over alphabet $\{a, b\}$ are equivalent:

$b(a^*)b, \quad bb \cup ba(a^*)b,$

$ba(a^*), \quad \{b, \epsilon\}a(a^*), \quad \{ba, a\}(a^*)$

Equivalences amongst regular expressions

Let R , S and T denote regular expressions. We can note some general rules governing equivalence of regular expressions such as

$$R \cup S = S \cup R$$

$$R \cup (S \cup T) = (R \cup S) \cup T$$

$$R(ST) = (RS)T$$

$$R(S \cup T) = (RS) \cup RT$$

Equivalences amongst regular expressions

Some properties of closure

$$(R^*)^* = R^*$$

$$R(R^*) = (R^*)R$$

$$R(R^*) \cup \epsilon = R^*$$

$$R(SR)^* = (RS)^*R$$

$$(R \cup S)^* = (R^* \cup S^*)^* = (R^*S^*)^* = R^*(S(R^*))^*$$

Example proof

Prove that $(R^*)^* = R^*$.

Suppose $w \in (R^*)^*$.

Then $w = w_1 w_2 \dots w_n$ for $w_i \in R^*$.

$w_i \in R^*$ means that $w_i = w_{i,1} w_{i,2} \dots w_{i,n(i)}$ for $w_{i,j} \in R$.

Hence w is a concatenation of words that belong to R , so $w \in R^*$.

Then, prove that if $w \in R^*$ then $w \in (R^*)^*$.

If $w \in R^*$ then it follows immediately that $w \in (R^*)^*$, since the closure of a language contains all words in that language and possibly more.

Alphabet A ,

$A^* = \{\text{all strings consisting symbols from } A\}$

the subsets of A^* are called languages

Question:

For a language L , does exist an algorithm to check for any x , if x is in L ?

For example:

$A = \{0, 1\}$

A^* is the set of all 1, 1 strings including empty string ϵ

Some languages over A :

$L1 = \{\}$

$L2 = \{\epsilon\}$

$L3 = \{0, 1\}$

$L4 = \{w: \text{ends with } 1\}$

$L5 = \{011, 0011, 000111, 00001111, \dots\}$

DFA $M = (Q, A, \delta, i, T)$ NFA

Where

$Q:$...

$A:$...

$\delta: Q \times A \rightarrow Q$

$\delta: Q \times A \rightarrow 2^Q$

$i:$...

$T:$...

Extension:

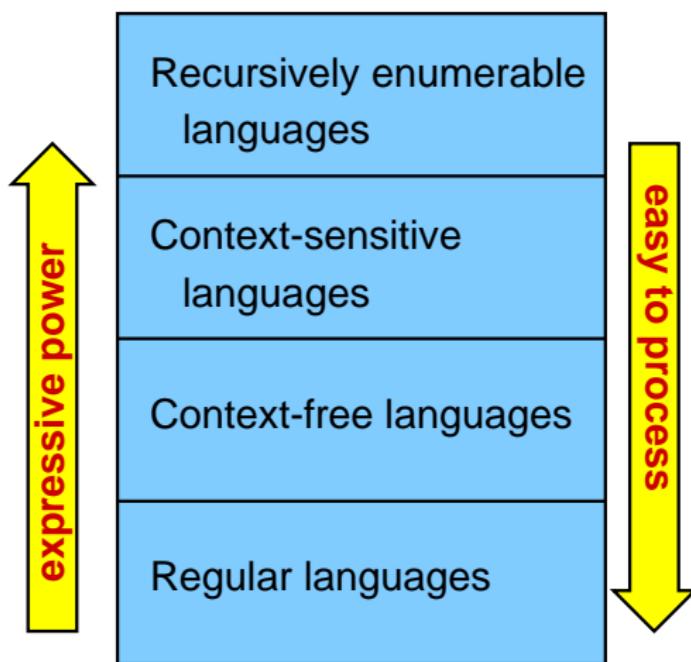
$\delta: Q \times A^* \rightarrow Q$

DFA \Leftrightarrow NFA (subset construction)

Regular Expression (recursively defined via Union, Concatenation, Closure)

DFA (NFA) \Leftrightarrow Regular Expression

Chomsky hierarchy



Two characterisations of regular languages

A “regular language” means a language that can be defined using a regular expression.

Kleene's theorem

Regular languages are those languages that can be accepted by finite automata.

We have already seen that any NFA has an equivalent DFA. So when we talk about languages that can be accepted by finite automata, we don't need to specify whether we are talking about DFAs or NFAs.

We prove Kleene's theorem by showing how to convert from a NFA to a reg expr, and vice versa.

Convert regular expression to NFA

recall: we have seen to convert NFA to DFA!

Recall how r.e's are defined. To show that any r.e. has an equivalent NFA:

- ① Construct a NFA that accepts any single one-letter word (easy!)
- ② Given two NFAs, construct a new one that accepts the concatenation of their languages
- ③ Given two NFAs, construct a new one that accepts the union of their languages
- ④ Given any single NFA, construct a new one that accepts the closure of its language

Claim 1

Let L_1 and L_2 be languages over an alphabet A . If there are finite automata accepting L_1 and L_2 then there is a finite automaton accepting $L_1 L_2$.

Proof. Assume L_1 and L_2 are accepted by DFAs

$$\mathcal{A}_1 = (Q_1, A, \phi_1, i_1, T_1) \text{ and } \mathcal{A}_2 = (Q_2, A, \phi_2, i_2, T_2).$$

Define \mathcal{A} to be the automaton $(Q_1 \cup Q_2, A, \psi, i_1, T_2)$ where ψ is defined for $q_1 \in Q_1$ by

$$\psi(q_1, a) = \{\phi_1(q_1, a)\} \text{ if } \phi_1(q_1, a) \notin T_1$$

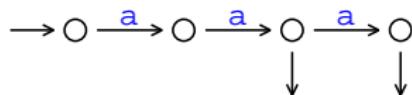
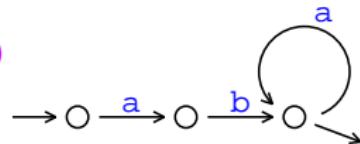
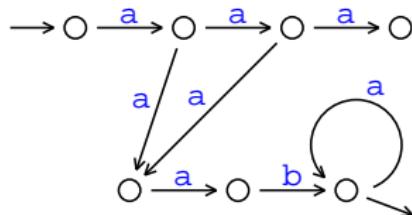
$$\psi(q_1, a) = \{\phi_1(q_1, a), i_2\} \text{ if } \phi_1(q_1, a) \in T_1.$$

and for $q_2 \in Q_2$ by

$$\psi(q_2, a) = \{\phi_2(q_2, a)\}.$$

Example

{aa,aaa}

ab(a^*){aa,aaa}ab(a^*)

Next: verify that $L_1 L_2$ is the language accepted by \mathcal{A} .

First, prove that $L_1 L_2 \subseteq L(\mathcal{A})$.

Suppose $w_1 \in L_1$ and $w_2 \in L_2$. Then $\phi_1(i_1, w_1) \in T_1$ and $\phi_2(i_2, w_2) \in T_2$. By the definition of ψ we see that $i_2 \in \psi(i_1, w_1)$ and $\phi_2(i_2, w_2) \in \psi(i_1, w_1 w_2)$. Now in \mathcal{A} the path with label $w_1 w_2$ has the form

$$\begin{aligned} i_1 &\longrightarrow \cdots \xrightarrow{w_1} \{\dots, i_2, \dots\} \xrightarrow{w_2} \cdots \longrightarrow \\ &\qquad\qquad\qquad \{\dots, \phi_2(i_2, w_2), \dots\} \end{aligned}$$

so the final set $\psi(i_1, w_1 w_2)$ contains a state from T_2 , i.e. an accepting state of \mathcal{A} . Hence $w_1 w_2$ is accepted by \mathcal{A} . This shows that $L_1 L_2 \subseteq L(\mathcal{A})$.

Then, prove that $L(\mathcal{A}) \subseteq L_1 L_2$.

Proof sketch (see textbook for full proof):

Suppose that $w \in L(\mathcal{A})$.

There is an accepting path in \mathcal{A} labelled by the letters of w . From each set of states on the path, choose one that is “the right choice” for the non-deterministic machine \mathcal{A} .

At some point the path passes from states in Q_1 to states in Q_2 , and never returns to Q_1 .

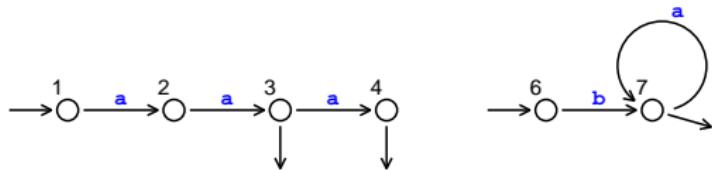
This point gives the right point to split w into two words w_1 and w_2 accepted by \mathcal{A}_1 and \mathcal{A}_2 respectively.

Claim 2

Let L_1 and L_2 be languages over an alphabet A . If there are finite automata accepting L_1 and L_2 then there is a finite automaton accepting $L_1 \cup L_2$.

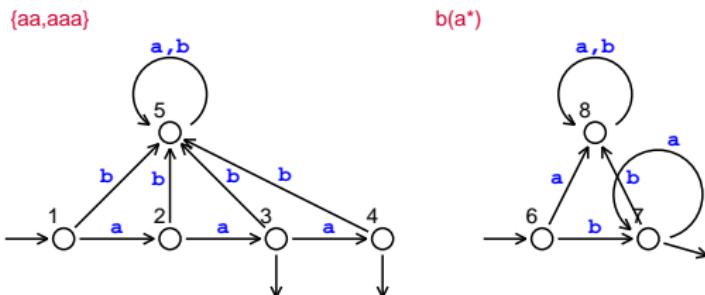
{aa,aaa}

b(a*)

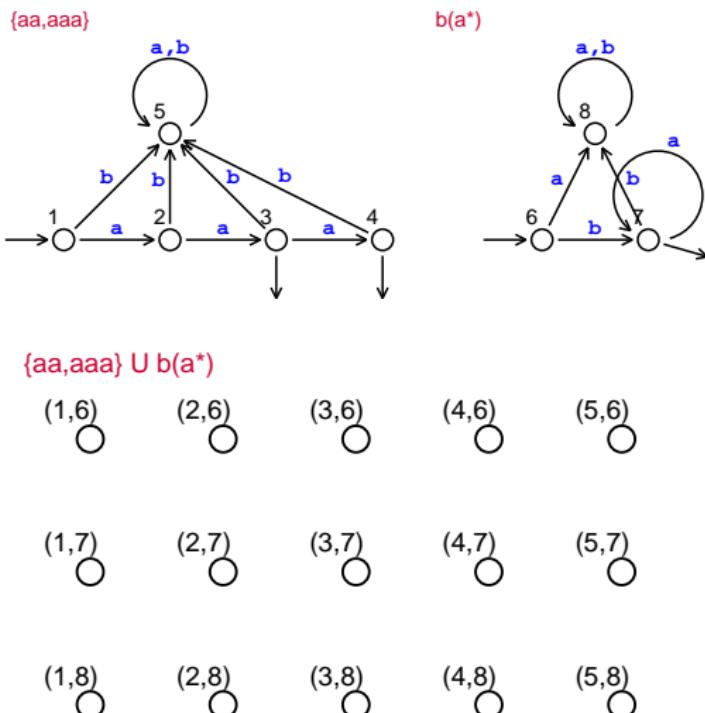


We will show how to do it on the above example, then give a general description.

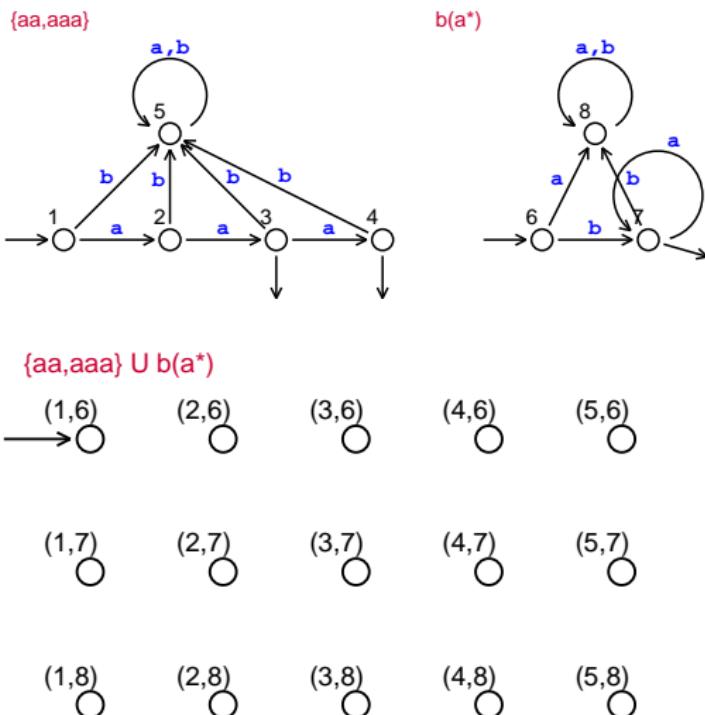
Example



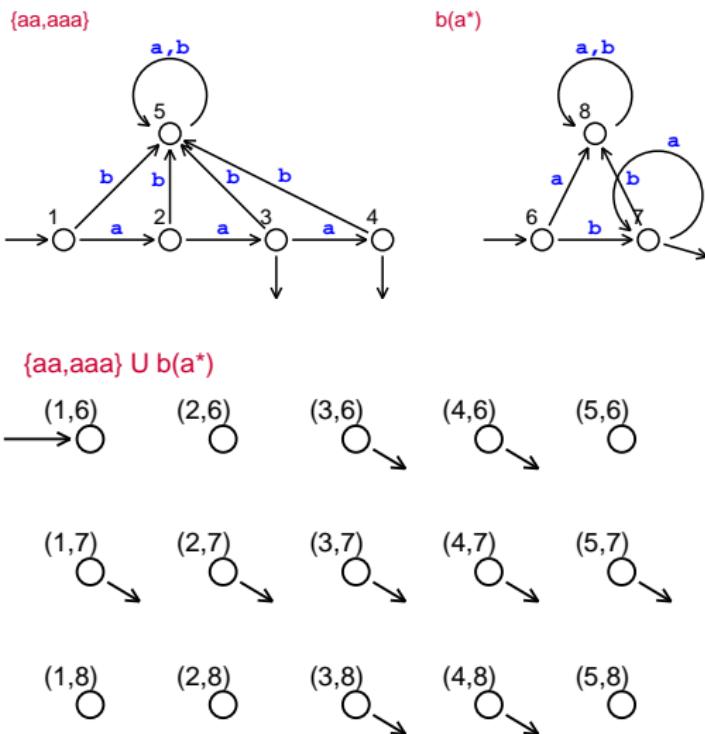
Example



Example

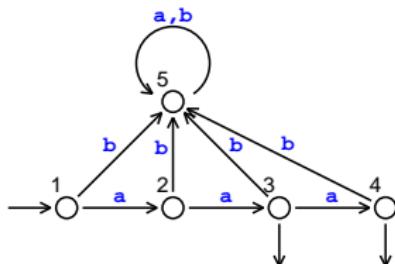


Example

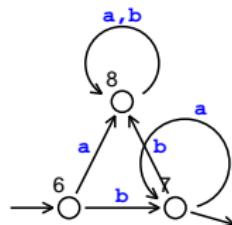


Example

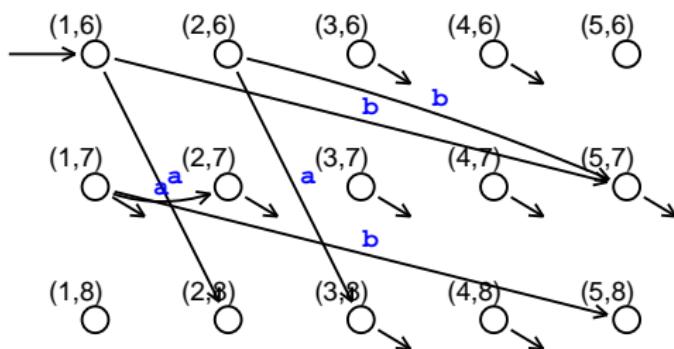
$\{aa, aaa\}$



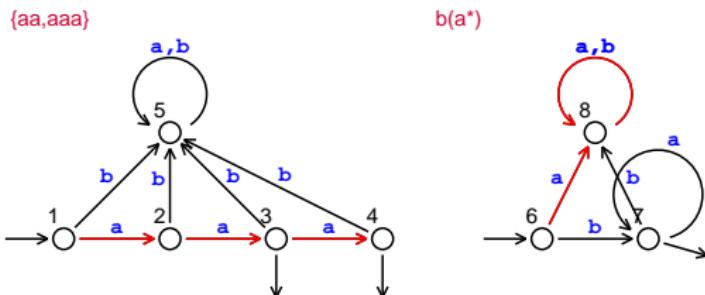
$b(a^*)$



$\{aa,aaa\} \cup b(a^*)$

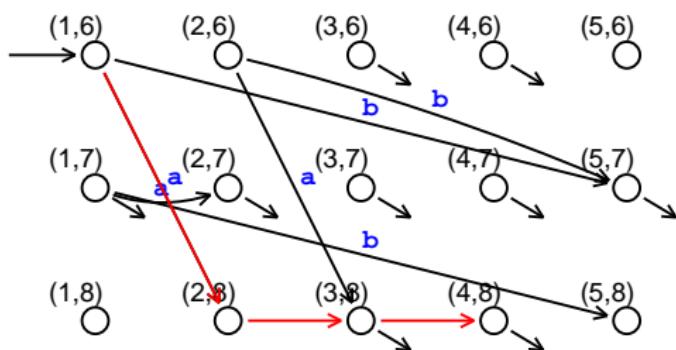


Example



input string aaa

$\{aa,aaa\} \cup b(a^*)$



Claim 2

Let L_1 and L_2 be languages over an alphabet A . If there are finite automata accepting L_1 and L_2 then there is a finite automaton accepting $L_1 \cup L_2$.

Proof. Let $\mathcal{A}_1 = (Q_1, A, \phi_1, i_1, T_1)$ be an automaton accepting L_1 and $\mathcal{A}_2 = (Q_2, A, \phi_2, i_2, T_2)$ be an automaton accepting L_2 , where we may assume both are deterministic. Define $\mathcal{A} = (Q, A, \phi, i, T)$ as follows:

$$\begin{aligned} Q &= Q_1 \times Q_2 \\ \phi((q_1, q_2), a) &= (\phi_1(q_1, a), \phi_2(q_2, a)) \\ &\quad \text{for all } q_1 \in Q_1; q_2 \in Q_2; a \in A \\ i &= (i_1, i_2). \\ T &= \{(p, q) : p \in T_1 \text{ or } q \in T_2\} \end{aligned}$$

Then

$$\phi(i, w) = (\phi_1(i_1, w), \phi_2(i_2, w)) \text{ for all } w \in A^*.$$

Now we have

$$\begin{aligned} w \in L_1 \cup L_2 &\iff \phi_1(i_1, w) \in T_1 \text{ or } \phi_2(i_2, w) \in T_2 \\ &\iff \phi(i, w) = (p, q) \\ &\quad \text{where } p \in T_1 \text{ or } q \in T_2 \text{ (or both)} \end{aligned}$$

So from the definition of the set T of terminal states of \mathcal{A} we have

$$w \in L_1 \cup L_2 \Leftrightarrow \phi(i, w) \in T \Leftrightarrow w \text{ is accepted by } \mathcal{A}.$$

Hence the language accepted by \mathcal{A} is $L_1 \cup L_2$.

Observation

The 2 constructions so far give a general way of constructing a finite automaton that accepts any *finite* language.

A word in a finite language is built from a sequence of concatenations

The language is built from a sequence of unions of sets of words

Claim 3

If language L is accepted by some finite automaton, then so is language L^* .

Make new initial state i' , with $i' \in T$.

Let ϕ' be the new transition function, where ϕ' contains all transitions of ϕ , and in addition:

If $\phi(i, a) = q$ for $a \in A$, $q \in Q$ then $\phi'(i', a) = q$.

If $\phi(q, a) = t \in T$ for $q \in Q$, then let $\phi'(q, a) = \{i', t\}$. (making the automaton non-deterministic)

Comment

“union” construction could be modified in obvious (?) way to get an “intersection” result; could also do symmetric difference, or complicated boolean combinations of more than 2 languages.

(eg: given 4 languages L_1, L_2, L_3, L_4 , define a new language L_{new} as $w \in L_{new}$ iff $w \in$ exactly 2 of L_1, L_2, L_3, L_4 .)

But it's not so obvious how to do that with regular expressions!

Next: conversion from any finite automaton to an equivalent r.e. (which completes Kleene's theorem)

Alphabet A ,

$A^* = \{\text{all strings consisting symbols from } A\}$

the subsets of A^* are called languages

Question:

For a language L , does exist an algorithm to check for any x , if x is in L ?

For example:

$A = \{0, 1\}$

A^* is the set of all 0, 1 strings including empty string ϵ

Some languages over A :

$L1 = \{\}$

$L2 = \{\epsilon\}$

$L3 = \{0, 1\}$

$L4 = \{w: \text{ends with } 1\}$

$L5 = \{011, 0011, 000111, 00001111, \dots\}$

DFA $M = (Q, A, \delta, i, T)$ NFA

Where

$Q:$...

$A:$...

$\delta: Q \times A \rightarrow Q$

$\delta: Q \times A \rightarrow 2^Q$

$i:$...

$T:$...

Extension:

$\delta: Q \times A^* \rightarrow Q$

DFA \Leftrightarrow NFA (subset construction)

Regular Expression (recursively defined
via Union, Concatenation, Closure)

Kleene's Theorem:

DFA (NFA) \Leftrightarrow Regular Expression

Regular Expression \Rightarrow NFA (done!)

NFA \Rightarrow Regular Expression (today's topic)

Algorithm

Given a DFA $M = (Q, A, \phi, i, T)$.

Construct a regular expression for $L(M)$

Recursive: express solution in terms of solutions to simpler instances of this problem.

Base case: M has no labelled transitions to an accepting state.

If $i \in T$ then $L(M) = \{\epsilon\}$

Else $L(M) = \emptyset$

The algorithm (continued)

Recursive case: there's a transition to accepting state

If $T = \{q_1, \dots, q_k\}$ where $k > 1$

Then $L(M) = \cup_{j=1}^k L(M_j)$

where $M_j = (Q, A, \phi, i, \{q_j\})$

Evaluations of regular expressions for $L(M_j)$ are done recursively.

The algorithm (continued)

Else /* one accepting state */

Let $T = \{t\}$ (i.e. $M = (Q, A, \phi, i, \{t\})$)

If $t = i$

Then $L(M) = L(M')^*$

where $M' = (Q \cup \{t'\}, A, \phi', i, \{t'\})$

and $\phi'(q, a) = t'$ whenever $\phi(q, a) = i$

otherwise $\phi'(q, a) = \phi(q, a)$.

Regular expression for $L(M')$ is found recursively.

The algorithm (continued)

Else /* $T = \{t\}$, $t \neq i$ */

If M has labelled transitions to i

Then $L(M) = L(M')^* L(M'')$

where $M' = (Q, A, \phi, i, \{i\})$,

M'' is like M but without the labelled transitions to i

Else /* no labelled transitions to i */

$L(M) = \cup_{a \in A} aL(M_a)$

where M_a is like M but without transition

from i labelled by a (don't include M_a)

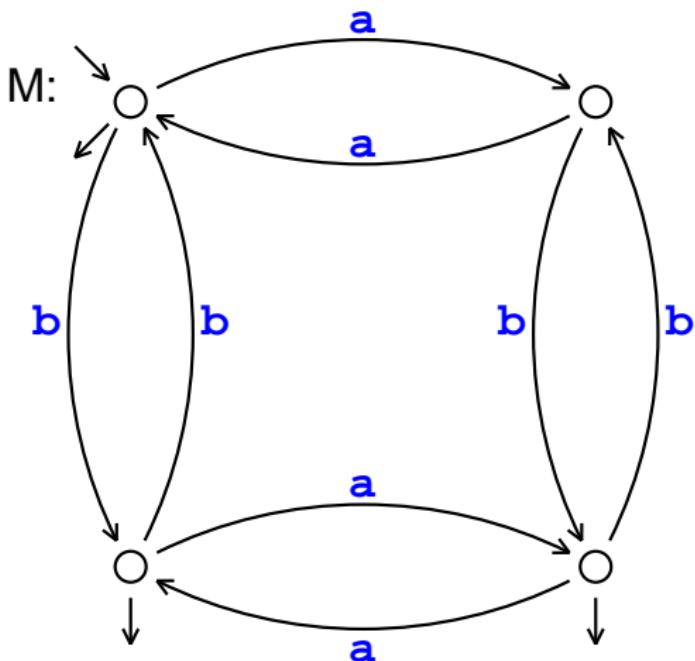
if no such transition is in M)

Initial state of M_a is $\phi(i, a)$.

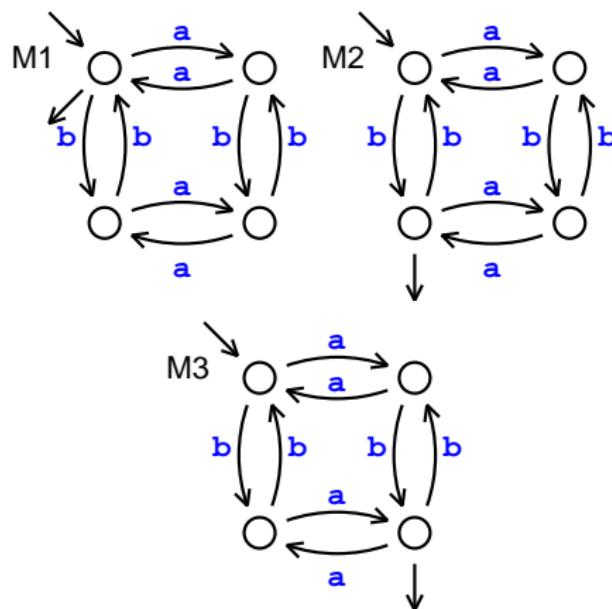
Regular expressions for $L(M')$, $L(M'')$, $L(M_a)$ etc are found recursively.

Convert this DFA M to equivalent r.e.

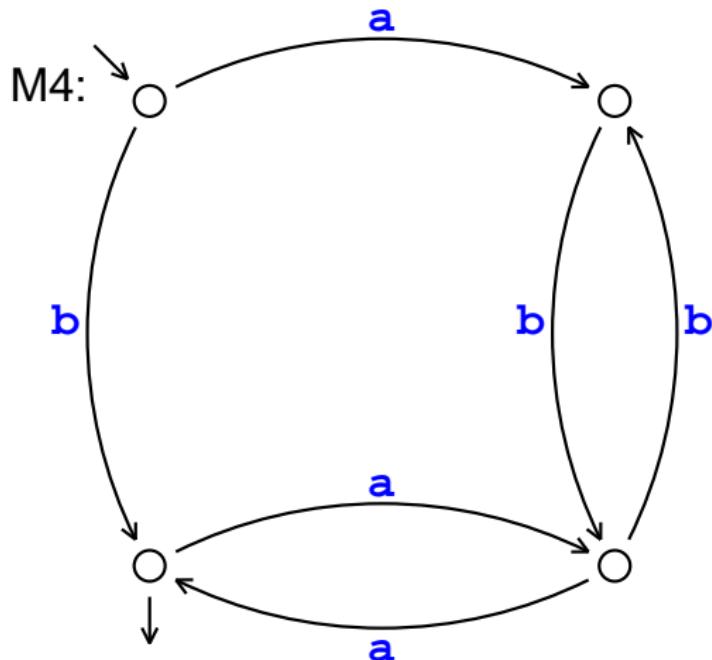
Language: even number of a 's or an odd number of b 's.



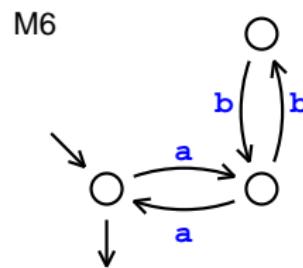
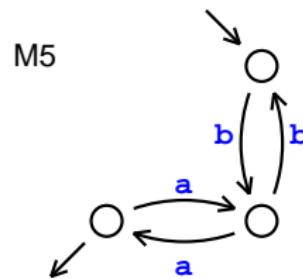
$$L(M) = L(M1) \cup L(M2) \cup L(M3)$$



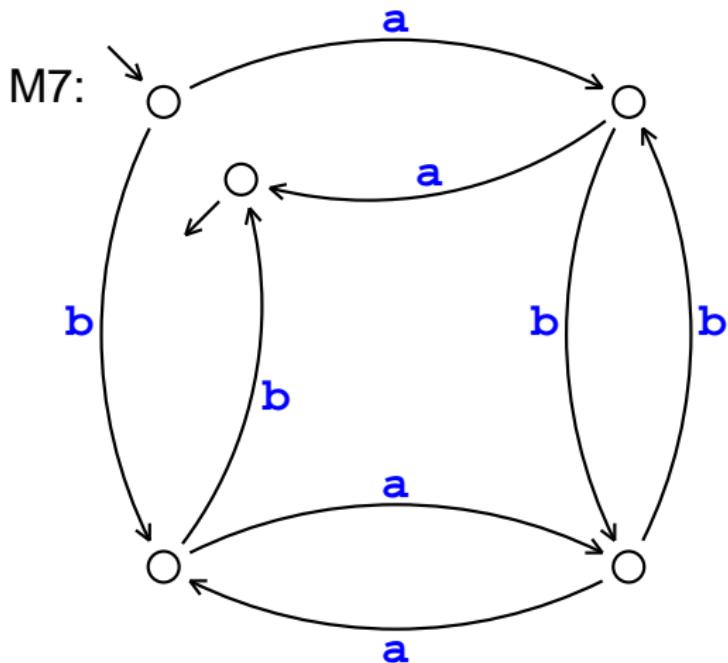
$$L(M2) = L(M1)*L(M4)$$



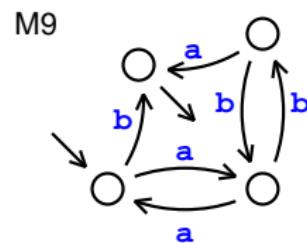
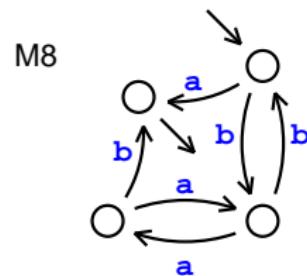
$$L(M4) = \textcolor{blue}{a}L(M5) \cup \textcolor{blue}{b}L(M6)$$



$$L(M1) = L(M7)^*$$



$$L(M7) = \textcolor{blue}{a}L(M8) \cup \textcolor{blue}{b}L(M9)$$



Justifying the DFA to r.e. algorithm

The algorithm expresses the language accepted by a given DFA M in terms of languages accepted by other DFAs - recursive

Want to verify that:

- ① $L(M)$ is the same as the language accepted by the combinations of DFAs constructed in the recursive calls
- ② The algorithm terminates.

Regarding (1): We saw that in all the cases that arose the languages are the same

Regarding (2): Define “simpler” such that the algorithm decomposes a task into “simpler” tasks, until we reach base case (the “simplest” tasks where it’s obvious what to do)

Proving that the algorithm terminates

Say what we mean by “simpler”

Verify that the algorithm always expresses a language in terms of languages accepted by “simpler” machines.

Prove that any sequence of machines M_1, M_2, M_3, \dots where M_i is simpler than M_{i-1} , is finite.

That will prove that the depth of recursion is finite.

Comparing finite automata M_1 and M_2

Define the number of transitions of M to be the number of pairs (q, a) for which $\phi(q, a)$ is defined.

- ① If M_1 and M_2 have differing numbers of transitions, the one with the smaller number is simpler.
- ② Failing that, the one with the smaller number of accepting states is simpler.
- ③ If one (but not both) of M_1, M_2 has no transitions to initial state, that one is simpler.
- ④ If one (but not both) of M_1, M_2 has initial state accepting, the other one is simpler.

Check that in each case given in the algorithm, the machines constructed for the recursive calls are simpler.

remarks

Priority of rules is top to bottom, ie apply rule 1, if that can't be used to compare DFAs, then apply rule 2, etc.

Not all pairs of distinct machines can be compared this way, but argue that M_1 and M_2 — where M_2 is generated by a recursive call from applying algo to M_1 — can be compared (and M_2 is simpler). That's good enough for us.

structure of this comparison method mimics structure of algorithm, but points out what we look for to be convinced M_2 is simpler.

Finally, prove that our definition of “simpler” does not allow infinite sequences of simpler and simpler machines.

Prove that any sequence of machines M_1, M_2, M_3, \dots where M_i is simpler than M_{i-1} , is finite.

Proof. Let N be the number of transitions in M_1 .

Going from M_i to M_{i+1} , rule 1 can only be used $\leq N$ times.

Let M_k be the machine we reach after the final usage of rule 1 in the sequence.

Let N' be the number of states of M_k .

Now rule 2 can only be used $\leq N'$ times. In between each usage of rule 2, rules 3 and 4 can only be used once.

Conclude the sequence is indeed finite.

Kleene's Theorem (overview)

- ① General method for converting from reg expr to DFA
- ② General method for converting from DFA to reg expr (note: current version is different from these slides)

Method 1 also needs general method for converting NFA to DFA (given earlier). (Due to the fact that the constructions for closure and concatenation of languages accepted by DFAs may create a NFA.)

Note that an expression/finite automaton may (typically) be much larger than smallest possible.

Comments

“short cuts” are possible if you don’t follow the general method. (The point of the general method is that it always works)

e.g. if a DFA has a “bottleneck” (a transition that accepting paths must use) that may suggest its language is the concatenation of 2 languages accepted by machines each side of that transition. (And, maybe the general algorithm would start by expressing the DFA as the union of several DFAs that are only slightly smaller...)

Summary

Regular expressions are useful for describing tokens of programming languages.

Convert reg expr to DFA; DFA is easy to implement

We have noted: sometimes the DFA obtained is much larger than necessary

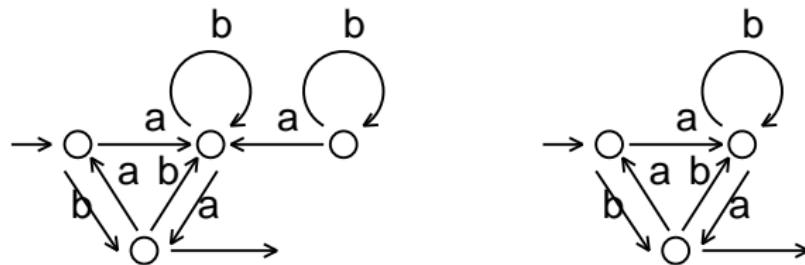
(Sometimes a reg expr necessarily has a much larger equiv DFA, often it doesn't need to be much larger)

Given a DFA, want to simplify it.

It turns out that we can (efficiently) find a simplest possible DFA equivalent to a given DFA.

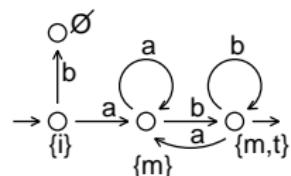
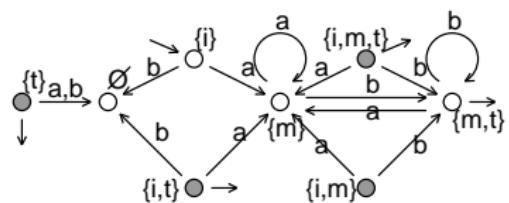
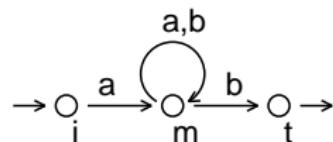
Identifying a DFA as being too complicated

“over-complicated” means more states than you need to accept the language. **Example:**



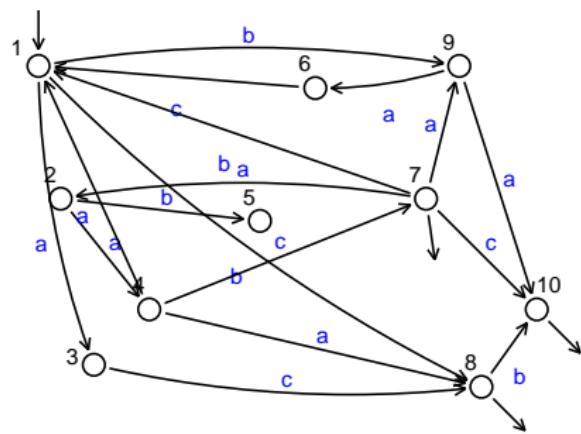
DFA on left has an inaccessible state that can be removed to get equiv DFA on right.

NFA to DFA conversion



Example

Certainly removal of inaccessible states is necessary for simplification.



not so obvious which states are inaccessible. (try to find them!)

Need a general method for removing inaccessible states.

A set of states is inaccessible if

- the initial state is not included
- there are no arcs coming into the set from outside it.

We don't want to test all sets. Build up collection of *accessible* states as follows.

$$S = \{i\}$$

repeat

$$S = S \cup \{\text{all states that are reached by an arc coming from a member of } S\}$$

until no extra states were added in last iteration

Claim: All accessible states are found by this procedure.

All states that are found in this way are certainly accessible.

we also need: every accessible state is found. To prove this, prove that:

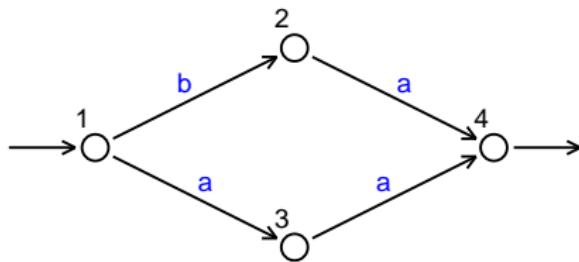
The new states found in the j -th iteration are those states reached for the first time with j -letter words. (and any accessible state is reached via some finite word)

If a state needs (say) a 10-letter word to reach it (from state i), it must be reached via a state that needed a 9-letter word to be reached.

So, if the 10-th iteration finds a new state, then the 9-th will have found a new state, and so on.

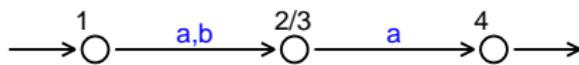
Merging Indistinguishable States

We are not done when we get rid of inaccessible states. There may be pairs of states that are “indistinguishable”.



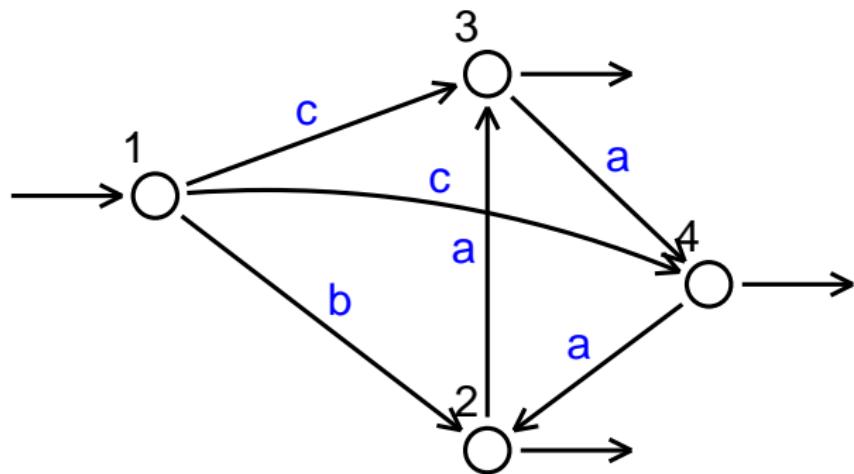
The above machine accepts $\{ba, aa\}$.

Claim: states 2 and 3 are really “the same” and can be merged:

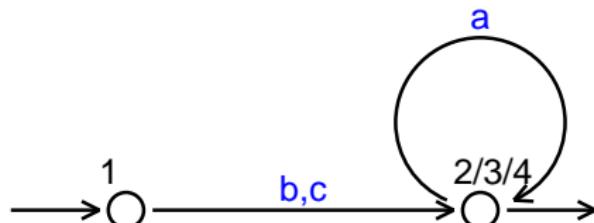


This smaller machine accepts the same language.

Another example:

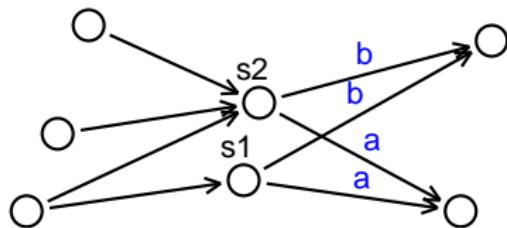


Three indistinguishable states can be replaced by one state:

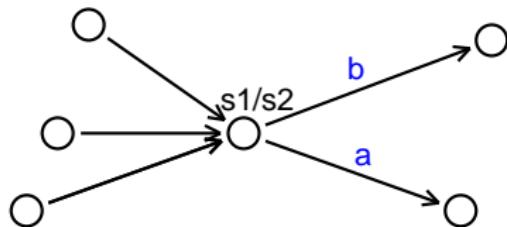


Two states are indistinguishable if the set of words labelling accepting paths from one state is the same as the set of words labelling accepting paths from the other state.

When we have two indistinguishable states, you can remove one and direct its arcs to/from the other. The resulting machine accepts the same language.



If s_1 and s_2 are indistinguishable, replace with:



Before we show how to check for indistinguishable states, we give the following result (in the handout):

Theorem: *If a DFA has no inaccessible states, and no indistinguishable states, then it has a minimal number of states.*

So there's nothing left to do when you have dealt with inaccessible and indistinguishable states.

So we will continue by *first* proving the theorem *then* showing the procedure for merging indistinguishable states, and give an algorithm that uses both of these procedures to minimise automata.

Proof: Suppose M has n states that are accessible and distinguishable.

Suppose M' is equivalent to M but has only $n - 1$ states.

We can find n words that reach each of the n states of M

use accessibility here

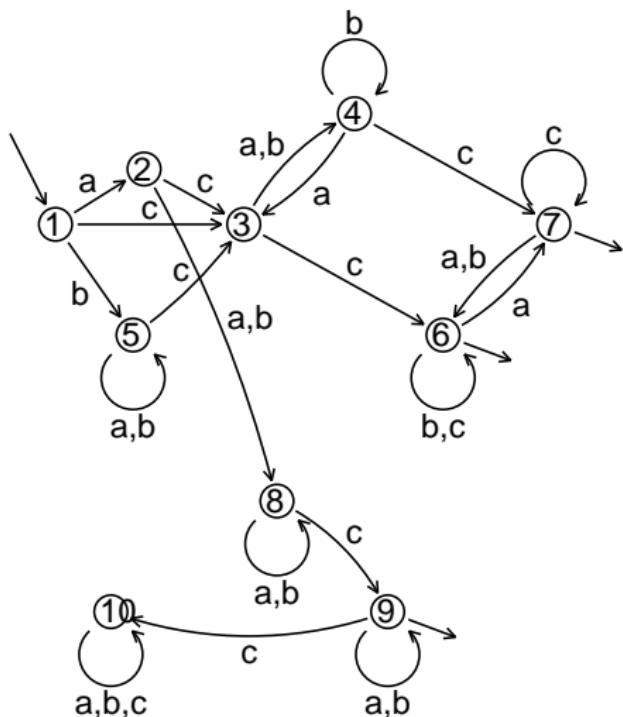
Two of these words must reach the same state of M' .

We can find a suffix for these two words such that M would accept one but not the other

here we use distinguishability

But both words, with any suffix attached, must reach the same state in M' . So M' cannot accept the same set of words. □

Example: Which States are Equivalent?



“Heuristic” analysis

Let L_i denote language accepted if you start at state i . We can see:

$$L_{10} = \emptyset$$

$$L_9 = \{a, b\}^*$$

(“no c ’s”)

$$L_8 = \{a, b\}^* c \{a, b\}^*$$

(strings containing exactly one c)

$$L_6, L_7 = \{a, b, c\}^*$$

(any string)

$$L_3, L_4 = \{a, b\}^* c \{a, b, c\}^*$$

(strings containing at least one c)

so far we have found 2 equivalences...

$$L_2 = cL_3 \cup \{a, b\}L_8$$

$$L_5 = \{a, b\}^*cL_3$$

$$L_1 = aL_2 \cup bL_5 \cup cL_3$$

Are any of the above equivalent?...

General method to determine whether pairs of states are distinguishable

Start with two equivalence classes,

T and $Q \setminus T$

Repeatedly subdivide as follows:

For each equivalence class, separate any pair of states for which some letter of the alphabet labels transitions to distinct equivalence classes.

We may stop when this process gives no more subdivisions.

(Intuitively, if an iteration of the above doesn't subdivide an equivalence class, then on further iterations you will be doing the same thing.)

Equivalence Classes (1)

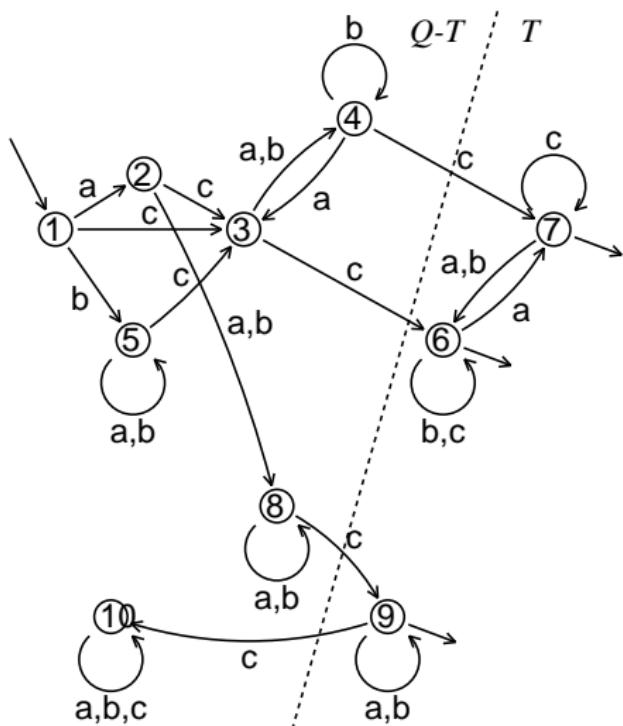
(To begin with, 2 states are equivalent if the empty string cannot “tell them apart”)

$$Q \setminus T = \{1, 2, 3, 4, 5, 8, 10\} \quad T = \{6, 7, 9\}$$

Look at states $s \in Q \setminus T$. Qn: Can any string of length 1 tell them apart?

s	$\phi(s, a)$	$\phi(s, b)$	$\phi(s, c)$
1	$Q \setminus T$	$Q \setminus T$	$Q \setminus T$
2	$Q \setminus T$	$Q \setminus T$	$Q \setminus T$
3	$Q \setminus T$	$Q \setminus T$	T
4	$Q \setminus T$	$Q \setminus T$	T
5	$Q \setminus T$	$Q \setminus T$	$Q \setminus T$
8	$Q \setminus T$	$Q \setminus T$	T
10	$Q \setminus T$	$Q \setminus T$	$Q \setminus T$

Round 1

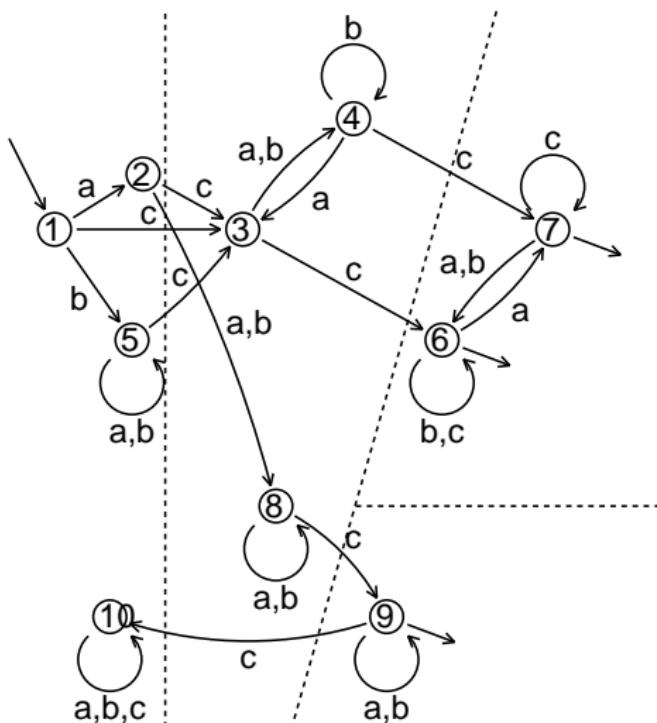


Now look at members of T .

s	$\phi(s, a)$	$\phi(s, b)$	$\phi(s, c)$
6	T	T	T
7	T	T	T
9	T	T	$\{1, 2, 5, 10\}$

$\{9\}$ is distinguished from $\{6, 7\}$.

Round 2



Equivalence Classes (2)

$\{9\}, \{6, 7\}, \{3, 4, 8\}, \{1, 2, 5, 10\}$

Try all transitions for members of each class:

s	$\phi(s, a)$	$\phi(s, b)$	$\phi(s, c)$
6	$\{6, 7\}$	$\{6, 7\}$	$\{6, 7\}$
7	$\{6, 7\}$	$\{6, 7\}$	$\{6, 7\}$

– we find nothing new.

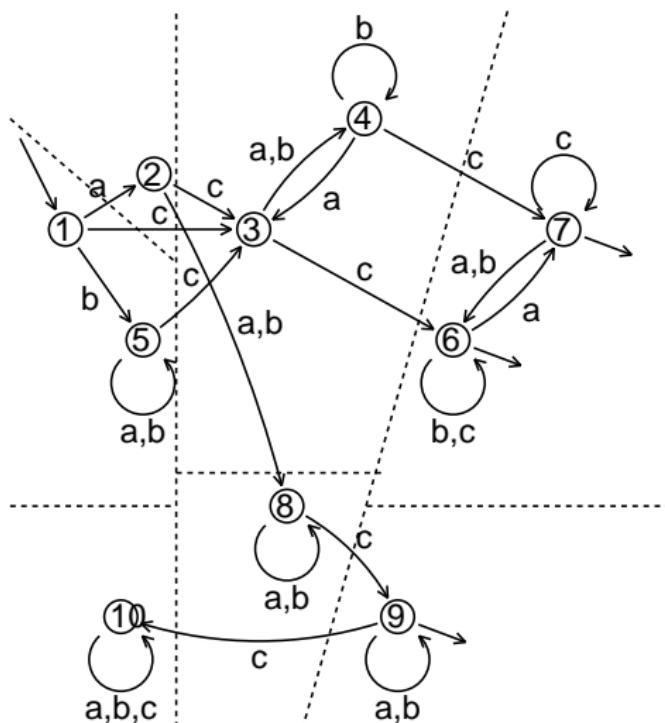
s	$\phi(s, a)$	$\phi(s, b)$	$\phi(s, c)$
3	$\{3, 4, 8\}$	$\{3, 4, 8\}$	$\{6, 7\}$
4	$\{3, 4, 8\}$	$\{3, 4, 8\}$	$\{6, 7\}$
8	$\{3, 4, 8\}$	$\{3, 4, 8\}$	$\{9\}$

$\{3, 4, 8\}$ splits into $\{3, 4\}, \{8\}$.

s	$\phi(s, a)$	$\phi(s, b)$	$\phi(s, c)$
1	{1,2,5,10}	{1,2,5,10}	{3,4,8}
2	{3,4,8}	{3,4,8}	{3,4,8}
5	{1,2,5,10}	{1,2,5,10}	{3,4,8}
10	{1,2,5,10}	{1,2,5,10}	{1,2,5,10}

Class splits into {1,5}, {2}, {10} (distinguishable with strings of length 2)

Round 3



Equivalence Classes (3)

$\{1, 5\}, \{2\}, \{3, 4\}, \{6, 7\}, \{8\}, \{10\}, \{9\}$

Same again:

s	$\phi(s, a)$	$\phi(s, b)$	$\phi(s, c)$
1	$\{2\}$	$\{1, 5\}$	$\{3, 4\}$
5	$\{1, 5\}$	$\{1, 5\}$	$\{3, 4\}$

s	$\phi(s, a)$	$\phi(s, b)$	$\phi(s, c)$
3	$\{3, 4\}$	$\{3, 4\}$	$\{6, 7\}$
4	$\{3, 4\}$	$\{3, 4\}$	$\{6, 7\}$

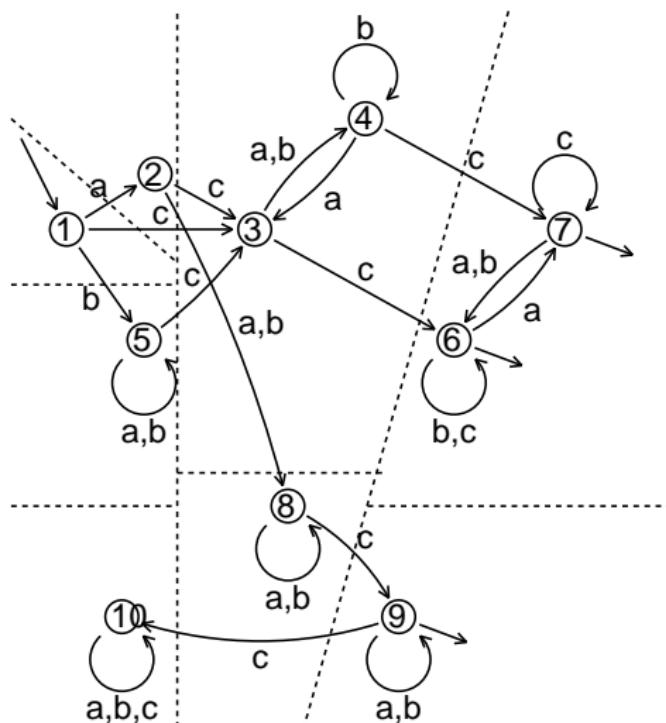
s	$\phi(s, a)$	$\phi(s, b)$	$\phi(s, c)$
6	$\{6, 7\}$	$\{6, 7\}$	$\{6, 7\}$
7	$\{6, 7\}$	$\{6, 7\}$	$\{6, 7\}$

and we now have:

$\{1\}, \{2\}, \{3, 4\}, \{5\}, \{6, 7\}, \{8\}, \{9\}, \{10\}$

Another iteration fails to distinguish 3 from 4 or 6 from 7.

Round 4



Proving that the algorithm for finding distinguishable states is correct:

Any pair of states in distinct equivalence classes are distinguishable.
(the algorithm essentially finds a distinguishing string.)

Any pair of states in the same class are indistinguishable. To prove this, prove that at the j -th iteration, we find pairs of distinguishable states that require a string of length j to distinguish them. If 2 states need a string of length j to distinguish them, there must be 2 states that need a string of length $j - 1$ to distinguish them. Hence the algorithm doesn't stop prematurely.

Alphabet A ,

$A^* = \{\text{all strings consisting symbols from } A\}$

The subsets of A^* are called languages

Question:

For a language L , does exist an algorithm to check for any x , if x is in L ?

We have identified a class of regular languages which can be described or defined by regular expression, DFA and NFA.

DFA \Leftrightarrow NFA (subset construction)

Kleene's Theorem:

DFA (NFA) \Leftrightarrow Regular Expression

Membership problem for regular language:

For a regular language L which is defined by a DFA M . To test if a word w is in L , input the w to M , **as for any input string, it always halts at a state**. If M halts at accepting state, w is in L , otherwise, w is not in L .

Question: Is there any languages which can not be defined by DFA (NFA, Regular expression)?

Answer: too many (infinite) (**today's topic**)

Exercise

Can either of the following languages (over alphabet $\{a, b\}$) be accepted by DFAs?

$a^n b^m$ for any $n, m \in \mathbf{N}$ (i.e. words such as ϵ , aaaabb, abbbb, bbb)

$a^n b^n$ for any $n \in \mathbf{N}$ (i.e. words such as ϵ , ab, aabb, aaabbb, ...)

We can argue that the second of these languages is not accepted by any DFA.

General idea: finite number of states means finite memory.

As an input string is scanned from left to right, DFA needs to count number of initial *a*'s occurring before first *b*.

But any DFA can only store a limited number of distinct numbers (at most the number of states, in fact). With no limit on length of possible input strings, any DFA will eventually be defeated.

(See handouts for a more rigorous proof by contradiction.)

The Pumping Lemma

The following result gives a property of languages that are recognised by deterministic finite automata. It can often be used to show that certain languages are not accepted by DFAs.

Let L be a language which is accepted by some deterministic finite automaton with N states. Then every word $z \in L$ with $|z| \geq N$ can be written in the form $z = uvw$ for some $u, v, w \in A^$ with*

1. $v \neq \epsilon$
2. $|uv| \leq N$
3. $uv^n w \in L$ for $n = 1, 2, 3, \dots$

Proof of Pumping Lemma (sketch)

Suppose a DFA has N states. If the DFA scans a word with more than N letters, it must enter some state on two (or more) distinct letters.

Consider the section of that word which lies between the two places where the same state is visited. What happens if instead of one copy of that sub-word, we inserted multiple copies?

At the end of each copy, the DFA returns to the same state.

Afterwards, things continue as if only one copy had been scanned. So if the original word was accepted, so also would be words obtained by inserting multiple copies of that section.

Notation: accepting/rejecting paths

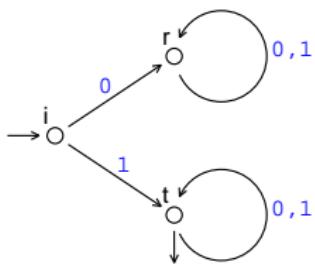
If $w = a_1 a_2 \dots a_n \in A^*$ and $\phi(p, w) = q$ then there are states r_1, r_2, \dots, r_{n-1} with

$$\begin{aligned}r_1 &= \phi(p, a_1), \quad r_2 = \phi(r_1, a_2), \quad r_3 = \phi(r_2, a_3), \quad \dots, \\q &= \phi(r_{n-1}, a_n).\end{aligned}$$

We say that the states p and q are connected by a path with label w and write

$$\begin{aligned}p \xrightarrow{a_1} r_1 \xrightarrow{a_2} r_2 \xrightarrow{a_3} r_3 \cdots \cdots r_{n-2} \xrightarrow{a_{n-1}} r_{n-1} \xrightarrow{a_n} q \\ \text{or } p \xrightarrow{w} q.\end{aligned}$$

In particular a word w is accepted by an automaton if and only if there is a path from the initial state to a terminal state with label w (which we call an *accepting path*)



If for example the word 0011 is input to this automaton then we obtain the following path:

$$i \xrightarrow{0} r \xrightarrow{0} r \xrightarrow{1} r \xrightarrow{1} r$$

so this word is rejected. The word 100 produces the following path

$$i \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{0} t$$

and so is accepted.

Proof of Pumping Lemma

If a word z belongs to a language accepted by an automaton with N states and $|z| \geq N$ then the accepting path for z must contain a repeated state and therefore will have the form

$$i \xrightarrow{u} \dots \xrightarrow{v} q \xrightarrow{v} \dots \xrightarrow{w} t$$

where i is the initial state, t is an accepting state and $z = uvw$, where $v \neq \epsilon$.

Now we can repeat the path labelled with v any number n of times and obtain an accepting path with label $uv^n w$. Hence $uv^n w$ is also accepted by the automaton. Also if q is taken to be the first repeated state then the path from i to the second q can contain at most $N + 1$ states, so its label uv has length at most N .

The word $uv^n w$ can be thought of as having been obtained by “pumping” the substring v of $z = uvw$.

Problem. Use pumping lemma to show that no DFA can accept palindromes over an alphabet with ≥ 2 letters.

Solution. Suppose for a contradiction that there is an N -state automaton that accepts palindromes. Let a, b be distinct letters in A . The word $a^{N+1}ba^{N+1}$ is a palindrome and has length $\geq N$ so by the pumping lemma we can write

$$a^{N+1}ba^{N+1} = uvw \text{ where } v \neq \epsilon \text{ and } |uv| \leq N$$

and $uv^n w$ is a palindrome for every $n \geq 1$. But now

$$a^{N+1}ba^{N+1} = uvw \text{ where } uv \text{ is shorter than } a^{N+1}.$$

Comparing the letters in these two words we see that uv must be a string of a 's and hence v must be a string of at least one a .

It now follows that $uv^2 w = a^{N+r}ba^{N+1}$ for some $r \geq 2$. By the pumping lemma this word must belong to the language P of all palindromes, but is clearly not a palindrome: contradiction!

Summary

We have seen various languages which are recognised by DFAs, and some that are not.

Pumping Lemma gives a way of proving that some languages do not have accepting DFAs. (example: strings of 1's whose length is a square number, ie 1, 1111, 1111111111, 1111111111111111, etc.)

It doesn't work for all languages not accepted by DFAs.

Example

"Strings of 1 's whose length is a square number" is not a regular language.

remarks: this language is a bit artificial, but it's certainly a formal language. Would like to know what sort of mechanism can recognise/describe it.

If the language is regular, then for some large enough m , 1^m is in the language, and some substring of length r can be "pumped"...

That is, 1^{m+r} , 1^{m+2r} , 1^{m+3r} etc are members of the language. This is impossible since the gap between 2 consecutive square numbers increases as the numbers increase, and so this sequence can't just contain square numbers. Conclude the language is not regular.

Another example

Use pumping lemma to prove that the set of words over $\{a, b\}$ having the same number of a 's as b 's is not regular.

If the language is regular, there exists some m such that all words longer than m have a sub-word within the first m symbols that can be "pumped"...

Consider the word $a^m b^m$, which is a member of the language. If we choose any subword to be pumped from amongst the first m symbols, we get a string of a 's. If we insert another copy of that subword, the new word is not a member of language. □

Example of language not accepted by a DFA for which you cannot directly use pumping lemma to prove that it is not accepted by a DFA:

The set of words over the $\{0, 1\}$ alphabet which *either* start with one 0 then have a square number of 1 's *or* start with at least two 0 's.

ie. $01, 01111, 0111111111, 0111111111111111$, etc. together with words such as $00, 0001, 001101$, etc.

Every word in this language has the prefix 0 which will generate further words in the language if it is “pumped”

Exercises

Given a language L that uses alphabet A , the complement of L is defined to be all words over A that are not members of L . The reverse of a language is the set of words which, if their letters were reversed, would give words in L .

- ① Prove that if L is a regular language then the complement of L is also regular.
- ② Prove that if L is a regular language then so is the reverse of L .

(then, use that second fact to prove that the language of the previous slide is irregular!)

Closure Properties of Regular languages

Closure properties express the idea that when one (several) languages are regular, then certain language obtained by some operation is also regular. Then we say, regular languages are closed under the operation.

Here is the summary of the principle closure properties for regular languages:

1. The union of two regular languages is regular.
2. The intersection of two regular languages is regular.
3. The complement of two regular languages is regular.
4. The difference of two regular languages is regular.
5. The reversal of a regular languages is regular.
6. The closure (star) of a regular language is regular.
7. The concatenation of two regular languages is regular.

Apply Closure Properties to Prove a language is not regular

To prove a language L , is not regular, we can find a regular language L' and operate it using (Union, difference, etc...) with L , if the resulted language is not regular, then L is not regular.

Decision Properties of Regular Languages

1. Is the language described (by DFA, NFA, R.E) empty?
2. Is a particular string w in the described (by DFA, NFA, R.E) language?
3. Do two descriptions of a language actually describe the same language? This question is often called "equivalence" of languages.

Alphabet A ,

$A^* = \{\text{all strings consisting symbols from } A\}$

the subsets of A^* are called languages

Question:

For a language L , does exist an algorithm to check for any x , if x is in L ?

For example:

$A = \{0, 1\}$

A^* is the set of all 0, 1 strings including empty string ϵ

Some languages over A :

$L1 = \{\}$

$L2 = \{\epsilon\}$

$L3 = \{0, 1\}$

$L4 = \{w: \text{ends with } 1\}$

$L5 = \{011, 0011, 000111, 00001111, \dots\}$

DFA $M = (Q, A, \delta, i, T)$ NFA

Where

$Q:$...

$A:$...

$\delta: Q \times A \rightarrow Q$

$\delta: Q \times A \rightarrow 2^Q$

$i:$...

$T:$...

Extension:

$\delta: Q \times A^* \rightarrow Q$

DFA \Leftrightarrow NFA (subset construction)

Regular Expression (recursively defined
via Union, Concatenation, Closure)

Kleene's Theorem:

DFA (NFA) \Leftrightarrow Regular Expression

Regular Expression \Rightarrow NFA (done!)

NFA \Rightarrow Regular Expression (today's topic)

Context-free Grammars

A grammar is a mechanism for representing a formal language. There are various kinds of grammar...

A Regular grammar is something that can represent a regular language, ie they are equivalent to regular expressions (and hence, finite automata).

A **context-free grammar** (CFG) is a very important concept; languages representable using CFGs are called context-free languages (CFLs). All regular languages are CFLs, and so also are various other languages such as palindromes.

Just like regular languages are also representable by finite automata, CFLs are representable by so-called pushdown automata.

Grammars

Later on, we consider generalisations:

Context-sensitive grammar and Unrestricted grammar, mechanisms that can represent even more languages (not just CFLs), but at a price - it's much harder to tell whether a word belongs to the language being represented...

Regarding the terminology: grammars have rules that allow certain symbols to be replaced by others. "context-free": rules allow symbols to be replaced no matter where those symbols occur in a string. "context-sensitive": rules allow replacement to occur in certain "contexts" of surrounding symbols. Details to follow...

Context-free Grammars

Show how CFGs can be converted into “normal forms”, ie equivalent CFGs that have additional syntactic restrictions

Use normal form to show that “pushdown automata” are the class of machines that accept CFLs.

Parsing is the process of checking that a sequence of symbols is generated by a context-free grammar

Consider classes of parsing algorithms expressible as restricted classes of pushdown automata (note: general pushdown automata are impractical to implement, unlike finite automata)

Context-free Grammars

Formally, a CFG consists of

- alphabet A
- set V of variables ($V \cap A = \emptyset$)
- start symbol $S \in V$
- set P of rules (a.k.a. productions) which allow variables to be replaced with strings of letters in A or other variables

A grammar describes a language - essentially the set of all strings of letters in A that can be obtained by applying sequences of rules starting from start symbol S .

Example

palindromes over alphabet $\{a, b, c\}$.

$$V = \{S\}.$$

Rules:

$$S \rightarrow aS_a$$

$$S \rightarrow bS_b$$

$$S \rightarrow cS_c$$

$$S \rightarrow a$$

$$S \rightarrow b$$

$$S \rightarrow c$$

$$S \rightarrow \epsilon$$

shorthand (Backus-Naur Form):

$$S \rightarrow aS_a \mid bS_b \mid cS_c \mid a \mid b \mid c \mid \epsilon$$

Example of using the grammar

For an example palindrome, write down sequence of rules that generates it.

e.g. string aabcbaa

$$S \Rightarrow aS_a \Rightarrow aaS_{aa} \Rightarrow aabS_{baa} \Rightarrow aabcbaa$$

(To prove that palindromes are the strings generated: prove by induction that intermediate strings generated are palindromic)

Another Example

Strings over the alphabet $\{a, b\}$ consisting of a string of a 's followed by the same number of b 's...

i.e. $\{a^n b^n : n = 1, 2, 3, \dots\}$

Consider the grammar, using just one variable S :

$$S \longrightarrow aSb|\epsilon$$

Example of using the grammar

e.g. string aaaabbbb

$$\begin{aligned} S &\Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaaSbbbb \\ &\qquad\qquad\qquad \Rightarrow \text{aaaabbbb} \end{aligned}$$

So, we have found another non-regular language that can be expressed using a CFG!

Finite languages

Let us convince ourselves using an example, that any finite language is a CFL, for example

$\{\text{cat}, \text{dog}, \text{mouse}\}.$

Use $V = \{S\}$ as before, and the rules

$S \rightarrow \text{cat}$

$S \rightarrow \text{dog}$

$S \rightarrow \text{mouse}$

Notation

lower case letters used to denote elements of A (on my slides, usually in blue)

capital letters (or else words enclosed in \langle and \rangle) used to denote elements of V

greek letters $\alpha, \beta, \gamma, \dots$ used to denote strings of letters and variable symbols

Notation

Derivations: $\alpha \Rightarrow \beta$ means that one application of a production gets from α to β . $\alpha \Rightarrow^* \beta$ means that a sequence of applications of productions gets from α to β .

A rule in a grammar is written using a single arrow \rightarrow as opposed to a double arrow \Rightarrow

$L(G)$ denotes language generated by G

More examples

Alphabet $\{a, b, c\}$. Consider the following (regular) languages:

The set of all strings of a 's.

The set of strings containing exactly one c .

The set $a^*b^*c^*$.

More examples (continued)

Alphabet $\{a, b, c\}$. Consider the following (regular) languages:

The set of all strings of an odd number of a 's.

The set of strings containing exactly two c 's.

(harder) Strings in which no letter occurs twice consecutively.

Regular Languages are CFLs

Let $G = (V, A, P, S)$, $G' = (V', A, P', S')$. Assume $V \cap V' = \emptyset$.

$L(G) \cup L(G')$ is generated by the grammar

$(V \cup V', A, P \cup P' \cup \{S \rightarrow S'\}, S)$

$L(G)^*$ is generated by

$(V, A, P \cup \{S \rightarrow SS, S \rightarrow \epsilon\}, S)$

$L(G)L(G')$ is generated by

$(V \cup V', A, P \cup P' \cup \{S'' \rightarrow SS'\}, S'')$

(where S'' is a new variable)

Regular Grammars

All productions must be of the form

$$X \rightarrow aY \quad \text{or} \quad X \rightarrow a \quad \text{or} \quad X \rightarrow \epsilon$$

where X, Y may be any variables and a any constant.

Any regular grammar generates a regular language (and any regular language has a regular grammar)

palindrome grammar contains productions like

$$S \rightarrow aSa$$

- not of the regular grammar form.

An equivalent definition

Allow a regular grammar to contain productions of the form

$$X \rightarrow wY \quad \text{or} \quad X \rightarrow w$$

where w is any non-empty *word* in A^* .

(The above def is used in note 11 and Hopcroft and Ullman. The previous def is used in Tremblay and Sorenson.)

To see that the two definitions are equivalent, need to show that any grammar of the above form can be re-expressed in the previous (more restrictive) form.

Conversion to “restricted” form

Given a rule such as

$$X \rightarrow abaY$$

replace it with rules

$$\begin{array}{l} X \rightarrow aU_1 \\ U_1 \rightarrow bU_2 \\ U_2 \rightarrow aY \end{array}$$

All new variable should be distinct from each other and pre-existing variables.

General remark: first example of how two restrictions on form of a grammar have same effect on languages expressible. Equivalence also extends to “variable change” productions eg

$$X \rightarrow Y$$

Regular Grammar to NFA

- Each variable becomes a state
- Start symbol becomes initial state
- Rules become transitions,

$$X \longrightarrow aY$$

becomes

$$Y \in \phi(X, a).$$

i.e. add Y to the states accessible by an a -transition from state X .

- If there are rules of the form $X \longrightarrow \epsilon$ make X an accepting state.
- Given rules of the form $X \longrightarrow a$ add accepting state T , add transition

$$\phi(X, a) = T$$

How to prove the construction is valid

Suppose word w is generated by the grammar. Argue that w is accepted by the NFA.

Then show that any w accepted by the NFA could be produced by the grammar.

Key observation:

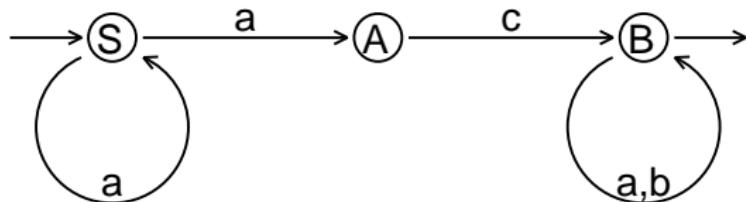
In any derivation using a regular grammar, all strings produced are a string of constants followed (perhaps) by a single variable.

Example

The language $aa^*c\{a,b\}^*$ has regular grammar:

$$\begin{array}{lcl} S & \longrightarrow & aS \\ S & \longrightarrow & aA \\ A & \longrightarrow & cB \\ B & \longrightarrow & aB \\ B & \longrightarrow & bB \\ B & \longrightarrow & \epsilon \end{array}$$

NFA:



Example of conversion from grammar to FA

Write down NFA which accepts the same language as

$$\begin{array}{l} S \rightarrow abS \mid bX \mid \epsilon \\ X \rightarrow aS \mid bX \mid b \end{array}$$

replace $S \rightarrow abS$ with

$$\begin{array}{l} S \rightarrow aY \\ Y \rightarrow bS \end{array}$$

Replace $X \rightarrow b$ with

$$\begin{array}{l} X \rightarrow bZ \\ Z \rightarrow \epsilon \end{array}$$

(eliminates rules with single letter on RHS, no need to add new accepting state)

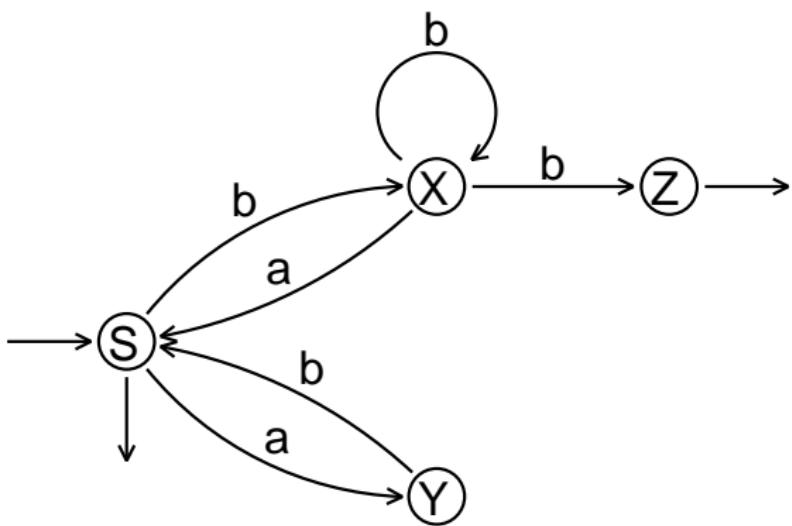
We have

$$\begin{array}{lcl} S & \longrightarrow & aY \mid bX \mid \epsilon \\ X & \longrightarrow & aS \mid bX \mid bZ \\ Y & \longrightarrow & bS \\ Z & \longrightarrow & \epsilon \end{array}$$

Corresponding transition function:

$$\begin{array}{ll} \phi(S, a) = \{Y\} & \phi(S, b) = \{X\} \\ \phi(X, a) = \{S\} & \phi(X, b) = \{X, Z\} \\ \phi(Y, a) = \emptyset & \phi(Y, b) = \{S\} \\ \phi(Z, a) = \emptyset & \phi(Z, b) = \emptyset \end{array}$$

S and Z are accepting states. S is initial state.



Regular expression: $\{ab, bb^*a\}^*\{\epsilon \cup bb^*b\}$

Another example

language L of expressions containing pairs of balanced parentheses

For example:

(() ()) ()

(((())))

The following words are not in L :

)) (

() (((

It's not a RL (why?)

Example (continued)

CFG:

start symbol E .

$$E \rightarrow EE$$

$$E \rightarrow (E)$$

$$E \rightarrow \epsilon$$

Example: some arithmetic expressions

arithmetic expressions using infix operators $+$, $*$, parentheses and “atoms” x and y .

start symbol E (no other variable symbols)

$$E \rightarrow E * E$$

$$E \rightarrow E + E$$

$$E \rightarrow (E)$$

$$E \rightarrow x$$

$$E \rightarrow y$$

To see that $x * (x + y)$ is in the language, find a derivation.

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E)$$

$$\Rightarrow x * (E + E) \Rightarrow x * (x + E) \Rightarrow x * (x + y)$$

notation: $E \xrightarrow{*} x * (x + y)$

Compiler

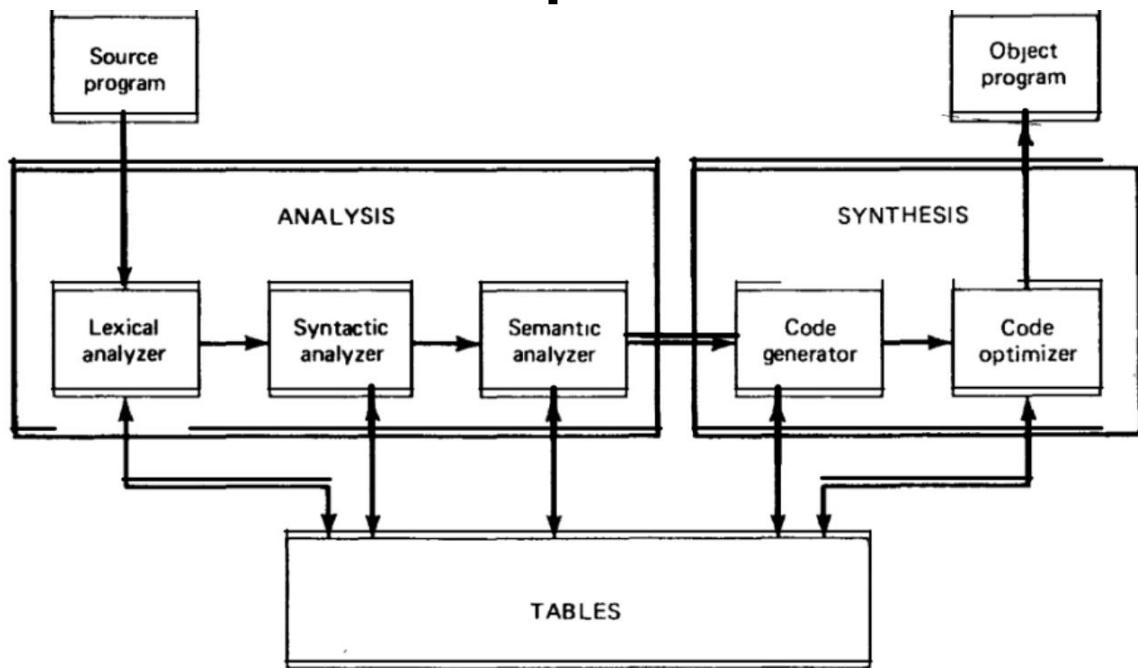


Figure 1-3 Components of a compiler.

Parsing

Parsing is the problem of: given a grammar and a string, find a derivation of the string using the grammar.

Given our claim that programming languages are often described using grammars, this is a key problem for compiler.

We prefer unambiguous CFGs (ones where all derivations of a string are "essentially the same", noting that variables of a CFG often correspond to specific structures of a program (eg, arithmetic expression, procedure, statement, method etc.

Arithmetic expressions

arithmetic expressions using infix operators $+$, $*$, parentheses and “atoms” x and y .

start symbol E (no other variable symbols)

$$E \rightarrow E * E$$

$$E \rightarrow E + E$$

$$E \rightarrow (E)$$

$$E \rightarrow x$$

$$E \rightarrow y$$

To see that $x * (x + y)$ is in the language, find a derivation.

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E)$$

$$\Rightarrow x * (E + E) \Rightarrow x * (x + E) \Rightarrow x * (x + y)$$

notation: $E \xrightarrow{*} x * (x + y)$

Ambiguity

$x * x + y$ has two alternative derivations:

$$E \Rightarrow E * E \Rightarrow E * E + E$$

$$\Rightarrow x * E + E \Rightarrow x * x + E \Rightarrow x * x + y$$

or alternatively

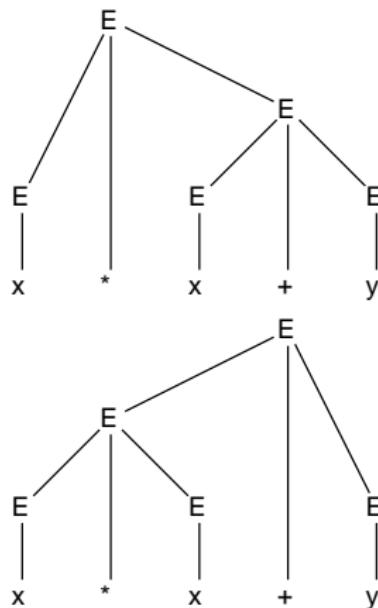
$$E \Rightarrow E + E \Rightarrow E * E + E \text{ (then as before)}$$

(actually you can also replace the final E's with x and y in a different order, but that's not important in a strong sense...)

The derivations above treat either $E + E$ or respectively $E * E$ as a subexpression.

Parse Trees (a.k.a. syntax trees, derivation trees)

Any derivation has a corresponding tree. But, a parse tree may have more than one corresponding derivation.



Convert an ambiguous grammar into an unambiguous one:

$$E \rightarrow F * E$$

$$E \rightarrow F + E$$

$$F \rightarrow (E)$$

$$E \rightarrow (E)$$

$$E \rightarrow x$$

$$E \rightarrow y$$

$$F \rightarrow x$$

$$F \rightarrow y$$

This grammar has the same language. But now I claim that any word in the language has only one parse tree.

There is no algorithm to decide whether a given CFG is ambiguous, but there are various sufficient conditions.

Explain why is the new grammar unambiguous but generates the same language:

Ambiguity arises from expressions having structure

$$E\langle op \rangle E\langle op \rangle E$$

where each $\langle op \rangle$ denotes either $+$ or $*$.

The modified grammar has 2 rules with $\langle op \rangle$ on RHS; the subexpression to left of $\langle op \rangle$ is initially an F – must be “self-contained”.

Given an expression generated by original grammar, we look for the shortest prefix which belongs to the language, and if it's not the whole word, make that prefix an F and the remainder is $\langle op \rangle E$.

Parse Trees

Recall the grammar we considered earlier

Grammar: $A = \{+, *, (,), x, y\}$, $V = \{E\}$, $S = E$, productions P :

$$E \rightarrow E * E$$

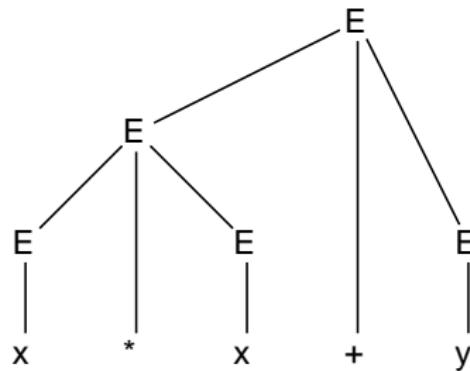
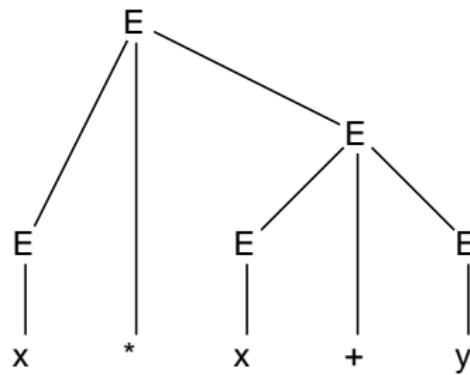
$$E \rightarrow E + E$$

$$E \rightarrow (E)$$

$$E \rightarrow x$$

$$E \rightarrow y$$

Recall the ambiguity:



Formal definitions

A derivation tree is a rooted tree with a node for each symbol introduced during the derivation.

The root is the start symbol.

A node corresponding to a constant is a leaf of the tree.

A node corresponding to a variable has as children the sequence of symbols on RHS of the rule used to replace it.

leftmost derivation: a derivation in which each step of the derivation consists of replacing the first, ie leftmost variable in a string with the RHS of some rule in which that variable is the LHS.

rightmost derivation: each derivation step replaces the rightmost variable in a string with the RHS of one of that variable's productions.

Given any derivation, its derivation tree can be used to construct a unique leftmost (or alternatively a unique rightmost) derivation.

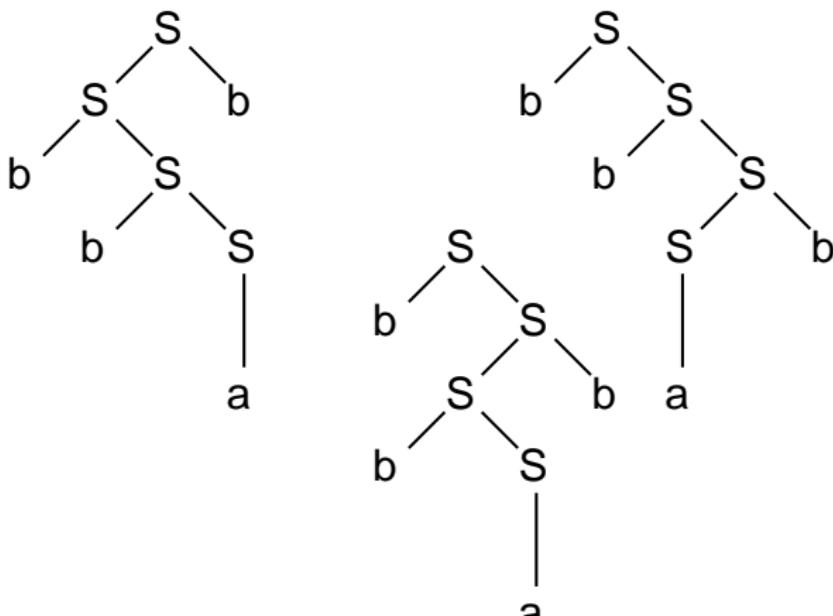
(and a derivation tree may typically have many other derivations (neither leftmost nor rightmost) associated with it.)

Recall that an ambiguous grammar was defined as one in which there is more than one derivation tree for some string. We see that this is equivalent to defining an ambiguous grammar to be one in which there is more than one leftmost (alternatively, rightmost) derivation for some string.

Example of simple ambiguous grammar

$$S \rightarrow bS \mid Sb \mid a$$

The word **bbab** has trees:



Example (continued)

The derivations corresponding to the parse trees (in order of appearance from left to right) are:

$$S \Rightarrow Sb \Rightarrow bSb \Rightarrow bbSb \Rightarrow bbab$$

$$S \Rightarrow bS \Rightarrow bSb \Rightarrow bbSb \Rightarrow bbab$$

$$S \Rightarrow bS \Rightarrow bbS \Rightarrow bbSb \Rightarrow bbab$$

Parsers

A *parser* for a grammar G takes as input a word $w \in L(G)$ and finds a derivation of w .

We continue by explaining why for any grammar you can find a parser, but naive method would create an inefficient and impractical one.

Subsequently we mention associated *parser generators* ie algorithms that create the parsers automatically from grammars.

“brute-force” parsing

Given word w and grammar $G = (V, A, P, S)$, find derivation of w .

w can be derived from S using say n steps.

$i = 0, \mathcal{S}_i = \{S\};$

repeat

$i = i + 1;$

\mathcal{S}_i is words derivable from elements of

\mathcal{S}_{i-1} in one step

(formally: $\mathcal{S}_i = \{\beta : \alpha \Rightarrow \beta; \alpha \in \mathcal{S}_{i-1}\}$)

until we find w in \mathcal{S}_i

exponential blowup in size of sets \mathcal{S}_i

Example of brute-force parsing

Look for a leftmost derivation.

Search the *leftmost graph* of the CFG, defined to be a rooted tree with

- ① start symbol at root
- ② each node has a string generated by the grammar, its children are strings derived from it by a single step of *leftmost* derivation.

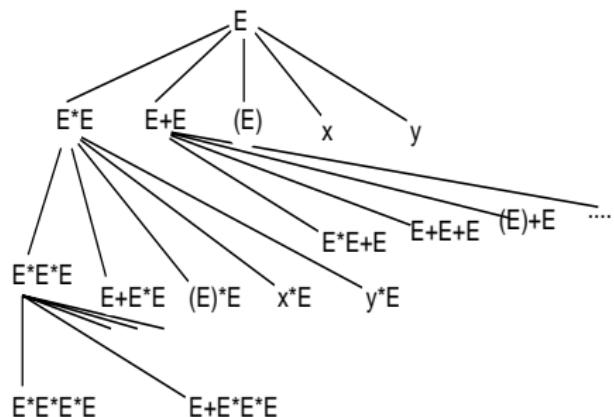
Example

$$S \longrightarrow XX$$

$$X \longrightarrow XXX \mid bX \mid Xb \mid a$$

leftmost graph of a CFG

$$E \longrightarrow E * E \mid E + E \mid (E) \mid x \mid y$$



It will take a long time to find $(((((y+y+y+y))))))$ using either BFS or DFS

Parsers, Parser generators

As noted, brute-force parsing of a string with respect to CFG G is (nearly always) impractical.

CFG can be converted to *pushdown automaton* (PDA); We would not want to implement an unrestricted PDA. A *deterministic* PDA (DPDA) would be OK. (we elaborate on this later)

A *deterministic context-free language* (DCFL) is a CFL that can be recognised by a DPDA.

Normal Forms

Chomsky normal form of a context-free grammar:

All productions must be of one of the forms:

$S \rightarrow \epsilon$ (S is start symbol)

$X \rightarrow a$

$X \rightarrow YZ$

" S is the only variable that can be replaced by empty string; RHS of all rules is of length at most 2; in particular a variable can be replaced by either a letter or 2 consecutive variables"

Any CFG is equivalent to some CFG in Chomsky normal form.

Chomsky normal form is used for a version of the pumping lemma for CFLs.

(compare with pumping lemma for regular sets)

Greibach normal form: all rules are of the form:

$$S \rightarrow \epsilon$$

$$X \rightarrow a$$

$$X \rightarrow aX_1X_2 \dots X_r$$

"You can replace a variable with a single constant followed by a possibly empty sequence of variables. As before start symbol only may be replaced by empty string."

Any CFG is equivalent to some CFG in Greibach normal form.

Greibach normal form is used in proof of equivalence of CFGs and pushdown automata (analogous to equivalence of regular grammars and finite automata)

Conversion to Chomsky normal form

First, get rid of empty productions.

(ie. rules that allow a variable other than the start symbol to be replaced with empty string.)

$$S \rightarrow X_a Y$$

$$X \rightarrow Y_c X$$

$$X \rightarrow b$$

$$X \rightarrow c$$

$$X \rightarrow \epsilon$$

$$Y \rightarrow \epsilon$$

Example:

Replace $X \rightarrow Y_c X$ with
 $X \rightarrow \epsilon$

$$S \rightarrow X_a Y$$

$$S \rightarrow a Y$$

$$X \rightarrow Y_c X$$

$$X \rightarrow Y_c$$

Conversion to Chomsky normal form

First, get rid of empty productions.

(ie. rules that allow a variable other than the start symbol to be replaced with empty string.)

$$S \rightarrow XaY$$

$$X \rightarrow YcX$$

$$X \rightarrow b$$

$$X \rightarrow c$$

$$X \rightarrow \epsilon$$

$$Y \rightarrow \epsilon$$

Example:

$$S \rightarrow XaY$$

$$S \rightarrow aY$$

Replace $X \rightarrow YcX$ with

$$X \rightarrow Yc$$

$$Y \rightarrow \epsilon$$

$$S \rightarrow XaY$$

$$S \rightarrow Xa$$

$$S \rightarrow aY$$

$$S \rightarrow a$$

$$X \rightarrow YcX$$

$$X \rightarrow cX$$

$$Y \rightarrow Y$$

General Rule

Any empty production $X \rightarrow \epsilon$ (assuming X is not the start symbol) can be re-expressed by making copies of rules where X appears on RHS, replacing that X in the RHS by ϵ .

Note: if you have e.g.

$$\begin{array}{l} S \rightarrow XaXbX \\ X \rightarrow \epsilon \end{array}$$

then you need rules for all patterns of deletion of the X 's, ie. replace with

$$\begin{array}{l} S \rightarrow XaXbX \\ S \rightarrow aXbX \\ S \rightarrow XabX \\ S \rightarrow XaXb \\ S \rightarrow Xab | aXb | abX | ab \end{array}$$

Next, get rid of variable changes.

$$S \rightarrow XaaX$$

$$S \rightarrow bWcY$$

$$X \rightarrow Y \quad -$$

$$Y \rightarrow Z \quad - \quad \text{variable changes}$$

Example: $X \rightarrow W \quad -$

$$X \rightarrow c$$

$$Y \rightarrow aa$$

$$Z \rightarrow a$$

$$W \rightarrow a$$

We note: X can be replaced by Y or Z or W , and Y can be replaced by Z .

replace	$S \rightarrow XaaX$ $X \rightarrow Y$ $X \rightarrow Z$ $X \rightarrow W$	with	$S \rightarrow XaaX$ $S \rightarrow YaaX$ $S \rightarrow YaaY$ $S \rightarrow XaaZ$ \vdots
---------	---	------	--

Generally, $S \rightarrow (W|X|Y|Z)aa(W|X|Y|Z)$

Replace $\begin{matrix} S \\ Y \end{matrix} \rightarrow \begin{matrix} bW_cY \\ Z \end{matrix}$ with $\begin{matrix} S \\ S \end{matrix} \rightarrow \begin{matrix} bW_cY \\ bW_cZ \end{matrix}$

General Rule

For each variable, identify which single variables may replace it using one or more variable change rules.

Suppose X can be replaced by X_1 or X_2 or $\dots X_r$.

For each (non-variable change) rule with X on RHS, add rules with all possible replacements of X 's on RHS by X_1, \dots, X_r .

Then (the variable change rules that allow X to be replaced by other variables become redundant) delete the variable change rules with X on LHS.

Do the above for all variables.

Next, get rid of rules where RHS contains a constant together with other symbols

$$S \rightarrow \epsilon$$

We have rules of the form $X \rightarrow a$ where S is the start symbol
 $X \rightarrow \alpha$

and α is a string of length ≥ 2 .

Given any rule of the form

$$X \rightarrow \beta a \gamma$$

(where $a \in A$), replace with

$$\begin{aligned} X &\rightarrow \beta U \gamma \\ U &\rightarrow a \end{aligned}$$

where U is a new variable that may occur only in these 2 rules.

Finally, replace all rules whose RHS is of length > 2.

By now all rules have the form

$$\begin{array}{lcl} S & \longrightarrow & \epsilon \\ X & \longrightarrow & a \\ X & \longrightarrow & X_1 X_2 X_3 \dots X_r \end{array}$$

where S is the start symbol, X and the X_i are variables.

For $r > 2$ replace $X \longrightarrow X_1 X_2 X_3 \dots X_r$ by

$$\begin{array}{lcl} X & \longrightarrow & X_1 X_2 X_3 \dots X_{r-2} U \\ U & \longrightarrow & X_{r-1} X_r \end{array}$$

where as before U is unique to the above 2 rules.

Obviously the first new rule above may need to be replaced using the same trick. Keep going until all RHS's have length ≤ 2 .

Nullable Variables

BASIS: If $A \rightarrow \epsilon$ is a production of G , then A is nullable.

INDUCTION: If there is a production $B \rightarrow C_1 C_2 \cdots C_k$, where each C_i is nullable, then B is nullable. Note that each C_i must be a variable to be nullable, so we only have to consider productions with all-variable bodies.

Now we give the construction of a grammar without ϵ -productions. Let $G = (V, T, P, S)$ be a CFG. Determine all the nullable symbols of G . We construct a new grammar $G_1 = (V, T, P_1, S)$, whose set of productions P_1 is determined as follows.

For each production $A \rightarrow X_1 X_2 \cdots X_k$ of P , where $k \geq 1$, suppose that m of the k X_i 's are nullable symbols. The new grammar G_1 will have 2^m versions of this production, where the nullable X_i 's, in all possible combinations are present or absent. There is one exception: if $m = k$, i.e., all symbols are nullable, then we do not include the case where all X_i 's are absent. Also, note that if a production of the form $A \rightarrow \epsilon$ is in P , we do not place this production in P_1 .

Unit Pair

BASIS: (A, A) is a unit pair for any variable A . That is, $A \xrightarrow{*} A$ by zero steps.

INDUCTION: Suppose we have determined that (A, B) is a unit pair, and $B \rightarrow C$ is a production, where C is a variable. Then (A, C) is a unit pair.

To eliminate unit productions, we proceed as follows. Given a CFG $G = (V, T, P, S)$, construct CFG $G_1 = (V, T, P_1, S)$:

1. Find all the unit pairs of G .
2. For each unit pair (A, B) , add to P_1 all the productions $A \rightarrow \alpha$, where $B \rightarrow \alpha$ is a nonunit production in P . Note that $A = B$ is possible; in that way, P_1 contains all the nonunit productions in P .

Useless Symbols: Non-Generating , Unreachable

BASIS: Every symbol of T is obviously generating; it generates itself.

INDUCTION: Suppose there is a production $A \rightarrow \alpha$, and every symbol of α is already known to be generating. Then A is generating. Note that this rule includes the case where $\alpha = \epsilon$; all variables that have ϵ as a production body are surely generating.

BASIS: S is surely reachable.

INDUCTION: Suppose we have discovered that some variable A is reachable. Then for all productions with A in the head, all the symbols of the bodies of those productions are also reachable.

1. First eliminate nongenerating symbols and all productions involving one or more of those symbols. Let $G_2 = (V_2, T_2, P_2, S)$ be this new grammar. Note that S must be generating, since we assume $L(G)$ has at least one string, so S has not been eliminated.
2. Second, eliminate all symbols that are not reachable in the grammar G_2 .

We can now summarize the various simplifications described so far. We want to convert any CFG G into an equivalent CFG that has no useless symbols, ϵ -productions, or unit productions. Some care must be taken in the order of application of the constructions. A safe order is:

1. Eliminate ϵ -productions.
2. Eliminate unit productions.
3. Eliminate useless symbols.

Theorem 7.14: If G is a CFG generating a language that contains at least one string other than ϵ , then there is another CFG G_1 such that $L(G_1) = L(G) - \{\epsilon\}$, and G_1 has no ϵ -productions, unit productions, or useless symbols.

To put a grammar in CNF, start with one that satisfies the restrictions of Theorem 7.14; that is, the grammar has no ϵ -productions, unit productions, or useless symbols. Every production of such a grammar is either of the form $A \rightarrow a$, which is already in a form allowed by CNF, or it has a body of length 2 or more. Our tasks are to:

- a) Arrange that all bodies of length 2 or more consist only of variables.
- b) Break bodies of length 3 or more into a cascade of productions, each with a body consisting of two variables.

The construction for (a) is as follows. For every terminal a that appears in a body of length 2 or more, create a new variable, say A . This variable has only one production, $A \rightarrow a$. Now, we use A in place of a everywhere a appears in a body of length 2 or more. At this point, every production has a body that is either a single terminal or at least two variables and no terminals.

For step (b), we must break those productions $A \rightarrow B_1B_2 \cdots B_k$, for $k \geq 3$, into a group of productions with two variables in each body. We introduce $k - 2$ new variables, C_1, C_2, \dots, C_{k-2} . The original production is replaced by the $k - 1$ productions

$$A \rightarrow B_1C_1, \quad C_1 \rightarrow B_2C_2, \dots, C_{k-3} \rightarrow B_{k-2}C_{k-2}, \quad C_{k-2} \rightarrow B_{k-1}B_k$$

Consider the following context free grammar

$$\begin{array}{l} S \rightarrow aAa \mid bBb \\ A \rightarrow C \mid a \\ B \rightarrow b \\ C \rightarrow CDE \mid \epsilon \\ D \rightarrow A \mid ab \end{array}$$

- a) Find all nullable variables and eliminate ϵ -productions.

Nullable variables: C, A, D

Elimination of ϵ -productions:

$$\begin{array}{l} S \rightarrow aAa \mid aa \mid bBb \\ A \rightarrow C \mid a \\ B \rightarrow b \\ C \rightarrow CDE \mid CE \mid DE \mid E \\ D \rightarrow A \mid ab \end{array}$$

- b) Find all unit pairs and eliminate all unit productions in the resulting grammar of a).

(A, A) A -> a
(A, C) A -> CDE | CE | DE
(A, E)
(B, B) B -> b
(C, C) C -> CDE | CE | DE
(C, E)
(D, D) D -> ab
(D, A) D -> a
(D, C) D -> CDE | CE | DE
(D, E)

The resulting grammar:

$$\begin{array}{ll} S & \rightarrow aAa \mid aa \mid bBb \\ A & \rightarrow CDE \mid CE \mid DE \mid a \\ B & \rightarrow b \\ C & \rightarrow CDE \mid CE \mid DE \mid E \\ D & \rightarrow CDE \mid CE \mid DE \mid a \mid ab \end{array}$$

c) Find all useless symbols and eliminate them in the resulting grammar of b).

Generating: S, A, B, D. Remove all productions containing non-generating symbols C, E:

$$\begin{array}{l} S \rightarrow aAa \mid aa \mid bBb \\ A \rightarrow a \\ B \rightarrow b \\ D \rightarrow a \mid ab \end{array}$$

Reachable: S, A, B, E. Remove all productions containing unreachable symbol D, we have

$$\begin{array}{l} S \rightarrow aAa \mid aa \mid bBb \\ A \rightarrow a \\ B \rightarrow b \end{array}$$

d) Put the resulting grammar in c) into Chomsky normal form.

$$\begin{array}{l} S \rightarrow AA1 \mid AA \mid BB1 \\ A1 \rightarrow AA \\ B1 \rightarrow BB \\ A \rightarrow a \\ B \rightarrow b \end{array}$$

The Pumping Lemma for CFLs

Given a CFL, any sufficiently long string in that CFL has *two* substrings (at least one of which is non-empty) such that if both of these substrings are “pumped” you generate further words in that CFL.

More formally...

Given a CFL L , there exists a number N such that any string $\alpha \in L$ with $|\alpha| \geq N$ can be written as

$$\alpha = sxtyu$$

such that

$$sx^2ty^2u, sx^3ty^3u, sx^4ty^4u, \dots$$

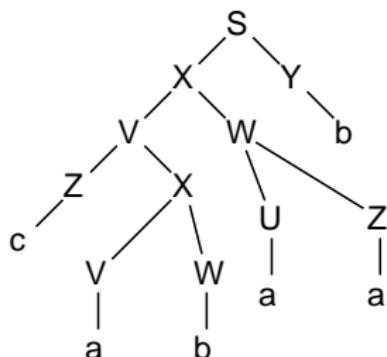
are all members of L , $|xty| \leq N$ and $|xy| > 0$ (which we need for all strings in this collection to be distinct).

How do we find suitable substrings x and y ?

Consider the following (Chomsky normal form) grammar

$$\begin{array}{lcl} S & \longrightarrow & XY \\ U & \longrightarrow & a \\ V & \longrightarrow & ZX \mid a \mid b \\ X & \longrightarrow & VW \mid a \\ Y & \longrightarrow & b \mid c \\ Z & \longrightarrow & a \mid c \\ W & \longrightarrow & UZ \mid b \end{array}$$

The string `cabaab` belongs to the language. Also it contains “suitable substrings” x and y which we can find by looking at a derivation tree.



We find two X 's on the same path. We can say:

$$X \Rightarrow^* cXaa$$

(via the sequence $X \Rightarrow VW \Rightarrow ZXW$
 $\Rightarrow cXW \Rightarrow cXUZ \Rightarrow cXaZ \Rightarrow cXaa$)

Generally: $X \Rightarrow^* ccXaaaa \Rightarrow^* cccXaaaaaaaa\dots$

The substrings c and aa can be pumped, because a derivation of $cabaab$ can go as follows:

$$\begin{aligned} S &\xrightarrow{*} Xb \\ &\xrightarrow{*} c\underline{X}aab \\ &\xrightarrow{*} cabaab \end{aligned}$$

but the underlined X could have been used to generate extra c 's and aa 's on each side of it.

Given a grammar, it's not hard to see that any sufficiently long string will have a derivation tree in which a path down to a leaf must contain a repeated variable.

If the grammar is in Chomsky normal form and has v variables, any string of length $> 2^v$ will necessarily have such a derivation tree.

Example

Prove the following is not a CFL:

$$\{ \textcolor{blue}{1}, \textcolor{blue}{101}, \textcolor{blue}{101001}, \textcolor{blue}{1010010001}, \\ \textcolor{blue}{101001000100001}, \dots \}$$

Proof by contradiction: suppose string s (in above set) has substrings x and y that can be pumped.

x and y must contain at least one $\textcolor{blue}{1}$, or else all new strings generated by repeating x and y would have same number of $\textcolor{blue}{1}$'s, a contradiction.

But if x contains a $\textcolor{blue}{1}$, then any string that contains x^3 as a substring must have 2 pairs of $\textcolor{blue}{1}$'s with the same number of $\textcolor{blue}{0}$'s between them, also a contradiction.

Another example

Prove the following language is not a CFL: Let L be words of the form ww (where w is any word over $\{a, b, c\}$)

(e.g. aa , $abcabc$, $baaabaaa$, ...)

Let N be “sufficiently large” word length promised by pumping lemma.

Choose $w = a^{N+1}b^{N+1}a^{N+1}b^{N+1}$, so $w \in L$

We can argue that there is no subword of w of length N which contains any pair of subwords which, if repeated once, give another member of L .

Greibach normal form: all rules are of the form:

$$S \rightarrow \epsilon$$

$$X \rightarrow a$$

$$X \rightarrow aX_1X_2 \dots X_r$$

“You can replace a variable with a single constant followed by a possibly empty sequence of variables. As before start symbol only may be replaced by empty string.”

Any CFG is equivalent to some CFG in Greibach normal form.

Conversion to Greibach NF

We can assume grammar is already in Chomsky NF.

We describe 2 useful tricks to help us convert from Chomsky NF to Greibach NF.

trick 1: Removing Directly Left Recursive Productions

If I have $X \rightarrow X\alpha$
 $X \rightarrow \beta$

I can replace with
 $X \rightarrow \beta$
 $X \rightarrow \beta Y$
 $Y \rightarrow \alpha$
 $Y \rightarrow \alpha Y$

Both sets of rules say X is a β followed by a string of α 's.

Generally, if there is more than one α or β , add new rules of above form for all α 's and β 's.

trick 2: Removing a Right-hand variable from a Production.

Given $\begin{array}{l} X \rightarrow \alpha Y \beta \\ Y \rightarrow \gamma_1 | \gamma_2 | \dots | \gamma_n \end{array}$

(above are all rules with Y on LHS) we can replace $X \rightarrow \alpha Y \beta$ with

$$X \rightarrow \alpha \gamma_1 \beta | \alpha \gamma_2 \beta | \dots | \alpha \gamma_n \beta$$

Converting a CFG to Greibach Normal Form

Start by converting to Chomsky Normal Form. Then suppose I have e.g.

- (1) $S \rightarrow XY$ $X \rightarrow a$
- (2) $S \rightarrow XW$ $Y \rightarrow b$
- (3) $X \rightarrow YZ$ $Z \rightarrow c$
- (4) $Y \rightarrow WZ$ $W \rightarrow d$

Rules (1-4) can be rewritten:

(4) can be written as $Y \rightarrow dZ$

Then (3) can be written as $X \rightarrow dZZ | bZ$

(2) can be written as $S \rightarrow dZZW | bZW | aW$

(1) can be written as $S \rightarrow dZZY | bZY | aY$

We have converted to Greibach normal form.

We were able to arrange the variables in a sequence such that if a variable begins the RHS of a rule, it occurs later in the sequence than the LHS variable. Then we worked backwards through the sequence converting to Greibach normal form.

But what if rule (4) had been:

$$Y \longrightarrow XZ$$

then substituting for X in (4) would give us

$$Y \longrightarrow aZ \mid YZZ$$

Now what?

Problem seems to be a directly left recursive production.

General Approach

Given a Chomsky normal form grammar with variables $\{S, X_1, \dots, X_n\}$, add new variables Y_0, \dots, Y_n that will allow conversion to rules of the form:

$$S \rightarrow \epsilon \mid \langle \text{string of vars not starting with } S \rangle$$

$$S \rightarrow a \langle \text{string of vars} \rangle$$

$$X_i \rightarrow a \langle \text{string of vars} \rangle$$

$$X_i \rightarrow X_j \langle \text{string of vars} \rangle \quad j > i$$

$$Y_i \rightarrow X_j \langle \text{string of vars} \rangle \quad j > i$$

First use Y_0 to get rid of any directly left recursive productions for S .

Use substitution procedure to remove productions of the form

$$X_i \rightarrow S\alpha.$$

Then all productions with S or Y_0 on LHS are OK.

Use Y_1 to get rid of any directly left recursive productions for X_1 . Then substitution procedure to remove productions of the form $X_i \rightarrow X_1\alpha$, where $i > 1$.

Then Y_2 for X_2 , etc.

General Approach (continued)

Continue forward through the sequence, introducing Y_i if needed for X_i . When we finish, rules are of desired form. Then we substitute for variables backwards through the sequence.

$X_n \rightarrow$ RHS is in Greibach normal form.

$X_{n-1} / Y_{n-1} \rightarrow$ RHS; Here RHS may start with X_n , if so we can substitute to get Greibach normal form.

$X_{n-2} \rightarrow$ RHS; substitute again (since RHS may start with X_{n-1} , X_n , Y_{n-1} or Y_n .)

Example

- (1) $S \rightarrow XY \quad X \rightarrow a$
 (2) $S \rightarrow XW \quad Y \rightarrow b$
 (3) $X \rightarrow YZ \quad Z \rightarrow c$
 (4) $Y \rightarrow XZ \quad W \rightarrow d$

Arrange in order: S, X, Y, Z, W

(4) is unsatisfactory (w.r.t above ordering); replace with

$$\begin{aligned} Y &\rightarrow aZ \\ Y &\rightarrow YZZ \end{aligned}$$

Replace 2nd of these new rules with

$$\begin{aligned} Y' &\rightarrow ZZY' \mid ZZ \\ Y &\rightarrow aZY' \mid bY' \mid aZ \mid b \end{aligned}$$

(using the trick for directly left recursive productions; note
 $Y \Rightarrow^* (aZ \mid b)(ZZ)^*$ and $Y' \Rightarrow^* (ZZ)^+$)

Order: S, X, Y, Y', Z, W

Example (continued)

We have sequence: S, X, Y, Y', Z, W

$$\begin{array}{ll}
 (1) & S \rightarrow XY & X \rightarrow a \\
 (2) & S \rightarrow XW & Y \rightarrow b \\
 (3) & X \rightarrow YZ & Z \rightarrow c \\
 & Y \rightarrow aZ & W \rightarrow d \\
 & Y \rightarrow aZY' \mid bY' \mid aZ \mid b & \\
 & Y' \rightarrow ZZY' \mid ZZ &
 \end{array}$$

Don't need to substitute for W .

bottom rule becomes (by substituting for Z):

$$Y' \rightarrow cZY' \mid cZ$$

(3) becomes (by substituting for Y):

$$X \rightarrow aZZ \mid aZY'Z \mid bY'Z \mid bZ$$

(Noting we also have $X \rightarrow a$,) (1) and (2) become:

$$\begin{array}{l} S \rightarrow aZZY \quad | \quad aZY'ZY \quad | \quad bY'ZY \quad | \quad bZY \\ aZZW \quad | \quad aZY'ZW \quad | \quad bY'ZW \quad | \quad bZW \\ aY \quad | \quad aW \end{array}$$

The Pumping Lemma for CFLs

Given a CFL, any sufficiently long string in that CFL has *two* substrings (at least one of which is non-empty) such that if both of these substrings are “pumped” you generate further words in that CFL.

More formally...

Given a CFL L , there exists a number N such that any string $\alpha \in L$ with $|\alpha| \geq N$ can be written as

$$\alpha = sxtyu$$

such that

$$sx^2ty^2u, sx^3ty^3u, sx^4ty^4u, \dots$$

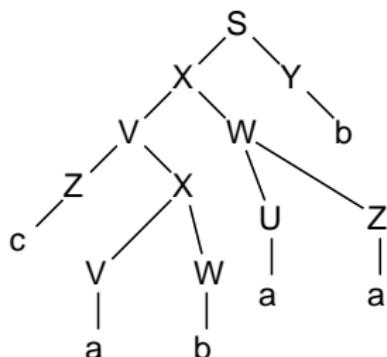
are all members of L , $|xty| \leq N$ and $|xy| > 0$ (which we need for all strings in this collection to be distinct).

How do we find suitable substrings x and y ?

Consider the following (Chomsky normal form) grammar

$$\begin{array}{lcl} S & \longrightarrow & XY \\ U & \longrightarrow & a \\ V & \longrightarrow & ZX \mid a \mid b \\ X & \longrightarrow & VW \mid a \\ Y & \longrightarrow & b \mid c \\ Z & \longrightarrow & a \mid c \\ W & \longrightarrow & UZ \mid b \end{array}$$

The string `cabaab` belongs to the language. Also it contains “suitable substrings” x and y which we can find by looking at a derivation tree.



We find two X 's on the same path. We can say:

$$X \Rightarrow^* cXaa$$

(via the sequence $X \Rightarrow VW \Rightarrow ZXW$
 $\Rightarrow cXW \Rightarrow cXUZ \Rightarrow cXaZ \Rightarrow cXaa$)

Generally: $X \Rightarrow^* ccXaaaa \Rightarrow^* cccXaaaaaaaa\dots$

The substrings c and aa can be pumped, because a derivation of $cabaab$ can go as follows:

$$\begin{aligned} S &\xrightarrow{*} Xb \\ &\xrightarrow{*} c\underline{X}aab \\ &\xrightarrow{*} cabaab \end{aligned}$$

but the underlined X could have been used to generate extra c 's and aa 's on each side of it.

Given a grammar, it's not hard to see that any sufficiently long string will have a derivation tree in which a path down to a leaf must contain a repeated variable.

If the grammar is in Chomsky normal form and has v variables, any string of length $> 2^v$ will necessarily have such a derivation tree.

Example

Prove the following is not a CFL:

{ 1 , 101 , 101001 , 1010010001 ,
 101001000100001 , ... }

Proof by contradiction: suppose string s (in above set) has substrings x and y that can be pumped.

x and y must contain at least one 1 , or else all new strings generated by repeating x and y would have same number of 1 's, a contradiction.

But if x contains a 1 , then any string that contains x^3 as a substring must have 2 pairs of 1 's with the same number of 0 's between them, also a contradiction.

Another example

Prove the following language is not a CFL: Let L be words of the form ww (where w is any word over $\{a, b, c\}$)

(e.g. aa , $abcabc$, $baaabaaa$, ...)

Let N be “sufficiently large” word length promised by pumping lemma.

Choose $w = a^{N+1}b^{N+1}a^{N+1}b^{N+1}$, so $w \in L$

We can argue that there is no subword of w of length N which contains any pair of subwords which, if repeated once, give another member of L .

1 Greibach Normal Form (GNF)

A CFG $G = (V, T, R, S)$ is said to be in GNF if every production is of the form $A \rightarrow a\alpha$, where $a \in T$ and $\alpha \in V^*$, i.e., α is a string of zero or more variables.

Definition: A production $\mathcal{U} \in R$ is said to be in the form **left recursion**, if $\mathcal{U} : A \rightarrow A\alpha$ for some $A \in V$.

Left recursion in R can be eliminated by the following scheme:

- If $A \rightarrow A\alpha_1|A\alpha_2|\dots|A\alpha_r|\beta_1|\beta_2|\dots|\beta_s$, then replace the above rules by
 - (i) $Z \rightarrow \alpha_i|\alpha_i Z, 1 \leq i \leq r$ and (ii) $A \rightarrow \beta_i|\beta_i Z, 1 \leq i \leq s$
- If $G = (V, T, R, S)$ is a CFG, then we can construct another CFG $G_1 = (V_1, T, R_1, S)$ in **Greibach Normal Form (GNF)** such that $L(G_1) = L(G) - \{\epsilon\}$.

The stepwise algorithm is as follows:

1. Eliminate null productions, unit productions and useless symbols from the grammar G and then construct a $G' = (V', T, R', S)$ in **Chomsky Normal Form (CNF)** generating the language $L(G') = L(G) - \{\epsilon\}$.
2. Rename the variables like A_1, A_2, \dots, A_n starting with $S = A_1$.
3. Modify the rules in R' so that if $A_i \rightarrow A_j\gamma \in R'$ then $j > i$
4. Starting with A_1 and proceeding to A_n this is done as follows:
 - (a) Assume that productions have been modified so that for $1 \leq i \leq k, A_i \rightarrow A_j\gamma \in R'$ only if $j > i$
 - (b) If $A_k \rightarrow A_j\gamma$ is a production with $j < k$, generate a new set of productions substituting for the A_j the body of each A_j production.
 - (c) Repeating (b) at most $k - 1$ times we obtain rules of the form $A_k \rightarrow A_p\gamma, p \geq k$
 - (d) Replace rules $A_k \rightarrow A_k\gamma$ by removing left-recursion as stated above.
5. Modify the $A_i \rightarrow A_j\gamma$ for $i = n - 1, n - 2, \dots, 1$ in desired form at the same time change the Z production rules.

Example: Convert the following grammar G into Greibach Normal Form (GNF).

$$\begin{array}{l} S \rightarrow XA|BB \\ B \rightarrow b|SB \\ X \rightarrow b \\ A \rightarrow a \end{array}$$

To write the above grammar G into GNF, we shall follow the following steps:

1. Rewrite G in Chomsky Normal Form (CNF)

It is already in CNF.

2. Re-label the variables

S with A_1

X with A_2

A with A_3

B with A_4

After re-labeling the grammar looks like:

$$A_1 \rightarrow A_2A_3|A_4A_4$$

$$A_4 \rightarrow b|A_1A_4$$

$$A_2 \rightarrow b$$

$$A_3 \rightarrow a$$

3. Identify all productions which do not conform to any of the types listed below:

$$A_i \rightarrow A_jx_k \text{ such that } j > i$$

$$Z_i \rightarrow A_jx_k \text{ such that } j \leq n$$

$$A_i \rightarrow ax_k \text{ such that } x_k \in V^* \text{ and } a \in T$$

4. $A_4 \rightarrow A_1A_4$ identified

5. $A_4 \rightarrow A_1A_4|b$.

To eliminate A_1 we will use the substitution rule $A_1 \rightarrow A_2A_3|A_4A_4$.

Therefore, we have $A_4 \rightarrow A_2A_3A_4|A_4A_4A_4|b$

The above two productions still do not conform to any of the types in step 3.

Substituting for $A_2 \rightarrow b$

$$A_4 \rightarrow bA_3A_4|A_4A_4A_4|b$$

Now we have to remove left recursive production $A_4 \rightarrow A_4A_4A_4$

$$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$$

$$Z \rightarrow A_4A_4|A_4A_4Z$$

6. At this stage our grammar now looks like

$$A_1 \rightarrow A_2A_3|A_4A_4$$

$$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$$

$$Z \rightarrow A_4A_4|A_4A_4Z$$

$$A_2 \rightarrow b$$

$$A_3 \rightarrow a$$

All rules now conform to one of the types in step 3.

But the grammar is still not in Greibach Normal Form!

7. All productions for A_2 , A_3 and A_4 are in GNF

for $A_1 \rightarrow A_2A_3|A_4A_4$

Substitute for A_2 and A_4 to convert it to GNF

$$A_1 \rightarrow bA_3|bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4$$

for $Z \rightarrow A_4A_4|A_4A_4Z$

Substitute for A_4 to convert it to GNF

$$Z \rightarrow bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4|bA_3A_4A_4Z|bA_4Z|bA_3A_4ZA_4Z|bZA_4Z$$

8. Finally the grammar in GNF is

$$A_1 \rightarrow bA_3|bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4$$

$$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$$

$$Z \rightarrow bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4|bA_3A_4A_4Z|bA_4Z|bA_3A_4ZA_4Z|bZA_4Z$$

$$A_2 \rightarrow b$$

$$A_3 \rightarrow a$$

Pushdown Automata

Pushdown automata (PDAs) have the same expressive power as CFGs (ie they accept CFLs)

A pushdown automaton is like a NFA but with an additional “memory stack” which can hold sequences of symbols from a memory alphabet.

Automaton scans an input from left to right - at each step it may push a symbol onto the stack, or pop the stack. It cannot read other elements of the stack.

Start with empty stack; accept if at end of string state is in subset $T \subseteq Q$ of accepting states and stack is empty.

Transitions

notation $\mathcal{P} = (Q, A, M, \delta, i, T)$ where M is stack alphabet and δ is transition function.

Action taken by machine is allowed to depend on top element of stack, input letter being read, and state. Action consists of new state, and possibly push/pop the stack.

Formally:

$$\delta : Q \times (A \cup \{\epsilon\}) \times (M \cup \{\epsilon\}) \rightarrow \mathbf{P}(Q \times (M \cup \{\epsilon\}))$$

i.e. for each combination of state, letter being read, and topmost stack symbol, we are given a set of allowable new states, and actions on stack.

for example:

$$\delta(q, a, m) = \{(q2, m2), (q3, m3)\}$$

means that in state q , if you read a with m at top of stack, you may move to state $q2$ and replace m with $m2$. Alternatively you may move to state $q3$ and replace m with $m3$.

$$\delta(q, a, \epsilon) = \{(q2, m2)\}$$

means in state q with input a , go to state $q2$ and push $m2$ on top of stack.

$$\delta(q, a, m) = \{(q2, \epsilon)\}$$

means in state q with input a , go to state $q2$ and pop m off stack.

Example: Palindromes

Input alphabet $A = \{a, b, c\}$

Use stack alphabet $M = \{a', b', c'\}$

states $Q = \{f, s\}$ (f is “reading first half”, s is “reading second half”)

Initial state f .

Accepting states $T = \{s\}$

Transitions:

$$\delta(f, a, \epsilon) = \{(f, a'), (s, \epsilon), (s, a')\}$$

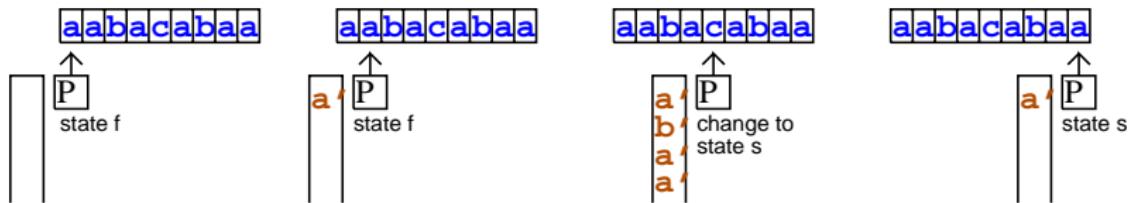
$$\delta(f, b, \epsilon) = \{(f, b'), (s, \epsilon), (s, b')\}$$

$$\delta(f, c, \epsilon) = \{(f, c'), (s, \epsilon), (s, c')\}$$

$$\delta(s, a, a') = \{(s, \epsilon)\} ; \delta(s, b, b') = \{(s, \epsilon)\} ; \delta(s, c, c') = \{(s, \epsilon)\} ;$$

$$\delta(\text{anything else}) = \emptyset$$

An Accepting Computation



Example of a Deterministic PDA

Consider palindromes over $\{a, b, c\}$ which contain exactly one c .

Use stack alphabet $M = \{a', b'\}$

states $Q = \{f, s\}$ (f is “reading first half”, s is “reading second half”)

Initial state f , accepting states $T = \{s\}$

Transitions:

$$\delta(f, a, \epsilon) = (f, a')$$

$$\delta(f, b, \epsilon) = (f, b')$$

$$\delta(f, c, \epsilon) = (s, \epsilon)$$

$$\delta(s, a, a') = (s, \epsilon); \delta(s, b, b') = (s, \epsilon);$$

$\delta(\text{anything else})$ is undefined (reject input).

EXAMPLE 2.9

The following is the formal description of the PDA from page 102 that recognizes the language $\{0^n 1^n \mid n \geq 0\}$. Let M_1 be $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where

$$Q = \{q_1, q_2, q_3, q_4\},$$

$$\Sigma = \{0, 1\},$$

$$\Gamma = \{0, \$\},$$

$$F = \{q_1, q_4\}, \text{ and}$$

δ is given by the following table, wherein blank entries signify \emptyset .

Input:	0	1	ϵ
Stack:	0 \$ ϵ	0 \$ ϵ	0 \$ ϵ
q_1			$\{(q_2, \$)\}$
q_2	$\{(q_2, 0)\}$	$\{(q_3, \epsilon)\}$	
q_3		$\{(q_3, \epsilon)\}$	$\{(q_4, \epsilon)\}$
q_4			

We can also use a state diagram to describe a PDA, as shown in the following three figures. Such diagrams are similar to the state diagrams used to describe finite automata, modified to show how the PDA uses its stack when going from state to state. We write " $a, b \rightarrow c$ " to signify that when the machine is reading an a from the input it may replace the symbol b on the top of the stack with a c . Any of a , b , and c may be ϵ . If a is ϵ , the machine may make this transition without reading any symbol from the input. If b is ϵ , the machine may make this transition without reading and popping any symbol from the stack. If c is ϵ , the machine does not write any symbol on the stack when going along this transition.

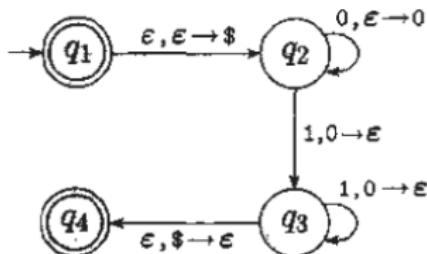


FIGURE 2.6

State diagram for the PDA M_1 that recognizes $\{0^n 1^n \mid n \geq 0\}$

Another deterministic example

PDA to recognise “well-formed” strings of parentheses

- A single state s (accepting)
- Input alphabet $\{(,)\}$
- Memory alphabet $\{x\}$

$$\delta(s, (, \epsilon) = \{(s, x)\}$$

$$\delta(s,), x) = \{(s, \epsilon)\}$$

Comments

- The number of x 's on the stack is the number of $($'s read so far minus number of $)$'s read.

PDA's and CFG's...

...define the same set of languages. To prove this,

- ① Given a CFL, construct a PDA that accepts it
- ② and given a PDA, construct a CFG that defines the language of that PDA.

We define “extended PDA”, a generalisation of PDA. But the extension will not allow extra languages to be accepted. Then we can show equivalence between extended PDA's and Greibach normal form grammars.

Extended PDAs

Allow transitions that write more than 1 memory symbol to the stack.

e.g.

$$\delta(s, a, x) = \{(t, uvw)\}$$

meaning: in state s, with input a and x on top of stack, change to state t, replace the x with uvw.

This could be replaced with:

$$\delta(s, a, x) = \{(s', w)\}$$

$$\delta(s', \epsilon, \epsilon) = \{(s'', v)\}$$

$$\delta(s'', \epsilon, \epsilon) = \{(t, u)\}$$

Converting GNF grammar to Ext. PDA

Use two states: initial state i , accepting state t

Variable symbols in grammar become the elements of the stack alphabet

Let S denote start symbol of grammar. Rules such as

$$S \longrightarrow a TUVW$$

become transitions

$$\delta(i, a, \epsilon) = \{(t, \textcolor{brown}{TUVW}), \dots\}$$

Rules such as

$$X \longrightarrow a TUVW$$

become transitions

$$\delta(t, a, X) = \{(t, \textcolor{brown}{TUVW}), \dots\}$$

Finally, to allow the empty string to belong to the language accepted by the PDA (when the grammar has the production $S \rightarrow \epsilon$):

Include transition

$$\delta(i, \epsilon, \epsilon) = \{(t, \epsilon)\}$$

So, to translate from a CFG to a PDA:

- ① Convert grammar to Greibach normal form
- ② Convert to extended PDA
- ③ Convert ext. PDA to a standard PDA

Example

Find a PDA that accepts “valid” expressions that use $(,), +, x$

e.g.

$$x + x + (x + x)$$

$$(x + (x + x))$$

Let us disallow consecutive pairs of nested parentheses e.g.

$$x + ((x + x))$$

also disallow singleton x in a pair of parentheses e.g.

$$x + (x) + x$$

Grammar

Start symbol E . Variable symbol F represents an expression with no matching pair of parentheses on the outside.

$$\begin{array}{l} E \rightarrow x \\ E \rightarrow (F) \\ E \rightarrow E+E \\ F \rightarrow E+E \end{array}$$

(We don't have e.g. $F \rightarrow x$, since (x) was disallowed as a substring.)

Need to convert to GNF.

Conversion to Greibach Normal Form

Eliminate leading variables on RHS's of productions. Luckily this is fairly simple (for this particular grammar), but we need to be careful in arguing new grammar is equivalent.

$$\begin{array}{l} E \rightarrow x \\ E \rightarrow (F) \\ E \rightarrow (F)+E \\ E \rightarrow x+E \end{array}$$

$$\begin{array}{l} F \rightarrow x+E \\ F \rightarrow (F)+E \\ F \rightarrow (F)+E+x \\ F \rightarrow x+E+E \end{array}$$

The last 2 F rules are redundant; we can remove them.

We now have the grammar

$$E \rightarrow x \mid (F) \mid (F)+E \mid x+E$$

$$F \rightarrow x+E \mid (F)+E$$

Now introduce some variables to represent non-leading alphabet symbols

$$X_j \rightarrow)$$

$$X_+ \rightarrow +$$

$$E \rightarrow (FX_j \mid x \mid (FX_j)X_+E \mid xX_+E$$

$$F \rightarrow xX_+E \mid (FX_j)X_+E$$

This is now in Greibach Normal Form.

Convert to (extended) PDA

states $Q = \{i, t\}$ (initial state, accepting state)

Input alphabet $A = \{\textcolor{blue}{(}, \textcolor{blue}{)}, \textcolor{blue}{+}, \textcolor{blue}{x}\}$

Memory alphabet $M = \{\textcolor{brown}{X)}, \textcolor{brown}{X_+}, \textcolor{brown}{E}, \textcolor{brown}{F}\}$

Transitions:

$$\delta(i, \textcolor{blue}{(}, \epsilon) = \{(t, \textcolor{brown}{FX})\}, (t, \textcolor{brown}{FX})\textcolor{brown}{X}_+\textcolor{brown}{E})\}$$

$$\delta(i, \textcolor{blue}{x}, \epsilon) = \{(t, \epsilon), (t, \textcolor{brown}{X}_+\textcolor{brown}{E})\}$$

$$\delta(t, \textcolor{blue}{(}, \textcolor{brown}{X}) = \{(t, \epsilon)\}$$

$$\delta(t, \textcolor{blue}{+}, \textcolor{brown}{X}_+) = \{(t, \epsilon)\}$$

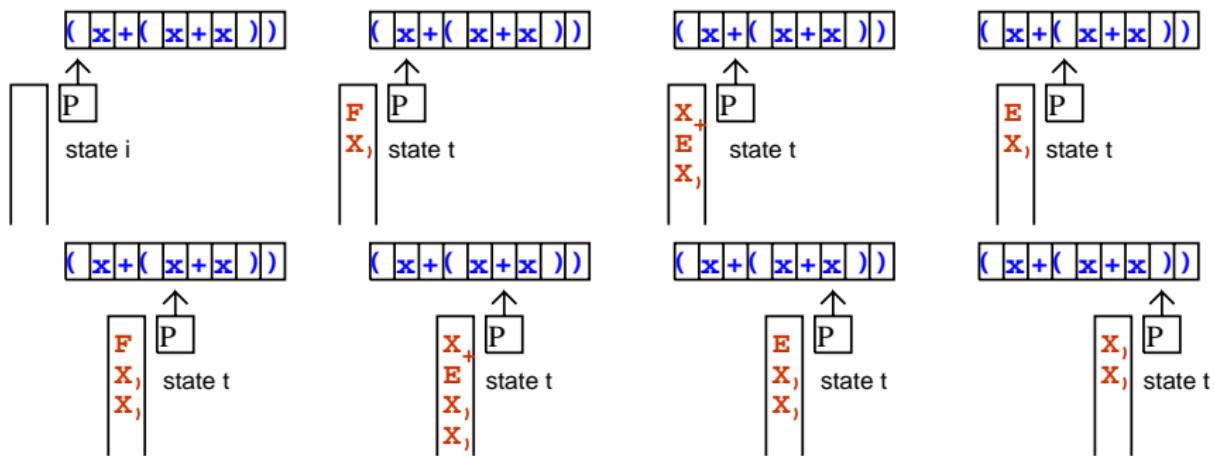
$$\delta(t, \textcolor{blue}{x}, \textcolor{brown}{F}) = \{(t, \textcolor{brown}{X}_+\textcolor{brown}{E})\}$$

$$\delta(t, \textcolor{blue}{(}, \textcolor{brown}{E}) = \{(t, \textcolor{brown}{FX})\}, (t, \textcolor{brown}{FX})\textcolor{brown}{X}_+\textcolor{brown}{E})\}$$

$$\delta(t, \textcolor{blue}{x}, \textcolor{brown}{E}) = \{(t, \epsilon), (t, \textcolor{brown}{X}_+\textcolor{brown}{E})\}$$

$$\delta(t, \textcolor{blue}{(}, \textcolor{brown}{F}) = \{(t, \textcolor{brown}{FX})\textcolor{brown}{X}_+\textcolor{brown}{E})\}$$

A Computation



The Associated Leftmost Derivation

$$\begin{aligned}
 E &\implies (\textcolor{blue}{F}X_j) \\
 &\implies (\textcolor{blue}{x}\ X_+EX_j) \\
 &\implies (\textcolor{blue}{x}+\textcolor{blue}{E}X_j) \\
 &\implies (\textcolor{blue}{x}+(\textcolor{blue}{F}X_j)X_j) \\
 &\implies (\textcolor{blue}{x}+(\textcolor{blue}{x}\ X_+EX_j)X_j) \\
 &\implies (\textcolor{blue}{x}+(\textcolor{blue}{x}+\textcolor{blue}{E}X_j)X_j) \\
 &\implies (\textcolor{blue}{x}+(\textcolor{blue}{x}+\textcolor{blue}{x}\ X_j)X_j) \\
 &\implies (\textcolor{blue}{x}+(\textcolor{blue}{x}+\textcolor{blue}{x})\ X_j) \\
 &\implies (\textcolor{blue}{x}+(\textcolor{blue}{x}+\textcolor{blue}{x}))
 \end{aligned}$$

Grammar:

$$E \rightarrow (\textcolor{blue}{F}X_j \mid \textcolor{blue}{x} \mid (\textcolor{blue}{F}X_j)X_+E \mid \textcolor{blue}{x}X_+E)$$

$$F \rightarrow \textcolor{blue}{x}X_+E \mid (\textcolor{blue}{F}X_j)X_+E$$

$$X_j \rightarrow)$$

$$X_+ \rightarrow +$$

Conversion from PDA to CFG

Problem: Given a PDA P , construct a CFG G , such that $L(P) = L(G)$, where $L(P)$ is the language accepted by P and $L(G)$ is the language defined by G .

Now we prove the reverse direction of Theorem 2.12. For the forward direction we gave a procedure for converting a CFG into a PDA. The main idea was to design the automaton so that it simulates the grammar. Now we want to give a procedure for going the other way: converting a PDA into a CFG. We design the grammar to simulate the automaton. This task is a bit tricky because “programming” an automaton is easier than “programming” a grammar.

LEMMA 2.15

If a pushdown automaton recognizes some language, then it is context free.

PROOF IDEA We have a PDA P , and we want to make a CFG G that generates all the strings that P accepts. In other words, G should generate a string if that string causes the PDA to go from its start state to an accept state.

To achieve this outcome we design a grammar that does somewhat more. For each pair of states p and q in P the grammar will have a variable A_{pq} . This variable generates all the strings that can take P from p with an empty stack to q with an empty stack. Observe that such strings can also take P from p to q , regardless of the stack contents at p , leaving the stack at q in the same condition as it was at p .

First, we simplify our task by modifying P slightly to give it the following three features.

1. It has a single accept state, q_{accept} .
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but does not do both at the same time.

Giving P features 1 and 2 is easy. To give it feature 3, we replace each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state, and we replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol.

To design G so that A_{pq} generates all strings that take P from p to q , starting and ending with an empty stack, we must understand how P operates on these strings. For any such string x , P 's first move on x must be a push, because every move is either a push or a pop and P can't pop an empty stack. Similarly the last move on x must be a pop, because the stack ends up empty.

Two possibilities occur during P 's computation on x . Either the symbol popped at the end is the symbol that was pushed at the beginning, or not. If so, the stack is empty only at the beginning and end of P 's computation on x . If not, the initially pushed symbol must get popped at some point before the end of x and thus the stack becomes empty at this point. We simulate the former possibility with the rule $A_{pq} \rightarrow aA_{rs}b$ where a is the input symbol read at the first move, b is the symbol read at the last move, r is the state following p , and s the state preceding q . We simulate the latter possibility with the rule $A_{pq} \rightarrow A_{pr}A_{rq}$, where r is the state when the stack becomes empty.

PROOF Say that $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ and construct G . The variables of G are $\{A_{pq} \mid p, q \in Q\}$. The start variable is $A_{q_0, q_{\text{accept}}}$. Now we describe G 's rules.

- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε) put the rule $A_{pq} \rightarrow aA_{rs}b$ in G .
- For each $p, q, r \in Q$ put the rule $A_{pq} \rightarrow A_{pr}A_{rq}$ in G .
- Finally, for each $p \in Q$ put the rule $A_{pp} \rightarrow \varepsilon$ in G .

You may gain some intuition for this construction from the following figures.

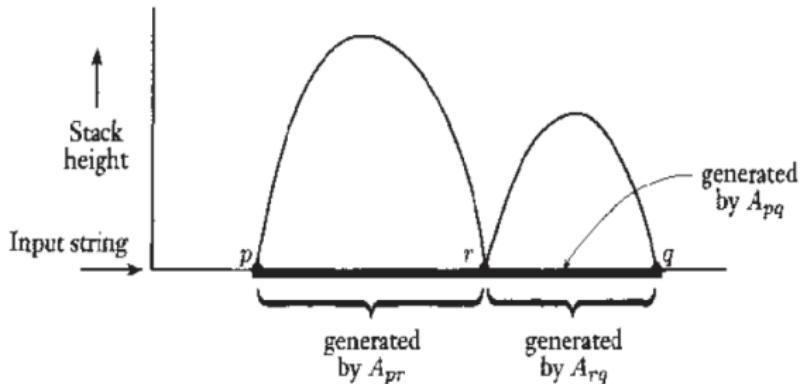


FIGURE 2.13

PDA computation corresponding to the rule $A_{pq} \rightarrow A_{pr}A_{rq}$

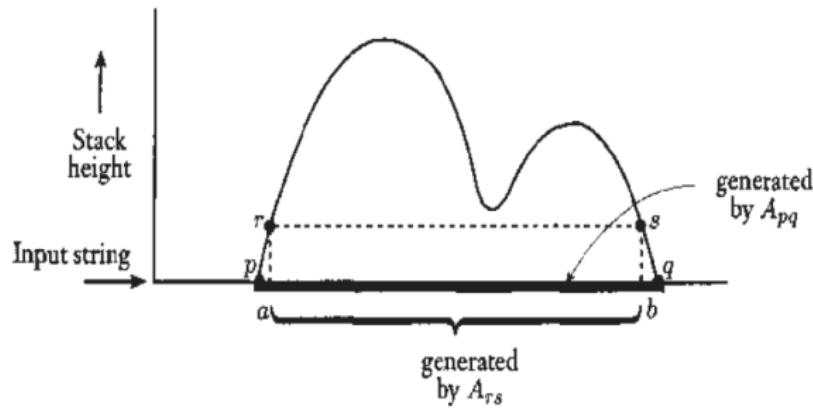


FIGURE 2.14

PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

CLAIM 2.16

If A_{pq} generates x , then x can bring P from p with empty stack to q with empty stack.

CLAIM 2.17

If x can bring P from p with empty stack to q with empty stack, A_{pq} generates x .

COROLLARY 2.18

Every regular language is context free.

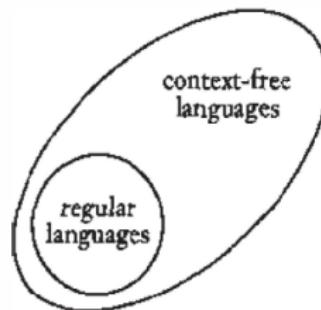


FIGURE 2.15

Relationship of the regular and context-free languages

EXAMPLES OF PUSHDOWN AUTOMATA

EXAMPLE 2.9

The following is the formal description of the PDA from page 102 that recognizes the language $\{0^n 1^n \mid n \geq 0\}$. Let M_1 be $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where

$$Q = \{q_1, q_2, q_3, q_4\},$$

$$\Sigma = \{0, 1\},$$

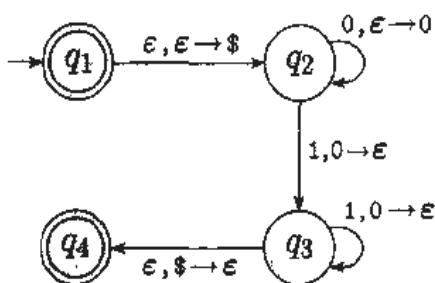
$$\Gamma = \{0, \$\},$$

$$F = \{q_1, q_4\}, \text{ and}$$

δ is given by the following table, wherein blank entries signify \emptyset .

Input:	0		1		ϵ	
Stack:	0	\$	ϵ	0	\$	ϵ
q_1						$\{(q_2, \$)\}$
q_2		$\{(q_2, 0)\}$	$\{(q_3, \epsilon)\}$			
q_3			$\{(q_3, \epsilon)\}$			$\{(q_4, \epsilon)\}$
q_4						

We can also use a state diagram to describe a PDA, as shown in the following three figures. Such diagrams are similar to the state diagrams used to describe finite automata, modified to show how the PDA uses its stack when going from state to state. We write " $a, b \rightarrow c$ " to signify that when the machine is reading an a from the input it may replace the symbol b on the top of the stack with a c . Any of a , b , and c may be ϵ . If a is ϵ , the machine may make this transition without reading any symbol from the input. If b is ϵ , the machine may make this transition without reading and popping any symbol from the stack. If c is ϵ , the machine does not write any symbol on the stack when going along this transition.

**FIGURE 2.6**

State diagram for the PDA M_1 that recognizes $\{0^n 1^n \mid n \geq 0\}$

EXAMPLE 2.14

We use the procedure developed in Lemma 2.13 to construct a PDA P_1 from the following CFG G .

$$\begin{aligned} S &\rightarrow aTb \mid b \\ T &\rightarrow Ta \mid \epsilon \end{aligned}$$

The transition function is shown in the following diagram.

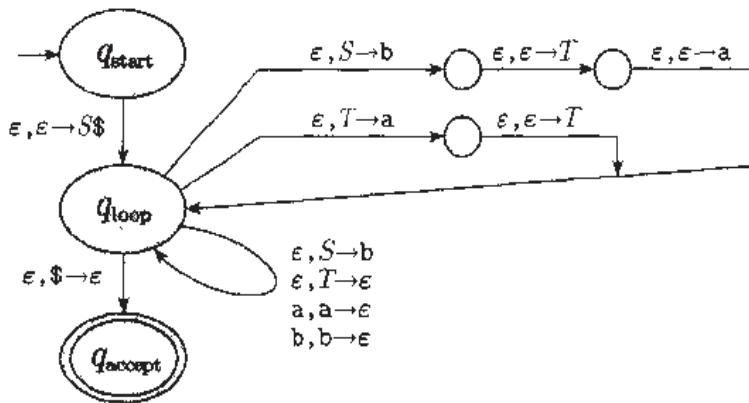


FIGURE 2.12
State diagram of P_1

Now we prove the reverse direction of Theorem 2.12. For the forward direction we gave a procedure for converting a CFG into a PDA. The main idea was to design the automaton so that it simulates the grammar. Now we want to give a procedure for going the other way: converting a PDA into a CFG. We design the grammar to simulate the automaton. This task is a bit tricky because “programming” an automaton is easier than “programming” a grammar.

LEMMA 2.15

If a pushdown automaton recognizes some language, then it is context free.

PROOF IDEA We have a PDA P , and we want to make a CFG G that generates all the strings that P accepts. In other words, G should generate a string if that string causes the PDA to go from its start state to an accept state.

To achieve this outcome we design a grammar that does somewhat more. For each pair of states p and q in P the grammar will have a variable A_{pq} . This variable generates all the strings that can take P from p with an empty stack to q with an empty stack. Observe that such strings can also take P from p to q , regardless of the stack contents at p , leaving the stack at q in the same condition as it was at p .

First, we simplify our task by modifying P slightly to give it the following three features.

1. It has a single accept state, q_{accept} .
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push move*) or pops one off the stack (a *pop move*), but does not do both at the same time.

Giving P features 1 and 2 is easy. To give it feature 3, we replace each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state, and we replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol.

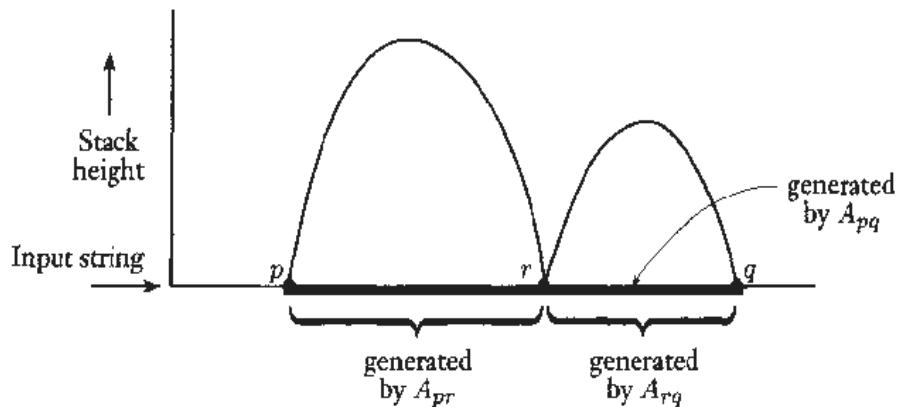
To design G so that A_{pq} generates all strings that take P from p to q , starting and ending with an empty stack, we must understand how P operates on these strings. For any such string x , P 's first move on x must be a push, because every move is either a push or a pop and P can't pop an empty stack. Similarly the last move on x must be a pop, because the stack ends up empty.

Two possibilities occur during P 's computation on x . Either the symbol popped at the end is the symbol that was pushed at the beginning, or not. If so, the stack is empty only at the beginning and end of P 's computation on x . If not, the initially pushed symbol must get popped at some point before the end of x and thus the stack becomes empty at this point. We simulate the former possibility with the rule $A_{pq} \rightarrow aA_{rs}b$ where a is the input symbol read at the first move, b is the symbol read at the last move, r is the state following p , and s the state preceding q . We simulate the latter possibility with the rule $A_{pq} \rightarrow A_{pr}A_{rq}$, where r is the state when the stack becomes empty.

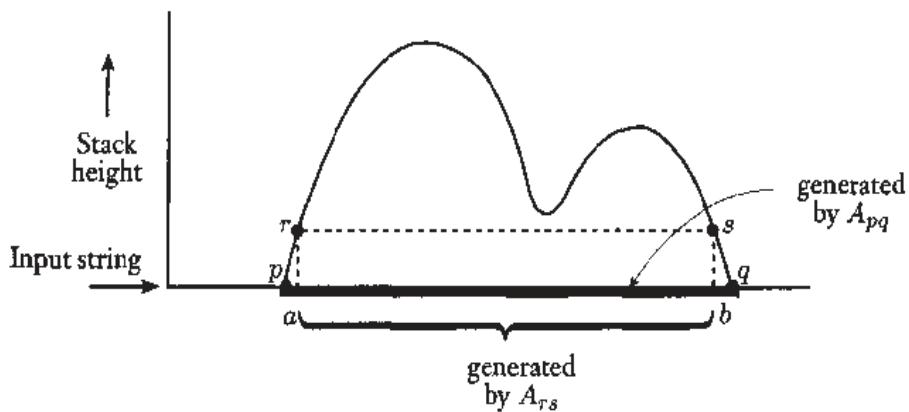
PROOF Say that $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ and construct G . The variables of G are $\{A_{pq} \mid p, q \in Q\}$. The start variable is $A_{q_0, q_{\text{accept}}}$. Now we describe G 's rules.

- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε) put the rule $A_{pq} \rightarrow aA_{rs}b$ in G .
- For each $p, q, r \in Q$ put the rule $A_{pq} \rightarrow A_{pr}A_{rq}$ in G .
- Finally, for each $p \in Q$ put the rule $A_{pp} \rightarrow \varepsilon$ in G .

You may gain some intuition for this construction from the following figures.

**FIGURE 2.13**

PDA computation corresponding to the rule $A_{pq} \rightarrow A_{pr}A_{rq}$

**FIGURE 2.14**

PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

Now we prove that this construction works by demonstrating that A_{pq} generates x if and only if (iff) x can bring P from p with empty stack to q with empty stack. We consider each direction of the iff as a separate claim.

CLAIM 2.16

If A_{pq} generates x , then x can bring P from p with empty stack to q with empty stack.

We prove this claim by induction on the number of steps in the derivation of x from A_{pq} .

Basis: The derivation has 1 step.

A derivation with a single step must use a rule whose right-hand side contains no variables. The only rules in G where no variables occur on the right-hand side are $A_{pp} \rightarrow \epsilon$. Clearly, input ϵ takes P from p with empty stack to p with empty stack so the basis is proved.

Induction step: Assume true for derivations of length at most k , where $k \geq 1$, and prove true for derivations of length $k + 1$.

Suppose that $A_{pq} \xrightarrow{*} x$ with $k + 1$ steps. The first step in this derivation is either $A_{pq} \Rightarrow aA_{rs}b$ or $A_{pq} \Rightarrow A_{pr}A_{rq}$. We handle these two cases separately.

In the first case, consider the portion y of x that A_{rs} generates, so $x = ayb$. Because $A_{rs} \xrightarrow{*} y$ with k steps, the induction hypothesis tells us that P can go from r on empty stack to s on empty stack. Because $A_{pq} \rightarrow aA_{rs}b$ is a rule of G , $\delta(p, a, \epsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ϵ) . Hence, if P starts at p with an empty stack, after reading a it can go to state r and push t on the stack. Then reading string y can bring it to s and leave t on the stack. Then after reading b it can go to state q and pop t off the stack. Therefore x can bring it from p with empty stack to q with empty stack.

In the second case, consider the portions y and z of x that A_{pr} and A_{rq} respectively generate, so $x = yz$. Because $A_{pr} \xrightarrow{*} y$ in at most k steps and $A_{rq} \xrightarrow{*} z$ in at most k steps, the induction hypothesis tells us that y can bring P from p to r , and z can bring P from r to q , with empty stacks at the beginning and end. Hence x can bring it from p with empty stack to q with empty stack. This completes the induction step.

CLAIM 2.17

If x can bring P from p with empty stack to q with empty stack, A_{pq} generates x .

We prove this claim by induction on the number of steps in the computation of P that goes from p to q with empty stacks on input x .

Basis: The computation has 0 steps.

If a computation has 0 steps, it starts and ends at the same state, say, p . So we must show that $A_{pp} \xrightarrow{*} x$. In 0 steps, P only has time to read the empty string, so $x = \epsilon$. By construction, G has the rule $A_{pp} \rightarrow \epsilon$, so the basis is proved.

Induction step: Assume true for computations of length at most k , where $k \geq 0$, and prove true for computations of length $k + 1$.

Suppose that P has a computation wherein x brings p to q with empty stacks in $k + 1$ steps. Either the stack is empty only at the beginning and end of this computation, or it becomes empty elsewhere, too.

In the first case, the symbol that is pushed at the first move must be the same as the symbol that is popped at the last move. Call this symbol t . Let a be the input read in the first move, b be the input read in the last move, r be the state after the first move, and s be the state before the last move. Then $\delta(p, a, \epsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ϵ) , and so rule $A_{pq} \rightarrow aA_{rs}b$ is in G .

Let y be the portion of x without a and b , so $x = ayb$. Input y can bring P from r to s without touching the symbol t that is on the stack and so P can go from r with an empty stack to s with an empty stack on input y . We have removed the first and last steps of the $k + 1$ steps in the original computation on x so the computation on y has $(k + 1) - 2 = k - 1$ steps. Thus the induction hypothesis tells us that $A_{rs} \xrightarrow{*} y$. Hence $A_{pq} \xrightarrow{*} x$.

In the second case, let r be a state where the stack becomes empty other than at the beginning or end of the computation on x . Then the portions of the computation from p to r and from r to q each contain at most k steps. Say that y is the input read during the first portion and z is the input read during the second portion. The induction hypothesis tells us that $A_{pr} \xrightarrow{*} y$ and $A_{rq} \xrightarrow{*} z$. Because rule $A_{pq} \rightarrow A_{pr}A_{rq}$ is in G , $A_{pq} \xrightarrow{*} x$, and the proof is complete.

That completes the proof of Lemma 2.15 and of Theorem 2.12.

We have just proved that pushdown automata recognize the class of context-free languages. This proof allows us to establish a relationship between the regular languages and the context-free languages. Because every regular language is recognized by a finite automaton and every finite automaton is automatically a pushdown automaton that simply ignores its stack, we now know that every regular language is also a context-free language.

COROLLARY 2.18

Every regular language is context free.

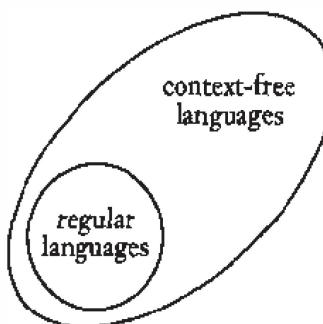


FIGURE 2.15

Relationship of the regular and context-free languages

Context-sensitive grammars: rules must be of the form

$$\alpha X \beta \longrightarrow \alpha \gamma \beta$$

where X is a variable; RHS is at least as long as LHS, which makes it possible to check whether a given word belongs to the language...

Unrestricted grammar: Rules of the form

$$\alpha \longrightarrow \beta$$

α and β can be any string of symbols.

...Problem: given a word, how to check that it is derivable? There is no obvious limit on how long the intermediate words in the derivation may be. Unrestricted grammars are equivalent to **Turing machines**, let's look at TMs first.

Turing Machines (TMs)

equivalent to unrestricted grammars (in terms of expressive power)

Like a PDA, a TM has access to an infinite memory. But TM's memory is a "tape" not a stack, and TM is free to scan the tape taking actions determined by its state and tape symbol(s) being scanned.

Not just a class of language acceptors, TMs are a standard representation of "effectively computable function"

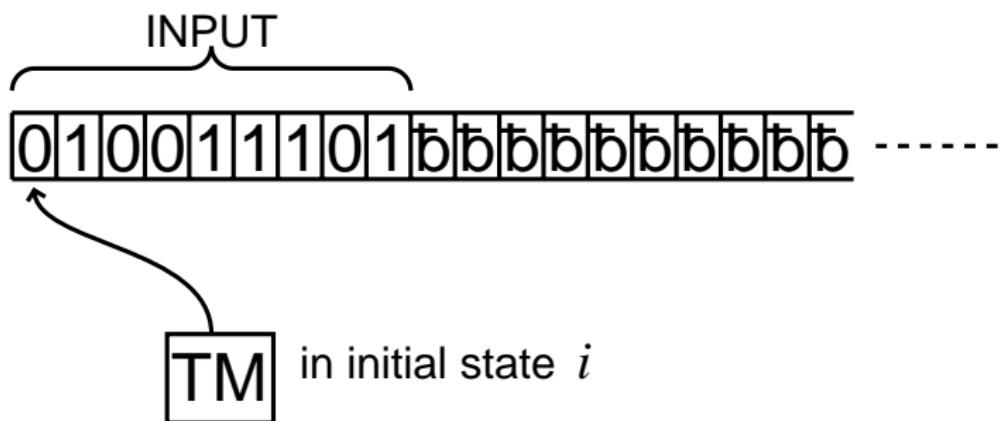
You have to look rather hard to find a language not accepted by some TM.

Definition

A TM has:

- input alphabet I (e.g. $\{0, 1\}$)
- “blank symbol” not in I , usually denoted \bar{b} .
- tape alphabet A containing I and \bar{b} , possibly also extra symbols such as $\#, \$$ or $\%$.
- finite set Q of states, where Q includes a subset T of accepting states
- Transition function $Q \times A \rightarrow Q \times A \times \{L, R\}$

Initial configuration of Turing Machine



TM's computation ends if it encounters a combination of alphabet symbol and state for which no transition is defined, *or* if it tries to go off left-hand end of tape.

When TM's computation ends we say it **halts**. An input is accepted if a TM halts in an accepting state.

Some comments

Unlike DFAs or PDAs, *no guarantee that TM will halt*. It may “end up” stuck at a particular square, or endlessly moving right, reading blank symbols. Deemed to reject.

If a TM halts on some given input, we can regard the contents of the tape at the end of the computation as being an output. So a TM can be viewed as computing a function.

TMs can simulate PDAs (We'll see how later on)

Example: Palindromes over binary alphabet

States: $\{i, p_0, p_1, q_0, q_1, r, t\}$

i – initial state

p_0 – look for 0 at RHS

p_1 – look for 1 at RHS

q_0 – found RHS; check it's 0

q_1 – found RHS; check it's 1

r – return to beginning

t – accepting



Transitions

$\delta(i, \bar{b}) = (t, \bar{b}, R)$ accept if tape is blank

$\delta(i, 0) = (p_0, \bar{b}, R)$ delete 0; look for 0 at RHS

$\delta(i, 1) = (p_1, \bar{b}, R)$ delete 1; look for 0 at RHS

$\delta(p_0, 0) = (p_0, 0, R)$ move right to RHS

$\delta(p_0, 1) = (p_0, 1, R)$ " " "

$\delta(p_1, 0) = (p_1, 0, R)$ " " "

$\delta(p_1, 1) = (p_1, 1, R)$ " " "

$\delta(p_0, \bar{b}) = (q_0, \bar{b}, L)$ found RHS; now check

$\delta(p_1, \bar{b}) = (q_1, \bar{b}, L)$ whether 0 or 1

$\delta(q_0, 0) = (r, \bar{b}, L)$ check RHS is 0; delete it

$\delta(q_1, 1) = (r, \bar{b}, L)$ check RHS is 1 and delete

Transitions

$\delta(q_0, \bar{b}) = (t, \bar{b}, R)$ accept if all tape is

$\delta(q_1, \bar{b}) = (t, \bar{b}, R)$ blank

$\delta(r, 0) = (r, 0, L)$ return to LHS

$\delta(r, 1) = (r, 1, L)$ " "

$\delta(r, \bar{b}) = (i, \bar{b}, R)$ found LHS; goto state 1

Example

Find a TM that accepts words $w2w$ where $w \in \{0, 1\}^*$

e.g. 01201, 110021100

The language is not a CFL.

Input alphabet $\{0, 1, 2\}$; we will use tape alphabet $\{0, 1, 2, \#, \bar{b}\}$

States:

- i initial state
- t accepting state
- s_0, s'_0 look for a corresp. 0
- s_1, s'_1 look for a corresp. 1
- s_2 check there are no more 0's and 1's to right
- r, r' look for leftmost 0/1 in first half of input

Transitions

$$\delta(i, 0) = (s_0, \#, R)$$

$$\delta(i, 1) = (s_1, \#, R) \quad \text{identify leftmost symbol}$$

$$\delta(i, \bar{b}) = (t, \#, R) \quad \text{that's not \#}$$

$$\delta(i, 2) = (s_2, \#, R)$$

$$\delta(s_0, 0) = (s_0, 0, R) \quad \delta(s_1, 0) = (s_1, 0, R)$$

$$\delta(s_0, 1) = (s_0, 1, R) \quad \delta(s_1, 1) = (s_1, 1, R)$$

$$\delta(s_0, 2) = (s'_0, 2, R) \quad \delta(s_1, 2) = (s'_1, 2, R)$$

$$\delta(s'_0, \#) = (s'_0, \#, R) \quad \delta(s'_1, \#) = (s'_1, \#, R)$$

$$\delta(s'_0, 0) = (r, \#, L) \quad \delta(s'_1, 1) = (r, \#, L)$$

$$\delta(s_2, \#) = (s_2, \#, R) \quad \text{accept if everything to}$$

$$\delta(s_2, \bar{b}) = (t, \bar{b}, R) \quad \text{right is \# or \bar{b}}$$

$$\delta(r, \#) = (r, \#, L)$$

$$\delta(r, 2) = (r', 2, L)$$

$$\delta(r', 0) = (r', 0, L) \quad \text{look for leftmost 0 or 1.}$$

$$\delta(r', 1) = (r', 1, L)$$

$$\delta(r', \#) = (i, \#, R)$$

Example

Design a TM that accepts

$$\{1, 11, 1101, 1101001, 11010010001, \dots\}$$

General idea:

Let P_i be smallest prefix of input containing i 1's.

TM will check that P_i is good. Then move to right of P_i .

If \bar{b} , accept.

If 1 , for $i > 1$, reject.

If 0 , check P_{i+1}

To check P_{i+1} : Check that next block of 0's is:

followed by 1

contains one more 0 than previous block.

Input alphabet {0, 1}

Tape alphabet {0, 1, #, \bar{b} }

States: i : initial state; i' : second state

p_0 : just moved to right of prefix P_i

q_0 : found 0 to right of P_i ; replace with # and check next block of 0's has equal length to previous

q_1, q'_1 : move to left, turn two 0's into #'s

r_0, r_1 : move to right to repeat process of turning two 0's into #'s

q_2, q'_2 : move to left, check we have no more 0's in either block (i.e. all 0's replaced with #'s); x_0 and x_1 have a similar function.

s_0, s_1, s_2 : verified a new block of 0's (and replaced them with #'s) – now replace them (left to right) with 0's.

t : accepting state

Transitions

$$\delta(i, \underline{1}) = (i', \underline{1}, R) \quad i: \text{ initial state} \quad \delta(i', \bar{\underline{b}}) = (\textcolor{red}{t}, \bar{\underline{b}}, R)$$

$$\delta(i', \underline{1}) = (p_0, \underline{1}, R)$$

$$\delta(p_0, \bar{\underline{b}}) = (\textcolor{red}{t}, \bar{\underline{b}}, R) \quad p_0: \text{ just moved right of } P_i$$

$$\delta(p_0, \underline{0}) = (q_0, \#, R) \quad \delta(q_1, \#) = (q_1, \#, L)$$

$$\delta(q_0, \underline{0}) = (q_1, \#, L) \quad \delta(q_1, \underline{1}) = (q'_1, \underline{1}, L)$$

$$\delta(q_0, \underline{1}) = (q_2, \underline{1}, L) \quad \delta(q'_1, \#) = (q'_1, \#, L)$$

$$\delta(q_0, \#) = (q_0, \#, R) \quad \delta(q'_1, \underline{0}) = (r_0, \#, R)$$

$$\delta(r_0, \underline{1}) = (q_0, \underline{1}, R) \quad \text{next, turn 2 more } \underline{0}'\text{s into } \#'\text{s}$$

$$\delta(q_2, \#) = (q_2, \#, L)$$

$$\delta(q_2, \underline{1}) = (q'_2, \underline{1}, L)$$

$$\delta(q'_2, \#) = (q'_2, \#, L)$$

$$\delta(q'_2, \underline{1}) = (s_0, \underline{1}, R)$$

s_0, s_1, s_2 : replace $\#$ with $\underline{0}$

$$\delta(s_0, \#) = (s_0, \underline{0}, R) \quad \delta(s_1, \underline{1}) = (s_2, \underline{1}, R)$$

$$\delta(s_0, \underline{1}) = (s_1, \underline{1}, R) \quad \delta(s_2, \#) = (s_2, \underline{0}, R)$$

$$\delta(s_1, \#) = (s_1, \underline{0}, R) \quad \delta(s_2, \underline{1}) = (p_0, \underline{1}, R)$$

Move to right, looking for another 0 in rightmost block being checked

so far: $\delta(r_0, 1) = (r_1, 1, R)$ $\delta(r_1, \#) = (r_1, \#, R)$
 $\delta(r_1, 1) = (x_0, 1, L)$

where the last one of the above 3 deals with finding a 1 when there are no zeroes left. When we find a 0 in rightmost block, get ready to look for another 0 in previous block as follows:

$$\delta(r_1, 0) = (d, 0, R)$$

$$\delta(d, \alpha) = (q_0, \alpha, L) \quad \text{for all symbols } \alpha$$

Return to left of place where #'s are located, prior to replacing them with 0's:

$$\delta(x_0, \#) = (x_0, \#, L)$$

$$\delta(x_0, 1) = (x_1, 1, L)$$

$$\delta(x_1, \#) = (x_1, \#, L)$$

$$\delta(x_1, 1) = (s_0, 1, R)$$

An accepting computation

exercise: write down accepting computation given input [1101001](#)

Languages accepted by TMs

Languages accepted by TMs are called *recursively enumerable* languages.

Other models of computation characterise the r.e. languages (e.g. Markov algorithms, random access machines); also infinite memory models in conjunctions with various programming languages.

Also equivalent are non-deterministic TMs, and other variants of TM. So definition is very "robust".

To show acceptability by a TM, it is sufficient to show acceptability by any of these other models.

Languages accepted by TMs that are guaranteed to halt are called *recursive* languages. (*not quite the same!*)

More on Recursively Enumerable

Languages

- “recursive” refers to existence of a computational procedure...
- “enumerable” ...which generates members of a language (i.e. enumerates the words)
- *very large* class of languages (try to think of a language that you *cannot* define a TM for)

Church-Turing Thesis (“Church’s hypothesis” in Hopcroft and Ullman’s textbook) says that TMs represent the class of all functions which can be computed.

An alternative TM definition

“Extended TM” has transition function

$$\delta : Q \times A \longrightarrow Q \times A \times \{L, R, S\}$$

where S means stay at same tape square.

Claim: extended TMs accept the same languages as TMs.

Anything accepted by a TM is accepted by an extended TM (since any standard TM is an extended TM). Need to show that any language accepted by an extended TM is accepted a standard TM.

Suppose language L is accepted by extended TM T .

Given any transition in T $\delta(q_1, a_1) = (q_2, a_2, S)$

Replace it with $\delta(q_1, a_1) = (q_{new}, a_2, R)$ and

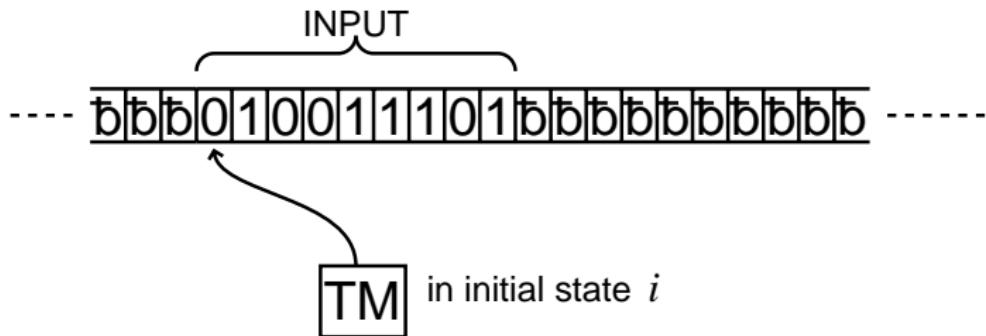
for all $a \in A$: $\delta(q_{new}, a) = (q_2, a, L)$

The new state q_{new} appears in all new transitions introduced above, but not in any other.

Do the same thing for all transitions with an S . Resulting TM accepts the same language.

Another (more interesting) variation

A TM *with doubly infinite tape* is one whose tape is infinite in both directions:



Claim: TMs with doubly infinite tape also accept the r.e. languages.

Converting “standard” TM into equivalent doubly infinite tape TM:

Use two new states that initially place a new character to left of input, then halt if that character is ever encountered by new machine.

New initial state i' has transitions

$$\begin{array}{lll} \delta(i', 0) = (i'_0, \Delta, R) & \delta(i'_0, 0) = (i'_0, 0, R) & \delta(i'_1, 0) = (i'_0, 1, R) \\ \delta(i', 1) = (i'_1, \Delta, R) & \delta(i'_0, 1) = (i'_1, 0, R) & \delta(i'_1, 1) = (i'_1, 1, R) \end{array}$$

Then, in states i'_0 and i'_1 , on encountering \bar{b} , replace \bar{b} with either 0 or 1 , enter a new state that moves to left, finds Δ , move right and enter state 1.

Converting doubly infinite tape TM into standard TM:

If A is original tape alphabet, use new alphabet $A \times A$. Treat the tape as doubly-infinite tape that is folded at the origin.

New blank symbol is (\bar{b}, \bar{b}) .

Number of states must be doubled to keep track of which side of the doubly-infinite tape the original machine is on.

TMs and grammars

Let M be a TM. Convert M into an unrestricted grammar G so that M accepts word w iff G can generate w .

The above transformation would show that unrestricted grammars have at least as much expressive power as TMs.

M has input alphabet I ; let I be the alphabet of G .

New symbol S as starting symbol of G .

Start with a grammar that generates words of the form wDM_1wB where D and M_1 are variable symbols.

The following is a modification of the unrestricted grammar for generating words of the form ww for $w \in \{a, b\}^*$. (we'll use 0 and 1 instead of a and b.)

$$S \rightarrow XDMYT$$

$$T \rightarrow R$$

$$YR \rightarrow RY$$

$$0R \rightarrow R0$$

$$1R \rightarrow R1$$

$$XR \rightarrow 0XA \mid 1XB$$

$$\begin{array}{rcl}
 T & \longrightarrow & R \\
 YR & \longrightarrow & RY \\
 S & \longrightarrow & \cancel{XDMYT} \quad \textcolor{blue}{0}R \longrightarrow R\textcolor{blue}{0} \\
 & & \textcolor{blue}{1}R \longrightarrow R\textcolor{blue}{1} \\
 & & XR \longrightarrow \textcolor{blue}{0}XA \mid \textcolor{blue}{1}XB
 \end{array}$$

(original grammar had $T \rightarrow F$ (not FB) and $XF \rightarrow \epsilon$ (not DM_1))

$$\begin{array}{rcl}
 A_0 & \longrightarrow & \textcolor{blue}{0}A \\
 A_1 & \longrightarrow & \textcolor{blue}{1}A \\
 B_0 & \longrightarrow & \textcolor{blue}{0}B \\
 B_1 & \longrightarrow & \textcolor{blue}{1}B \\
 AY & \longrightarrow & \textcolor{blue}{0}YT \\
 BY & \longrightarrow & \textcolor{blue}{1}YT
 \end{array}
 \quad
 \begin{array}{rcl}
 T & \longrightarrow & FB \\
 YF & \longrightarrow & F \\
 \textcolor{blue}{0}F & \longrightarrow & F\textcolor{blue}{0} \\
 \textcolor{blue}{1}F & \longrightarrow & F\textcolor{blue}{1} \\
 XF & \longrightarrow & DM_1
 \end{array}$$

idea : $S \Rightarrow^* wXwYT \quad w \in \{\textcolor{blue}{0}, \textcolor{blue}{1}\}^*$

use $T \rightarrow R$ to extend w

use $T \rightarrow FB$ to stop extending w

So far we have

$$S \implies^* wDM_1wB \text{ for all } w \in \{0, 1\}^*$$

Next, interpret sequence of symbols as state of a Turing machine computation. M_q represents TM in state q (where 1 is the initial state), B represents an infinite sequence of blanks.

The first w is a copy of the input, separated from the computation using D . Introduce grammar rules to simulate transitions. We want

$$M \text{ accepts } w \iff DM_1wB \Rightarrow^* \epsilon$$

and furthermore, if M does not accept w , DM_1wB does not derive any word in I^* .

Further rules of G will simulate TM computation.

G gets variable M_i for each state i of M

For each symbol of the tape alphabet A , give G a variable that represents that symbol. For $x \in A$ let V_x be a variable that represents x .

\overline{B} denotes a blank tape symbol

B denotes an infinite sequence of blanks.

Rules of G : a string derivable from S should represent a state of the TM computation, ie. a tape with a sequence of symbols and the TM located somewhere on that tape.

Example: suppose that M got input 1110 and reached a point later on with a tape $10001\bar{b}1\overline{bbb}\dots$ where M is located at the 3rd tape square in state q_2 , the following string should be derivable from S :

$1110D10M_2001\bar{B}1B$

note we can derive

$1110DM_11110B$

corresponding to starting configuration of M .

Transitions of M become productions of G :

transition: $\delta(Q_1, 0) = (Q_2, 1, R)$ becomes production:

$$M_1 0 \longrightarrow 1 M_2$$

transition: $\delta(Q_1, 0) = (Q_2, 1, L)$ becomes productions:

$$\text{for all } x \in A, \quad x M_1 0 \longrightarrow M_2 x 1$$

$\delta(Q_1, \bar{b}) = (Q_2, 0, R)$ becomes productions:

$$M_1 \bar{B} \longrightarrow 0 M_2$$

$$M_1 B \longrightarrow 0 M_2 B$$

Example: starting configuration of T will have T in state Q_1 at LHS of tape. Suppose input is the word [10011](#).

Derivation:

Finally, we want to make the grammar leave behind the input word w provided that M accepts w .

Given accepting state t and $a \in A$ for which there is no transition in M , add grammar rule

$$M_t a \longrightarrow F'$$

F' is a symbol that “mops up” other symbols – use rules of the form:

$$F' \alpha \longrightarrow F' \quad \text{and} \quad \alpha F' \longrightarrow F'$$

where α is any symbol other than D . One more rule:

$$D F' \longrightarrow \epsilon$$

Going the other way: how to convert a grammar into an “equivalent” Turing machine?

Easiest to use a nondeterministic TM.

General idea: for $n = 1, 2, 3 \dots$ program the TM to generate all words derivable from S using n steps of derivation. If the TM finds word w , it halts and accepts. Otherwise, keep trying! The tape is infinite, so the machine can store the set of all words derivable after n steps on the tape. Lots of implementation detail has been omitted.

It is undecidable to figure out whether a given word is derivable from a given unrestricted grammar. We have essentially reduced from the problem of recognising L_{accept} .

Recall “context-sensitive grammar” – the restriction is that RHS of a rule must be at least as long as LHS.

We can verify that context-sensitive grammars are more restricted than context-free grammars in terms of what languages they can represent, by showing that the problem of determining whether a word w is derivable from a given CSG G is decidable.

Deciding whether w is derivable using (context-sensitive) G :

For $n = 1, 2, 3\dots$ let S_n be the set of all words of length $\leq n$ that are derivable using G .

finding S_n : any word w in S_n must have a derivation of length at most $|V \cup A|^n$ steps. A longer derivation is possible but would contain a repetition

$$S \Rightarrow^* w' \Rightarrow^* w' \Rightarrow^* w$$

since all intermediate words w' have length at most n .

So, there's a limit to how many steps of derivation we need to try out to see if a given word is derivable.

CSGs and automata

Context-sensitive grammars correspond to the class of “linearly bounded Turing machines” - this is a nondeterministic TM with a linear bound on the number of tape squares that may be used during computation.

For given input length n , if kn is the upper bound on how many squares may be used, we have a bound on the total number of configurations reachable during computation. Hence the acceptance problem can be seen to be decidable.

Alphabet A,

$$A^* = \{\text{all strings consisting symbols from } A\}$$

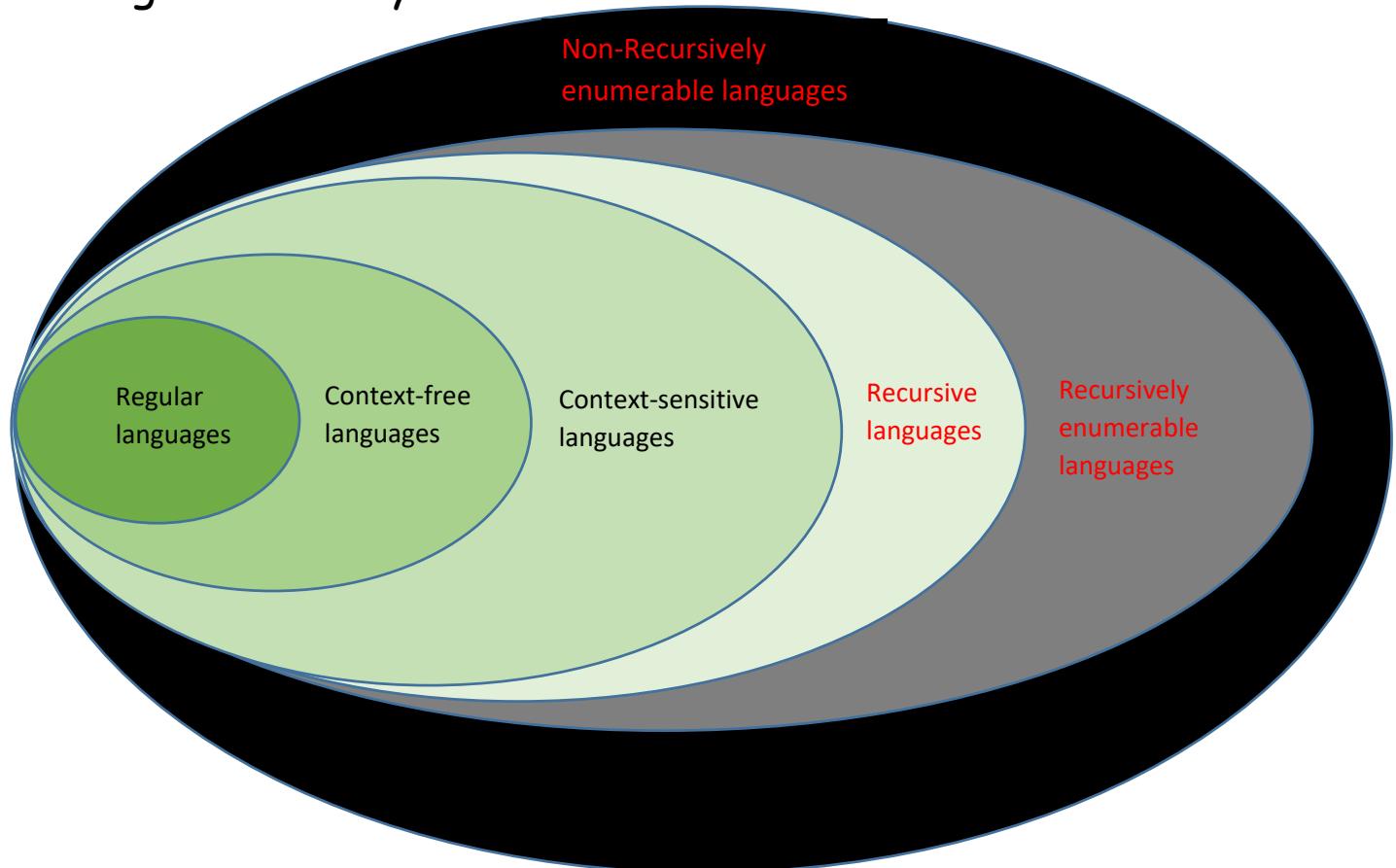
The subsets of A^* are called languages

Question: For a language L, does exist an algorithm to check for any x, if x is in L?

We have identified a class of

- regular languages
- Context-free languages, eg: $\{a^n b^n \mid n \geq 1\}$
- Context-sensitive languages, eg: $\{a^n b^n c^n \mid n \geq 1\}$

Membership problem for these languages can be solved algorithmically.



Recursively Enumerable Languages

A language L is said to be recursively enumerable if there exists a Turing machine that accepts it. That is, there exists a Turing machine M , such that, for every $w \in L$

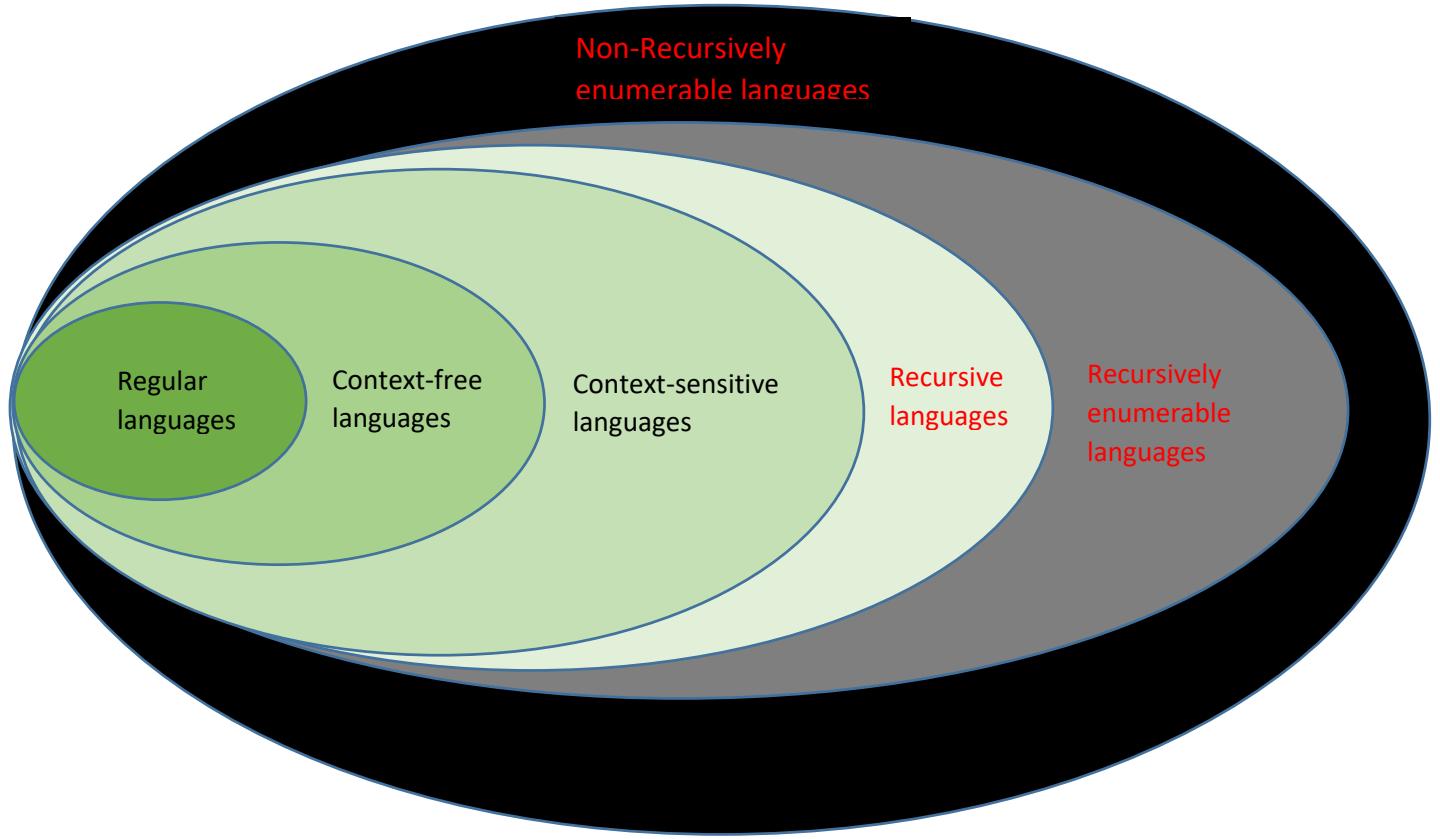
$$q_0 w \vdash^* x_1 q_f x_2$$

where q_0, q_f are initial state and accepting state respectively

Recursive Languages

A language L over A is said to be recursive if there exists a Turing machine that accepts L and that halts on every w in A^+ .

In other words, a language is recursive if and only if there exists a Turing machine (algorithm) to check its membership.



Decidability and Undecidability

“Decidable” is a synonym for “recursive.” We tend to refer to languages as “recursive” and problems (which are languages interpreted as a question) as “decidable”.

If a language is not recursive, then we call the problem expressed by that language “undecidable” talk about a problem being decidable or undecidable.

More formally:

A language is said to be undecidable if there is no TM that

- Accepts that language, and
- Always halts and rejects any string that is not in the language.

Next:

- Construct an undecidable language.
- Show that it really is undecidable
- This shows that “recursively enumerable” is not the same as “recursive”
- Look at various other undecidable languages - technique of reduction allows us to use a previously-obtained undecidability result to deduce that some new language is undecidable.

An undecidable language

The general idea (which we are about to do in detail): Choose a sensible way to encode Turing machines as strings of symbols over a fixed alphabet. The language we construct will consist of encodings of TMs that halt on their input.

Define a “**universal**” **Turing machine** that accepts this language — it executes (or simulates) the encoded TM on some input.

(Analogy with various programs being executable on the same computer)

Universal Turing Machines

A universal TM takes input representing (or encoding) any TM together with an additional input, and simulates the represented TM on the given input.

Each transition of the encoded machine will correspond to a sequence of transitions of the “universal” machine, the one that accepts encodings of machines that halt.

Notation: L_{halt} denotes representations of TMs/inputs where the represented TM halts given that input.

A universal TM accepts members of L_{halt} but may loop forever on non-members.

Undecidability of L_{halt}

The purpose of the universal TM will be to show that L_{halt} is accepted by a Turing machine. After that, we show that L_{halt} is *not* accepted by any TM that always halts on non-members of its language.

This shows a distinction between TMs that are guaranteed to halt, and TMs that aren't.

(Later, we can construct a formal language that is even “worse” than L_{halt} , in that no TM can recognise it, or its complement.)

Defining L_{halt}

Given TM with alphabet $\{0, 1, \overline{b}\}$, encode it using alphabet $\{0, 1, B, *\}$ as follows

States are encoded as members of $\{0, 1\}^k$

initial state: 0^k

single accepting state: 1^k

Encode direction L as 0 , R as 1

Encode transitions $\delta(p, x) = (q, y, D)$
as concatenation of strings for p, x, q, y, D

List the transitions, separated by $*$

Example of encoding

M_{pred} computes predecessor of input

1. $\delta(i, 0) = (i, 0, R)$
2. $\delta(i, 1) = (i, 1, R)$
3. $\delta(i, \bar{b}) = (p, \bar{b}, L)$
4. $\delta(p, 1) = (t, 0, R)$
5. $\delta(p, 0) = (p, 1, L)$

Encode states i as 00, p as 01, t as 11

0000001*0010011*00B01B0*0111101*0100110

Encoding TM + input

Input word to universal TM is

$$\$0^{k+1} * \langle M \rangle \$ \# w$$

M_{pred} (with input 1011) becomes:

\$000*0000001*0010011*00B01B0*0111101*0100110\$#1011

Simulating the encoded TM

M_u simulates M using everything to right of right-hand $\$$ as working tape for M .

M_u executes a sequence of cycles, where each cycle simulates a single step of M .

1. Read symbol scanned by M

Find $\#$

Store symbol to right of $\#$

Move to left-hand $*$

Print symbol to left of left-hand $*$

2. Find next transition to use

Find sequence of symbols to right of $*$ which matches symbols between initial $\$$ and $*$

Halt if no matching string is found

Accept if LHS of tape is $\$1^k$

3. Fetch the new state and symbol

We have found correct transition to use. Copy the $k + 1$ symbols corresponding to new state and symbol, over to LHS of tape (between initial $\$$ and $*$).

4. Printing the new symbol

Enter state that encodes new symbol and direction to move. Find $\#$ and replace symbol to its right with new symbol.

5. Move the tape head (i.e. M 's pointer to its tape)

If direction (above) is R, switch $\#$ with symbol to its right, else switch $\#$ with symbol to its left. (Unless that's a $\$$, in which case halt and accept/reject depending on state of M)

Preliminary Notation

Countable Set: An infinite set S is said to be countable if there exist a one-to-one mapping between S and the set of positive integers.

In other words, the elements in S can be indexed by 1, 2, 3,

Fact: If S is an infinite countable set, then its powerset $2^S = \{L \mid L \text{ is subset of } S\}$ is not countable.

The set of all Turing machines over an alphabet is countable.

The Halting Problem

Given a TM together with input word, will the resulting computation halt? *Note: we don't care whether it accepts or why it halts*

We know how to encode TMs/inputs as strings

Hence we have associated language recognition problem: Does a word represent a TM + input on which it halts?

If so, the language should be accepted by some halting Turing machine M .

$L_{halt} = \{ \$\langle N \rangle \$u : u \in \{0, 1\}^* \text{ and } N \text{ is a TM which halts on input } u\}$

Note: This is really a result about languages and computation, not Turing machines.

or if you prefer to think in terms of computer programs rather than TMs

Think of a computer program that is supposed to take an initial input and tell you something about it. Given a particular input the program may

- Say that it has some property of interest
- fail to do so
- go into an infinite loop

Can we decide *automatically* when we will get the infinite loop?

You can't just run the program.

But you might be able to tell in advance, eg some infinite loops in programs are obvious.

Entry to table says what happens when any input is given to any program (or, TM)

		input						
		0	1	2	3	4	5	...
program		0	Y	N	?	N	Y	...
		1	Y	?	?	?	...	
		2	Y	Y	Y	Y	...	
		3	N	:				
		4	:					
		:						

Now try to define a “program” that accepts a number n whenever the n th program in our list fails to accept the n th input...

We find that such a program cannot exist.

However, if we could solve the Halting problem we could in fact use the solution to construct such a program as follows:

- Use HP solver to decide whether program n halts on input n
- If not, accept number n .
- If so, simulate program n on input n , and reverse the answer given.

L_{halt} is not recursive: proof in terms of TMs

Prove by contradiction: Take the imaginary halting TM M that accepts L_{halt} ; we will use M to construct an input (to itself) that leads to a contradiction.

Construct M' from M : given any input, M' behaves exactly like M , but if M accepts, M' is designed to loop for ever instead.

Define machine M'' as follows.

M'' with binary input string v starts by replacing it with $v\$v$ and then behaves like M' on the “input” $v\$v$.

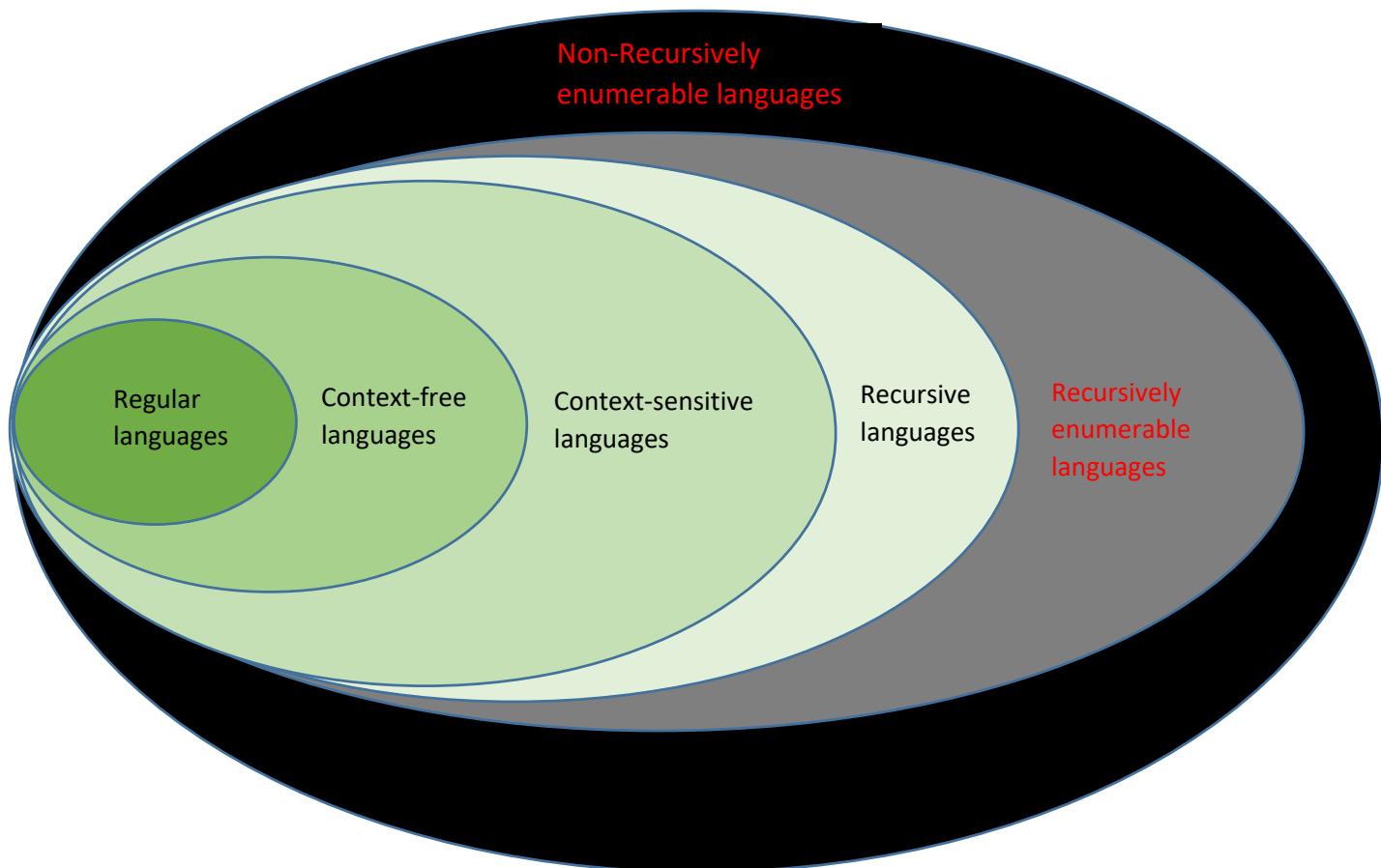
So if a Turing machine M really exists (as specified) then it should be possible to construct M'' from it.

M'' halts on input $\langle M'' \rangle \iff M'$ rejects $\langle M'' \rangle \$ \langle M'' \rangle$

LHS says $\langle M'' \rangle \$ \langle M'' \rangle \in L_{halt}$

RHS says $\langle M'' \rangle \$ \langle M'' \rangle \notin L_{halt}$

Contradiction!



Reductions

We can use the undecidability of the halting problem to show that various other problems are undecidable.

A *reduction* from L_1 to L_2 is a TM-computable function f such that

$$\forall w \in L_1 \quad f(w) \in L_2$$

$$\forall w \notin L_1 \quad f(w) \notin L_2$$

If L_2 is recursive then L_1 is recursive

equivalently:

If L_1 is not recursive then L_2 is not recursive.

Uniform Halting Problem

Given a TM, does it halt on *all* inputs?

$$L_{uhalt} = \{\langle M \rangle : M \text{ halts on all inputs}\}$$

Find reduction L_{halt} to L_{uhalt}

Reduction to uniform H.P.

Given $\langle M \rangle \$ w$, construct M' as follows.

M' first of all writes word w on tape (use $|w|$ states to do this).

Then M' deletes every non-blank symbol to the right of w (two more states)

Then M' returns to LHS of input tape (another $|w|$ states can be used)

Then M' imitates M (using however many states M had).

The construction is TM-computable, and constructs M' that accepts all its inputs *if and only if* M accepts w .

Acceptance Problem

Define L_{accept} to be the language

$$L_{accept} = \{\langle M \rangle \$ w : M \text{ accepts input } w\}$$

This problem is also undecidable. Reduce from L_{halt}

Given $\langle M \rangle \$ w$, construct $\langle M' \rangle \$ w$ as follows. We want M' to accept input w if and only if M halts on input w .

So, general idea is: modify M such that if it halts, it must accept.

Introduce new accepting state t . For all undefined transitions of M (which would lead to halt and reject) introduce a transition to t . In any accepting state of M , transfer automatically to t .

Finally we need to catch the case when M moves off LHS of tape. Make M' place a new character at LHS initially and copy input 1 square to right, then come back to 2nd square of tape and simulate M , accepting if that new character is ever scanned.

notation: Given language L over alphabet A , its complement $A^* \setminus L$ is denoted \bar{L} .

Fact: L is recursive iff \bar{L} is recursive.

(easy to prove)

Definition: Language L is *co-recursively enumerable* provided that its complement \bar{L} is recursively enumerable.

Fact: Language L is recursive iff both L and \bar{L} are recursively enumerable.

from which we can deduce that L_{halt} is not co-recursively enumerable.

Proof that L is recursive iff L, \bar{L} are r.e:

Easy to prove that L is recursive $\Rightarrow L, \bar{L}$ are r.e.

Prove that L is recursive $\Leftarrow L, \bar{L}$ are r.e:

There exist TMs T_1 and T_2 that accept L and \bar{L} respectively.

Given an input word w , run T_1 and T_2 in parallel - as soon as one or the other accepts, then w can be accepted or rejected as appropriate.

Observe that L_{halt} is r.e. but not co-r.e.

Also $\overline{L_{halt}}$ is co-r.e. but not r.e.

Question: Is any language neither r.e. nor co-r.e.?

Define L_{inf} to be

$\{\langle M \rangle : M \text{ halts on infinitely many distinct inputs}\}$

L_{inf} is neither r.e. nor co-r.e.

Prove this by reducing L_{halt} to L_{inf} and to $\overline{L_{inf}}$.

$L_{halt} \leq L_{inf}$:

Convert $\langle M \rangle \$ w$ to M' where M' writes w on tape, then goes to LHS and simulates M .

$L_{halt} \leq \overline{L_{inf}}$:

Let M'' be a TM which scans input word left-to-right and accepts.

Construct M' to simulate M'' and M in parallel, and if M'' halts (and accepts) first then M' should halt, but if M halts first, then M' should go into an infinite loop.

M' halts for all inputs whose length is shorter than the computation of M on input w . (Infinitely many of those iff M fails to halt.)