Introduction to the course topics

The course covers a variety of topics dealing with finite state machines, regular expressions, grammars and program verification, as well as, an introduction to the theoretical limits of algorithmic computation.

The relationship between programs and formal languages provides an example of the impact of theory on practice. Uses of formal theory include the following:

- grammars are used in lexical and parsing stages of compiler construction
- use of regular expressions in text editors
- state-charts in object-oriented modeling
- circuit-design
- DNA and protein sequence matching: regular expression matching algorithms

On the other hand, theory acts also as an "early warning system" by providing a science of the impossible:

- what should not be attempted because it is impossible (or provably too costly)

A fundamental question in computing is whether there exist tasks/problems that, even in principle, cannot be solved algorithmically and, if yes, which tasks are algorithmically solvable and which are not. In fact, it can be established that the number of different computational problems is *larger* than the number of all possible programs (in some programming language such as Java or C), which means that there must exist problems that are not solvable by any program (or by any algorithm).

Note that the number of programs is infinite, and to show that the number of computational problems is larger, we need to compare the *sizes* of different infinite numbers.

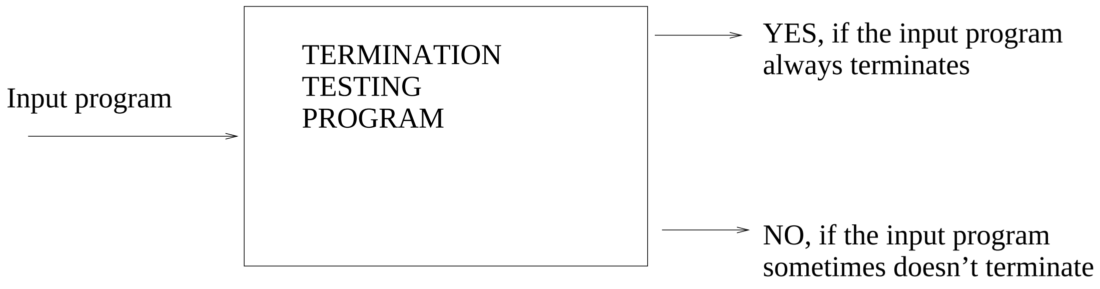
Input program
TERMINATION TESTING PROGRAM
YES, if the input program always terminates
NO, if the input program sometimes doesn't terminate

Figure 1: Example of an uncomputable problem.

In this course we use a different approach. Using a technique called *diagonalization* we establish that certain specific (and "useful") computational problems cannot be solved by any program written in the language C. The most well-known example of an unsolvable problem is the so called *halting problem* that asks whether an arbitrary program given as input terminates.

Example. A program with behavior as depicted in Figure 1 does not exist!

However, having an algorithm A for a computational problem P does not mean that P is solvable *in practice.* It might be the case that for inputs of moderate size A would need more time than the age of the universe, even when executed on modern computers ...

These observations can be summarized as a coarse classification of algorithmic problems/functions:

1. Non-computable problems (that is, even in principle impossible to solve using an algorithm/a computer)

2. Possible–with–unlimited–resources BUT impossible–with–limited–resources

3. Possible–with–limited–resources: these are the problems "solvable in practice"

Typical questions we can consider:

- Program existence: Does there exist a program for a given algorithmic problem (or function)?
- Software specification: How should programs be specified?
- Software validation: Is a given program correct, i.e., does it satisfy the specification?
- Software construction: How is a correct program obtained?
- Semantics: What does a given program do? (this is related to correctness)
- Efficiency: Is there a more efficient (faster) program for the given problem?
- Hardware comparison: Is one machine more powerful than another one?

Given a programming problem it is easier to convince someone that there is a program which solves the problem (if a program exists) than to convince someone that there is *no* program for the problem (if a program does not exist). In the former case it is sufficient to give the program and, in fact, usually it is sufficient to just outline the solution informally, or in pseudo code (if the purpose is just to convince the reader that a program exists).

Example. A program to compute the function $f(n) = n^2$.

On the other hand, if we want to show that a program for the given problem does *not* exist we need to show that none of the infinitely many possible programs solves the given problem (or computes a given function).

In order to be able to deal with negative results of this kind, we need to be precise about what constitutes a legal program! (or a legal algorithm)

Using a more practical perspective, a problem may be "uncomputable" also due to other types of reasons, for example, predicting the weather for a month in advance is impossible because the required input would be "too large" to be collected.

Instead of considering general algorithms[1], in the course we begin here with a simpler problem:

- test whether arbitrary input strings (= sequences of symbols) can be matched by a given pattern.

Special notation and implementation techniques have been developed to specify and recognize such *patterns:*

- state-transition diagrams (finite automata): simple simulated machines
- regular expressions: rules for building patterns
- grammars: rules for generating patterns

Alphabets, strings and languages

This material is from Chapter 7 in the textbook.

We begin with an introduction to basic concepts of formal languages.

- An alphabet is a finite, nonempty set of elements. The elements of the alphabet are called symbols (or tokens, characters).
- A string over an alphabet Σ is a finite sequence of symbols of Σ. (Strings are sometimes called also *words.*)
- A language over Σ is a set of strings over Σ.

Examples.

[1]The general limits of algorithmic computability will be discussed in the last part of the course.

1. English alphabet $\{a, b, c, d, \ldots, z\}$

   Strings: cat, dog, mouse, xzrbstuph, ...

   Language: the set of all correct English sentences

   — not precisely defined ...

2. Alphabet: $\{a, b\}$

   Strings: $\varepsilon, a, b, ab, ba, aa, bb, aaa, \ldots$

   Language: $\{aaa, aab, aba, baa, abb, bab, bba, bbb\}$ - a finite language

   Another language over the same alphabet $\{a^i b^i \mid i \geq 0\} = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$
   This is an infinite language.

3. Alphabet: Java reserved words and identifiers

   Example string: a Java program

   Language: the set of all syntactically correct Java programs

We use the following notions and definitions concerning strings:

- The <u>empty string</u> is denoted ε.
  ε is a string over any alphabet.

- The <u>length</u> of a string is the number of occurrences of symbols in it. The length of a string s is denoted $|s|$.

  Examples:

  - The length of ε is 0, that is, $|\varepsilon| = 0$.
  - The length of the string $bccb$ is 4, that is, $|bccb| = 4$.

- The <u>concatenation</u> of strings x and y is denoted $x \cdot y$, or xy for short. It is the string obtained by appending y to x.

Examples: If $x = abc$ and $y = de$, then $xy = abcde$ and $yx = deabc$.

As seen from the example, the concatenation operation is not commutative. Note also that ε acts as an identity for string concatenation: $x \cdot \varepsilon = \varepsilon \cdot x = x$ for all strings x, in particular, $\varepsilon \cdot \varepsilon = \varepsilon$.

Since concatenation is associative we do not need to use parentheses:
for all strings x, y, z we have $x(yz) = (xy)z$. How would you prove this?

- If x is a string, x^n denotes the concatenation of n copies of x (the power of a string). Here $n \geq 0$.

  Inductive definition of powers of a string:

  1. $x^0 = \varepsilon$
  2. $x^{i+1} = x \cdot x^i$, for $i \geq 0$.

  Example.

  $(abc)^0 = \varepsilon$

  $(abc)^3 = abcabcabc$

- If $s = xy$, we say that x is a prefix of s and y is a suffix of s. If $s = xyz$, we say that y is a substring of s. Note that here x and/or z may be the empty string.

  Examples:

  1. ab is a prefix of aba. The other prefixes of aba are ε, a and aba.
  2. ba is a suffix of aba
  3. ε is a prefix/suffix/substring of any string
  4. a string is always a prefix/suffix/substring of itself
  5. What are the substrings of $cbcc$?

Formal languages

A formal language has to be precisely defined, the word *formal* refers to the fact that we have a precise set of rules which tell us exactly which strings are in the language, respectively, are not in the language (or an algorithm that answers this question).

- A finite language can (at least in principle) be defined by listing all strings in it.

  Example: $\{00, 01, 10, 11\}$

- Infinite languages can be defined by giving some condition that exactly characterizes the strings in the language.

  Examples.

  $\{0^n \mid n \geq 0\}$

  $\{0^n 1^n \mid n \geq 1\}$

  $\{0^n 1^m \mid \sqrt{n} \leq m \leq n\}$

Note that $\emptyset$ (the empty set) is a language over any alphabet Σ. Also $\{\varepsilon\}$ (the language having only the string ε) is a language over any alphabet Σ. It is important to remember that $\emptyset \neq \{\varepsilon\}$. Why?

As indicated in the above example, languages can be speficied by "set notation". The problem with set specifications is that the notation is too powerful and, if the conditions can be specified using first-order logic, membership in the language will be an unsolvable problem.

In the first part of the course we want to consider language specification tools that allow the development of efficient algorithms for the language. The first idea is that we can define new languages from "simpler" ones using operations on languages. Three important operations we consider are

- union
- concatenation
- closure

Later we will see that all regular languages can be built from elements of Σ, the empty string ε and the empty set $\emptyset$ using these operations.

Union

If R and S are languages over Σ, their union is denoted $R + S$. It consists of all strings that are in R *or* in S. (Thus $R + S$ is just a different notation for the union of sets, $R \cup$S.)

Concatenation

If R and S are languages, their concatenation is defined as

$$R \cdot S = \{rs \mid r \in R, s \in S\},$$

usually written simply as RS.

Examples.

If $R = \{a, ab\}$, $S = \{bc, c\}$, what is their concatenation RS? Note: the concatenation consists of 3 different strings.

If $R_1 = \{a, \varepsilon\}$ and $S_1 = \{ab, b\}$, what is $R_1 S_1$?

Here it is useful to consider how the empty set and the set consisting of the empty string behave with respect to set concatenation. What are the following languages:

$\emptyset \cdot R = \ldots\ldots?$

$\{\varepsilon\} \cdot R = \ldots\ldots?$

$\{\varepsilon\} \cdot \emptyset = \ldots\ldots?$

When union or concatenation is applied to finite languages, the result is always finite. The third operation allows us to specify infinite languages.

Closure of languages

The set of all strings over alphabet Σ is denoted Σ^*. The closure operation can be extended for any language S:

$$\begin{aligned} S^* &= \{s_1 \cdot \ldots \cdot s_n \mid s_i \in S, i = 1, \ldots, n, n \geq 0\} \\ &= \{\varepsilon\} + S + S^2 + S^3 + \ldots \end{aligned}$$

Example. Let $S = \{01, 1\}$. Then

$S^0 = \{\varepsilon\}$

$S^1 = S = \{01, 1\}$

$S^2 = \{0101, 011, 101, 11\}$

$S^* = \{\varepsilon, 1, 01, 11, 011, 101, 111, 0101, \ldots\}$

Here S^* is an infinite language and we cannot explicitly write down all strings of S^*.

Example. $\emptyset^* = \{\varepsilon\}$. This can be verified using the definition of the closure operator.

We denote also

$$\begin{aligned} S^+ &= \{s_1 \cdot \ldots \cdot s_n \mid s_i \in S, i = 1, \ldots, n, n \geq 1\} \\ &= S + S^2 + S^3 + \ldots \end{aligned}$$

Note that $S^* = S^+ + \{\varepsilon\}$ for any language S.

Now we come to a definition that will be central in the first part of the course.

Definition. A language S over alphabet Σ is said to be regular if S can be defined from elements of Σ and $\emptyset$ using the operations *union, concatenation* and *closure.* The description of the language S in this form is called a regular expression for S.

- Note that, as observed before, $\emptyset^* = \{\varepsilon\}$. Sometimes the definition of regular languages includes ε as one of the "basic building blocks", but ε can be produced using the closure operator.
- In particular, all finite languages are regular. Why?

As we will see, the regular languages have "nice" properties and can be easily implemented. However, regular languages form only a "small" family of languages and we will develop techniques for showing that a language is not regular.

Application: *Sequence matching problem*

Each DNA molecule is composed of two strands that are made up of a sequence of nucleotides. Each nucleotide has one of four bases, represented by symbols A, T, C, G (and other parts). Proteins are large molecules that are composed of a sequence of amino acids. There are 20 amino acids that occur in proteins, denoted by standard one-letter symbols. In this way, DNA or protein molecules can be represented as strings. Analyzing DNA or protein sequences can help to determine which function they perform, or what parts of the sequence are important for a particular function. Comparing different DNA sequences tells us which organisms are related.

Related sequences are not necessarily identical. When comparing DNA or protein sequences we want to *align* them in a way that minimizes some type of distance between the sequences.

Regular expressions are used to specify *patterns* that describe a related set of strings (sequences). In this way we can compare the pattern to an individual sequence or to a database of sequences to find good matches. The applications often use *extended regular* *expressions* that allow operations other than union, concatenation and closure. (Examples in class.)

The tools used to solve sequence matching problems typically involve also *finite state machines* that are discussed in out next topic. The BLAST family of search engines use heuristic techniques to build a large *deterministic finite automaton* that, for a given query string, finds from a database of known sequences the most closely related ones.

State-transition diagrams

We introduce simple simulated machines that turn out to model exactly the class of regular languages. Before going to the formal definition we consider a couple of examples. This material is from Chapter 8 in the textbook.

Example. An identifier can be defined as a string of letters and digits that begins with a letter. The states are depicted by circles and the transitions between states are indicated by arrows labeled by the corresponding tokens. A final state (or accepting state) is a double circle. The start state is indicated by an incoming arrow.

- Token **letter** stands for any of the symbols a, . . . , z, A, . . . , Z.
- Token **digit** stands for 0, 1, . . . , 9.

The set of identifiers is specified by the state-transition diagram in Figure 1.

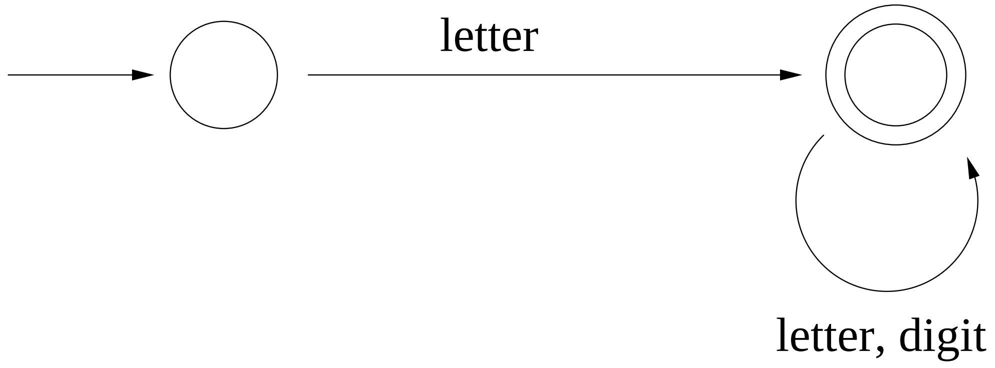

Figure 1: A state-diagram for specifying identifiers.

Example. We design a “sequential lock” as described below. The lock has 1-bit sequential input. Initially, the lock is closed. If the lock is closed it will open when the last three input bits are “1”, “0”, “1”, and then remains open.

In other words, the state-transition diagram should accept exactly all strings that contain substring 101. The state-transition diagram will be constructed in class.

What is a regular expression that denotes the same language?

A state diagram describes a **deterministic finite automaton** (DFA), a machine that at any given time is in one of finitely many states, and whose state changes according to a predetermined way in response to a sequence of input symbols.

Formally a deterministic finite automaton is defined as a five-tuple and the definition is below.

Definition. A DFA is defined as a tuple

$$M = (Q, \Sigma, \delta, s, F)$$

where the components are as follows:

- Q is the finite nonempty set of states
- Σ is the input alphabet
- $\delta : Q \times \Sigma \longrightarrow Q$ is the transition function
- $s \in Q$ is the starting state
- $F \subseteq Q$ is the set of accepting states

A finite state automaton (DFA) is conveniently depicted using a state diagram, that is, a directed graph. Especially when the set of states is small the operation of the machine is intuitively easier to understand based on a graphical representation. On the other hand, when we want to implement a DFA we use notation that lists all transitions, or a *transition table* as illustrated in the next example.

Example. Consider the state diagram in Figure 2. Note that now we have added names for the states (q_i, $i = 1, 2, 3, 4$). The state names are needed to write down explicitly the transition table (but the state names do not affect the operation of the state diagram).

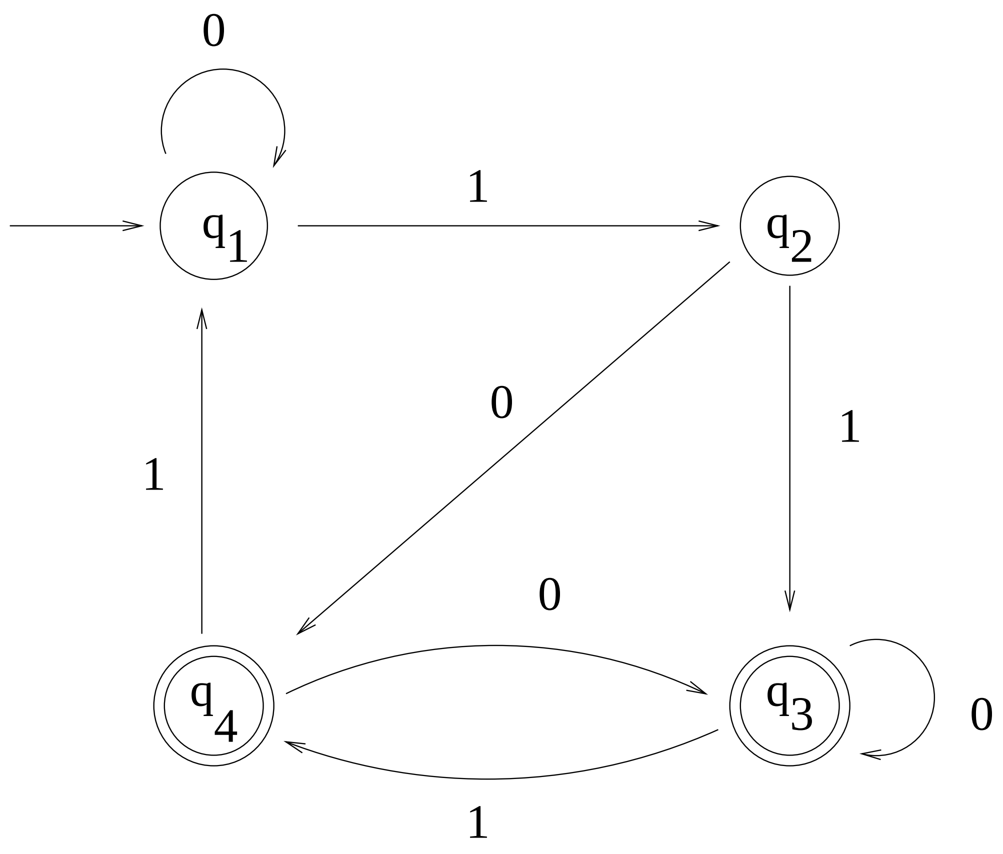

Figure 2: A state-transition diagram.

In a formal notation this automaton can be specified as

$$M = (Q, \Sigma, \delta, s, F)$$

where

- the set of states is $Q = \{q_1, q_2, q_3, q_4\}$,
- the input alphabet is $\Sigma = \{0, 1\}$,
- the starting state is q_0,
- the set of accepting states is $\{q_3, q_4\}$, and,
- the transition function $\delta : Q \times \Sigma \longrightarrow Q$ is given by the below transition table:

Current state/input	0	1
q_1	q_1	q_2
q_2	q_4	q_3
q_3	q_3	q_4
q_4	q_3	q_1

In a state diagram the starting state is denoted by a circle with an "incoming arrow" and an accepting state is denoted by a double circle.
Note: A state diagram has only one starting state. There can be more than one accepting states (or no accepting states).

When a DFA is implemented a graphical state diagram notation is not sufficient. One needs to, basically, give names for the states and encode the transition table in the code, see the examples in section 8.2 of the textbook.

Definition. A state diagram (or DFA) *accepts* a string

$$c_1 c_2 \cdot \ldots \cdot c_n$$

if there is a path from the starting state to an accepting state that is labeled by symbols $c_1, \ldots, c_n$. The language *recognized* by the state diagram (or DFA) consists of all strings accepted by it.

A DFA accepts exactly the strings that label a path from the start state to an accepting state.

Example. Construct a state diagram for recognizing comments that may go over several lines:
`/* ......*/`

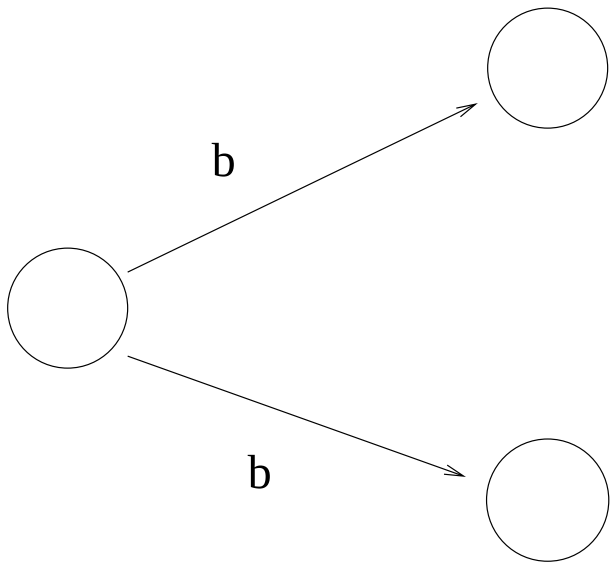

Figure 3: Nondeterministic transitions: a state has two outgoing transition labeled by the same input symbol.

Nondeterminism

- The state diagrams that we have considered up to now were *deterministic:* for any state and input symbol pair there can be at most one outgoing transition.
- A *nondeterministic* state diagram allows the following type of situations as depicted in Figure 3:

In a DFA, a given string labels a unique path of transitions beginning from the start state. When dealing with nondeterministic state diagrams it is important to remember that now a given string may label more than one path beginning from the starting state and we have to be careful how the acceptance of a string is defined:

Definition. *Nondeterministic acceptance:* a string is accepted if it appears on **any** path from the starting state to an accepting state.

Why nondeterminism?

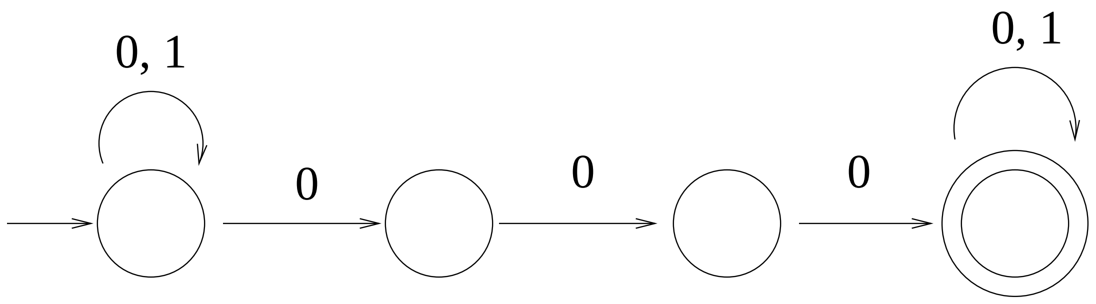

Figure 4: A nondeterministic state diagram.

- Often it is much easier to construct a nondeterministic state diagram than a deterministic one. Example will be discussed in class.
- A nondeterministic state diagram can be *much* smaller (i.e., have fewer states) than the smallest possible deterministic state diagram that recognizes the same language.
- On the other hand, it is not immediately clear how we can implement nondeterministic state diagrams since program behavior is deterministic (or, at least, it *should* be deterministic). Fortunately, there is a way to convert a nondeterministic state diagram into an equivalent deterministic one, and the conversion can even be automated.

  Example. Consider the following nondeterministic state diagram with input alphabet $\{0, 1\}$:

  - Where does the nondeterminism appear in the state diagram of Figure 4?
  - What is the language recognized by the state diagram of Figure 4?
  - How would you construct an equivalent DFA? A direct construction of a DFA is, perhaps, not immediate.

Nondeterministic state diagrams are also called *nondeterministic finite automata,* or NFA. An NFA can be implemented as a DFA using the so called *subset construction.* The subset construction is explained on pages 179–181 in the textbook. We illustrate (to be done

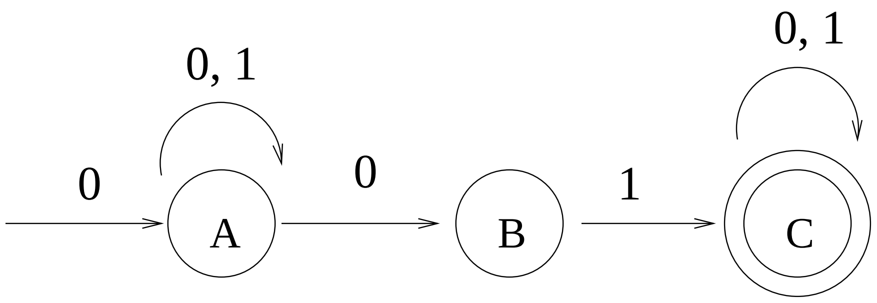

Figure 5: A nondeterministic state diagram.

in class) the subset construction by applying it to the simple NFA depicted in Figure 5. The NFA has only 3 states – for larger NFAs the resulting DFA obtained from the subset construction can be large and not convenient to wrrite down "by hand".

Example. Consider the NFA depicted in Figure 5.

We construct an equivalent DFA using the subset construction. The DFA keeps track of the set of *all* possible states that the NFA may be in after reading the current sequence of input symbols, that is, states of the DFA will be labeled by subset of $A, B, C\}$.

The starting state is $\{A\}$. A set of states is accepting if it contains at least one accepting state of the original NFA (in this case the state C). The transitions of the DFA are defined by following all possible nondeterministic transitions from the current state (as explained on pages 179–181 in the textbook). The transition table and the state diagram of the DFA that is obtained by applying the subset construction to the NFA of Figure 5 will be done in class. *Question:* How could you simplify the resulting deterministic state diagram?

ε-transitions

Sometimes it is convenient to allow in nondeterministic state diagrams transitions that do not read an input token ("spontaneous" transitions). These are called ε-transitions. NFAs with ε-transitions will be needed, in particular, for the algorithm that converts regular expressions to an equivalent state diagram (to be considered next week).

Note: A deterministic state diagram cannot have ε-transitions. A state diagram having ε-transitionsis always nondeterministic. A state diagram with ε-transitions is called an ε-NFA.

ε-transitions make it easy to simulate operations on languages. The following example illustrates constructing a state diagram for $S_1 \cup S_2$ when state diagrams for the components S_1 and S_2 are known.

Example. Let $\Sigma = \{0, 1\}$ and define

$S_1 = \Sigma^* 01 \Sigma^*$

$S_2 = \{w \in \Sigma^* \mid w \text{ has an even number of 1's }\}$

The following state diagram recognizes $S_1 \cup S_2$:

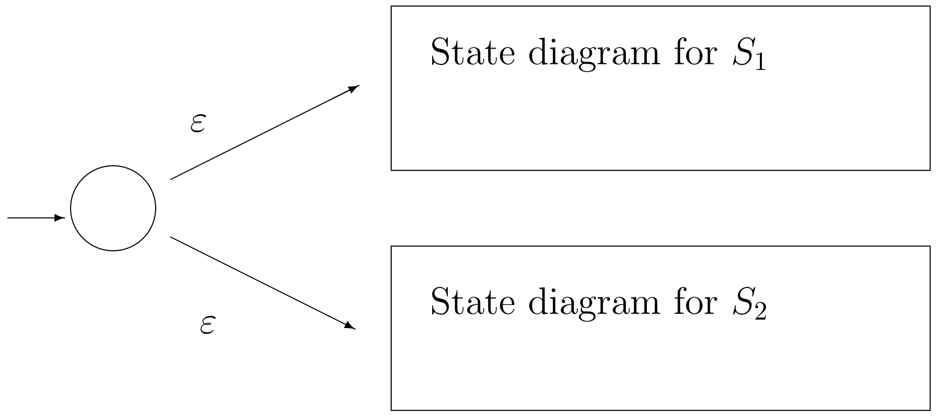

Figure 6: A state diagram for $S_1 \cup S_2$.

Next we consider an algorithm that converts an ε-NFA to an equivalent ordinary NFA without ε-transitions.

Given an ε-NFA M we can construct an equivalent NFA without ε-transitions as follows (see page 183 in the textbook):

1. Make a copy M' of M where the ε-transitions have been removed. Remove states that have only ε-transitions coming in, however, the starting state is not removed.

2. Add transitions to M' as follows: whenever M has a chain of ε-transitions followed by a "real" transition on $x \in \Sigma$, see Figure 7:

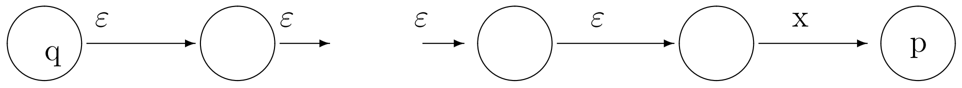

Figure 7: A chain of ε-transitions followed by a "real" transition.

we add to M' a transition from state q to state p that is labeled by x. Note that here q and p may be any states. For example, the above construction step is used also in the case where $q = p$.

3. If M has a chain of ε-transitions from a state r to an accepting state, then r is made to be an accepting state of M'.

Example. How does the construction work with the state diagram:

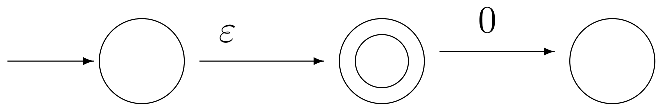

Figure 8: An ε-NFA.

Example. The state diagram depicted in Figure 9 recognizes unsigned decimal numbers: We construct an equivalent nondeterministic state diagram without ε-transitions – to be done in class.

Note: By combining the above two transformations, that is, the elimination of ε-transitions and the subset construction, we can always convert a state diagram with ε-transitions to a

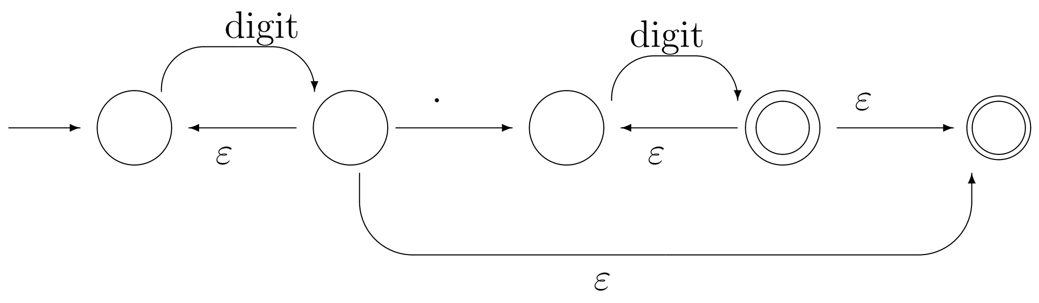

Figure 9: An ε-NFA for unsigned decimal numbers.

deterministic state diagram. This will form the basis for the algorithm for converting regular expressions to state diagrams.

Applications: Natural language processing

Noam Chomsky, one of the pioneers of language theory, showed already in the 50's that English (or any natural language) is not a finite-state language. Natural languages contain unlimited nested dependencies, and it is impossible to construct a finite automaton that keeps track of such dependencies. Next week (in section 9.4) we will develop techniques that allow us to formally prove that a given language cannot be recognized by *any* finite automaton.

For many decades, computational linguistics concentrated on more powerful formalisms, namely on extensions of context-free grammars (that we will study in chapters 10 and 11).

However, finite-state machines have made a comeback in modern natural language processing [1]. It turns out that writing large-scale high-level grammars for languages such as English is very hard. Although English as a whole is not a finite-state language, there are subsets of English for which a finite-state description is quite adequate and much easier to construct than an equivalent (phrase-structure) grammar. Also it was discovered that formal descriptions of phonological alterations used by linguists were, in fact, finite-state models.

Researchers have developed special finite-state formalisms that are suited for the description of linguistic phenomena, and compilers that efficiently produce finite automata from such descriptions [1]. The automata in linguistic applications are much too large and complex to be produced by hand. For example, the transition tables of automata used for text–to–speech translation of languages such as English, French or German typically require 25–30 Mbytes.

References

[1] K.R. Beesley and L. Karttunen: *Finite State Morphology,* CSLI Publications, 2003. Web page for the book: `http://www.fsmbook.com`

Converting regular expressions to state diagrams and vice versa

Up to now we have considered two language specification mechanisms: regular expressions and state diagrams. We show that regular expressions and state diagrams define exactly the same class of languages. This fact is remarkable because superficially state diagrams and regular expressions appear to be quite different.

To show that regular expressions and state diagrams are equivalent we need to do the conversion in both directions. This material is from Chapter 9 in the textbook.

Converting regular expressions to state diagrams

For an arbitrary regular expression we construct an equivalent nondeterministic state diagram with ε-transitions that satisfies the following conditions:

1. There is exactly one accepting state and it is distinct from the start state.
2. There are no transitions into the start state.
3. There are no outgoing transitions from the accepting state.

The recursive construction relies on the fact that the previously constructed state-transition diagrams satisfy the above conditions 1., 2., 3. (In general, an ε-NFA does not need to satisfy the above three conditions.)

The state diagrams for the <u>base cases</u> (i) $E = \emptyset$, (ii) $E = \varepsilon$, (iii) $E = a$, ($a \in \Sigma$) are depicted in Figure 1.

<u>Inductive step:</u> Next suppose that E_i, $i = 1, 2$, are regular expressions and we have constructed a state diagram S_i that is equivalent to E_i and satisfies the conditions 1., 2., 3. An

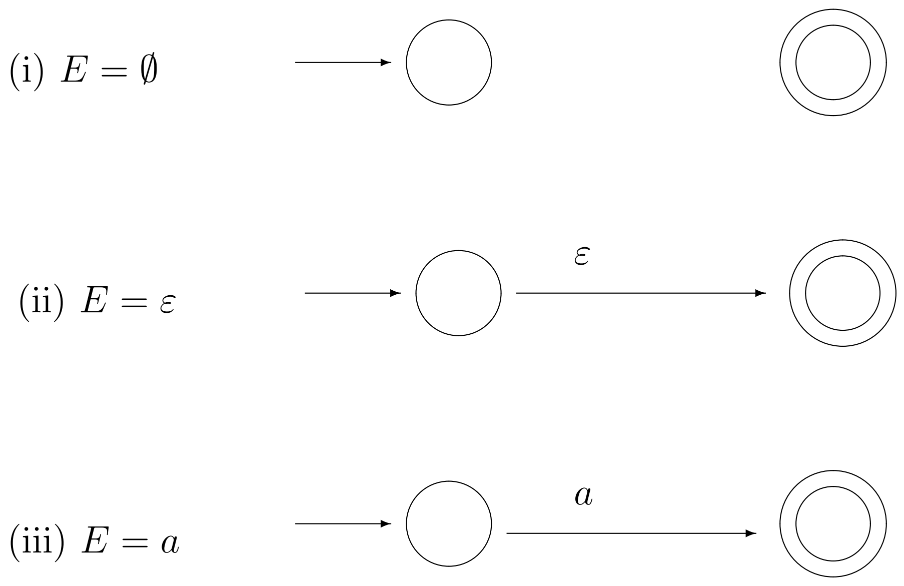

Figure 1: State diagrams for the base cases.

abstract representation of S_i is given in Figure 2: S_i has no incoming transtions to start state, has one accepting state, and no outgoing transitions from the accepting state.

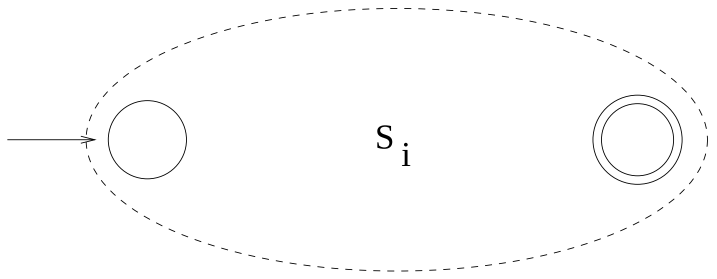

Figure 2: State diagram S_i that is constructed for the regular expression E_i, $i = 1, 2$.

For the inductive step we have to show how to construct state diagrams for the regular expressions $E_1 + E_2$, $E_1 \cdot E_2$ and E_1^*.

(iv) The state diagram for $E_1 + E_2$ is depicted in Figure 3. The resulting state diagram has all states of S_1 and S_2 with the exception that the start states (respectively, the accepting states) of S_1 and S_2 are merged together. The conditions 1., 2., 3. guarantee that the constructed state diagram accepts exactly the strings accepted by S_1 and the strings accepted by S_2 (more details explained in class). Note that the resulting state diagram again satisfied the conditions 1., 2., and 3.

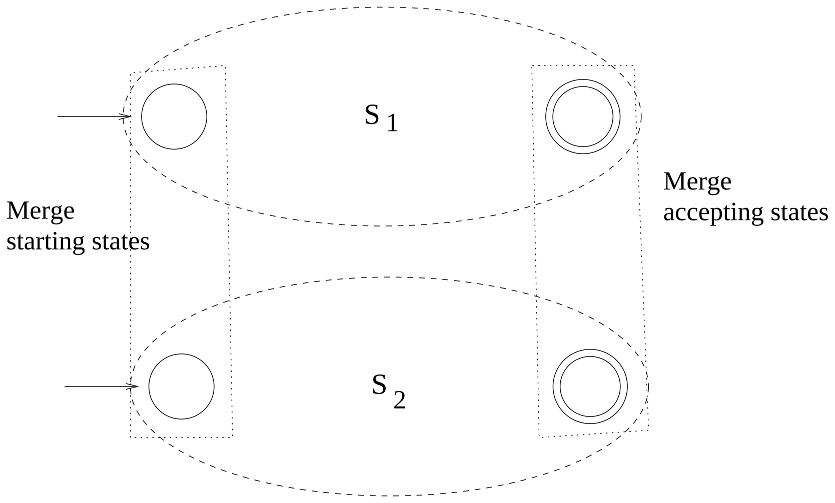

Figure 3: State diagram for regular expression $E_1 + E_2$.

(v) The state diagram for the regular expression $E_1 \cdot E_2$ is constructed by merging the accepting state of S_1 with the start state of S_2 and the merged state is not accepting. The construction is depicted in Figure 4. Again the conditions 1., 2., 3. guarantee that to reach the new accept state from the start state of S_1, the input has to first spell out a string accepted by S_1 followed by a suffix that is accepted by S_2. Note that since the merged state has no outgoing transitions in S_1 and no incoming transition in S_2, the resulting state diagram cannot have "mixed computations" that would first simulate S_1, then simulate S_2, and go back to simulating S_1. (Note that for correctness of this part of the construction only one of the conditions 2. or 3. would be sufficient.)

(vi) To conclude the recursive construction, we have to consider the closure operation. For the regular expression E_1^* we construct from S_1 a state diagram as follows: merge the accepting state of S_1 with the start state of S_1, the merged state is neither an

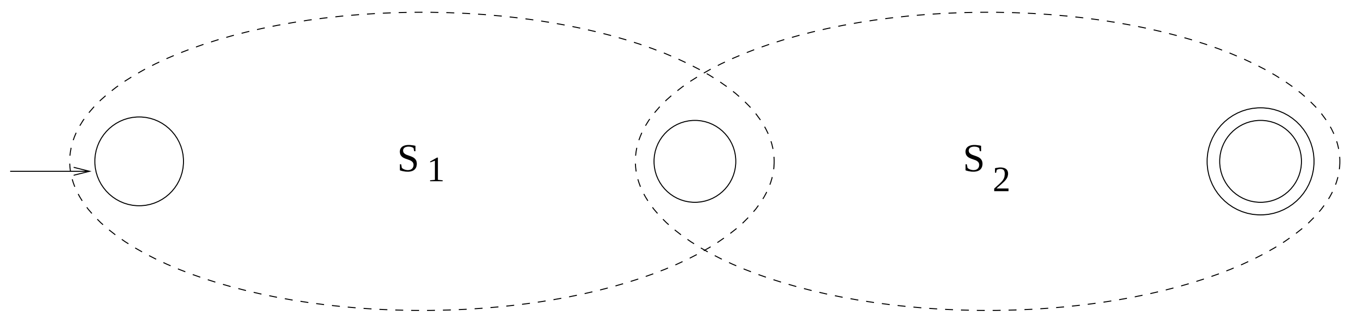

Figure 4: State diagram for the regular expression $E_1 \cdot E_2$.

accepting nor the start state. Then we introduce a new starting state and a new accepting state connected to the merged state by ε-transitions. The construction is depicted in Figure 5.

The constructed state diagram accepts exactly all strings that are the concatenation of a finite number of strings accepted by S_1. Due to the ε-path from the start state to the accepting state the finite number can be zero. More details will be discussed in class.

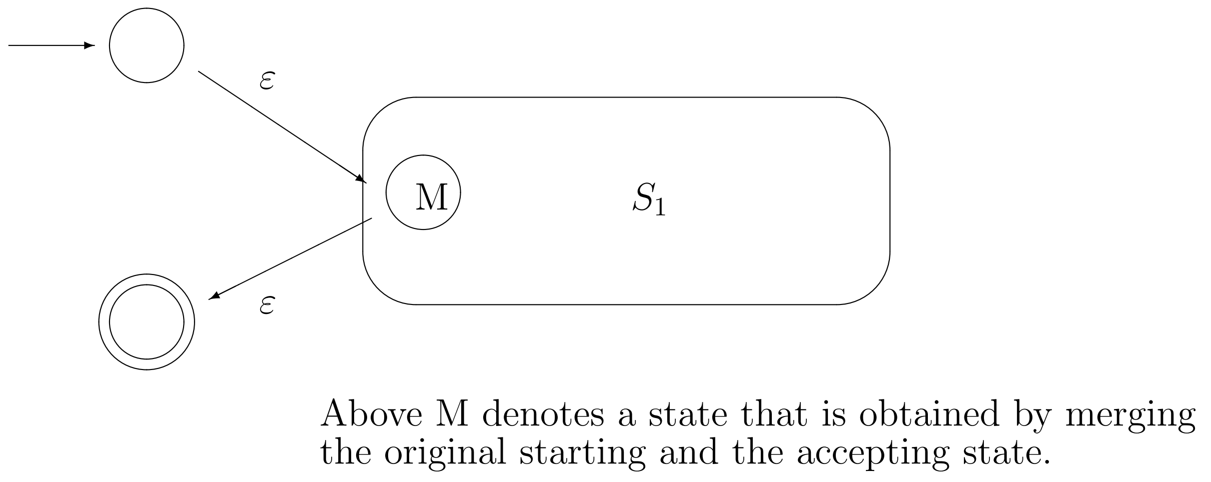
ε
M
S_1
ε
Above M denotes a state that is obtained by merging the original starting and the accepting state.

Figure 5: State diagram for the regular expression E_1^*.

It should be noted that in each of the above cases (iv), (v) and (vi) the resulting state diagram again satisfies the properties 1., 2., 3. This is needed for the correctness of the recursive construction.

Using techniques from the previous chapter (last week) we can first eliminate ε-transitions and then convert the nondeterministic state diagram into an equivalent deterministic state diagram (subset construction).

Example. We apply the construction to the regular expression $(01 + 1)^*1$. To be done in class.

State diagrams to regular expressions (Section 9.2)

Next we consider the converse transformation. For a given state diagram we construct an equivalent regular expression using the so called *state-elimination procedure.* The intermediate stages in the construction are *generalized state-transition diagrams* where the edges can be labeled by any regular expressions (instead of individual alphabet symbols).

The input state diagram is assumed to have exactly one accepting state that is not the start state. A nondeterministic state diagram (NFA) can always be converted into this form. How?

The procedure removes (eliminates) states one at a time. The start state and the accepting state are not removed. The state-elimination step is illustrated in Figure 6. In the figure represents part of a generalized state-transition diagrams and the labels of the arrows (X, Y, Z, W) are regular expressions. If state C is eliminated, the transition from A to B in the modified diagram will be labeled by $X + YW^*Z$.

Important note: When a particular state (like C in the figure) is eliminated, we have to modify transitions between all pairs of states that are connected to the eliminated state as indicated in the figure. In particular, it is possible that A and B may be the same state.

Since the start state and the unique accepting state are not eliminated, at the end of the

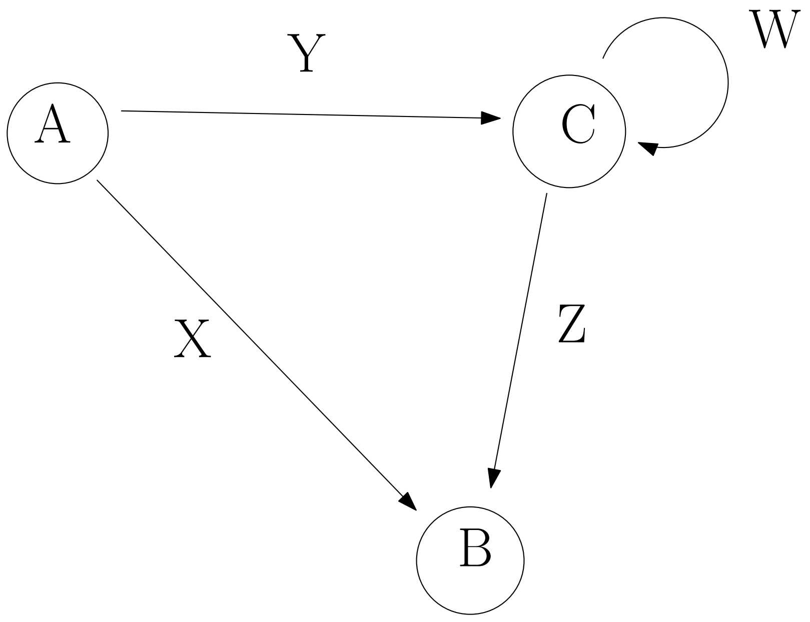

Figure 6: State elimination step: state C is removed and in the new diagram the transition from A to B will be labeled by $X + YW^*Z$.

process we have a two state diagram as depicted in Figure 7. From this diagram we can read the resulting regular expression to be:

$$U^*R(V + SU^*R)^*$$

Explained in class. Note that in the two-state diagram obtained at the end of the process, some of the transitions may be missing, that is, some of the regular expressions R, S, U, V may be $\emptyset$. In this context it is useful to remember what is the result of applying closure to $\emptyset$ or the result of concatenating a language and the empty set.

The state elimination algorithm is described in more detail on pp. 196–197 in the textbook. Below we consider an example.

Example. We consider the state diagram given in Figure 8.

In the first stage of the construction we modify the state diagram so that it has only one accepting state. The resulting state diagram is given in Figure 9, in the figure e stands for

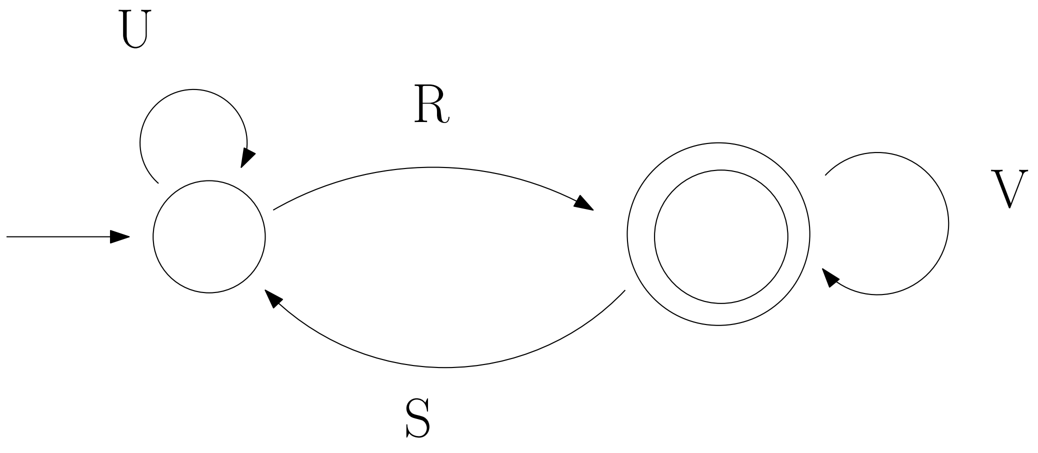

Figure 7: The state diagram (with two states) after all states except the start state and the accepting state have been eliminated.

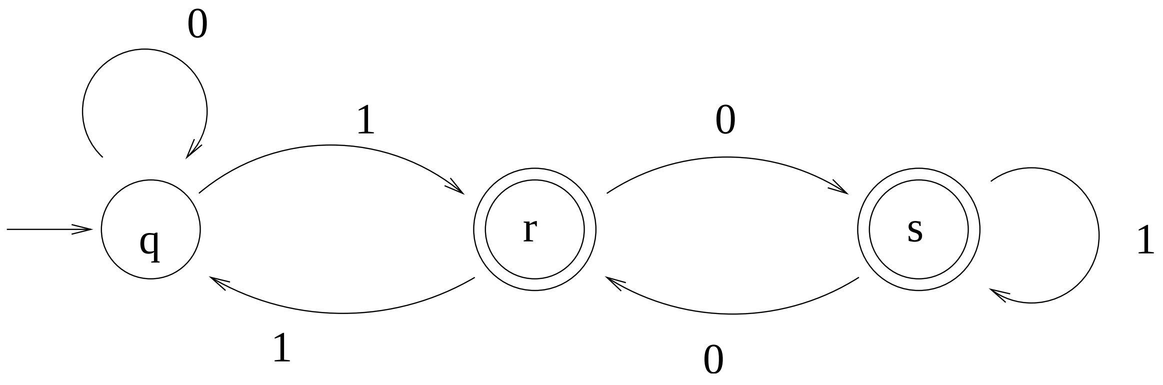

Figure 8: A state diagram to be used as input for the state-elimination algorithm.

the empty string.

Note: The high level description of the state-elimination algorithm does not specify the order in which state states should be eliminated. Thus when applying the procedure to the state diagram of Figure 9 we can first eliminate either state r or state s. Recall that the start state or the unique accepting state cannot be eliminated.

Assuming we apply the state elimination technique to the state diagram of Figure 9 by first eliminating the state r and then the state s, the resulting regular expression will be

$$(0 + 11 + (10(1 + 00)^*01))^*(1 + 10(1 + 00)^*(0 + \varepsilon))$$

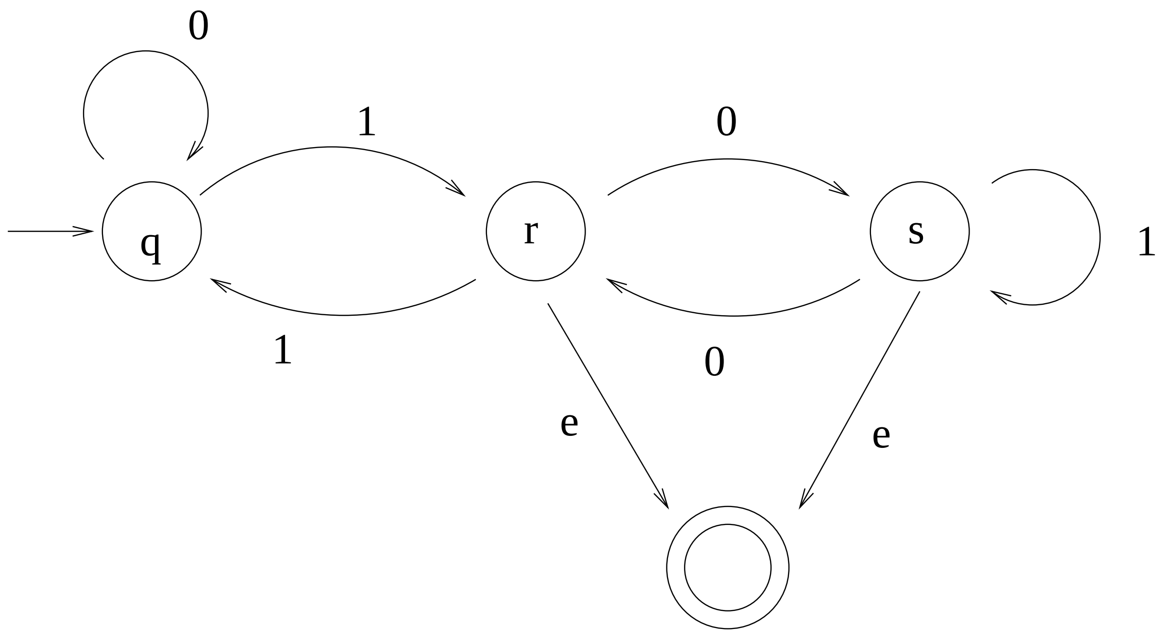

Figure 9: The modified state diagram that has only one accepting state. In the figure e denotes the empty string.

The details will be done in class.

<u>Note:</u> If, in the example, we would begin by eliminating the state s we will get a different regular expression. Generally, using different orders of state-elimination usually produces very different looking regular expressions, however they all denote the same language as the original state diagram.[1]

[1]Although this may not be easy to see just by comparing the regular expressions.

Pumping lemma for regular languages

While the regular languages have nice properties (as we have seen in the early part of the course), unfortunately many languages that we encounter in practice are nonregular. We discuss a method to prove that a given language is nonregular. Such a method is important for understanding the limits of state diagrams and regular expressions. If we can establish nonregularity of a language, there is no need to try construct a regular expression for it.

This material is from Section 9.4 in the textbook.

All regular languages L have the following property:

- corresponding to L there is a constant value called the "pumping length" such that all strings in the language of length at least the "pumping length" can be "pumped", that is, some substring can be repeated arbitrarily many times and the resulting string must remain in the language L.

The result is stated more formally in the below pumping lemma. The pumping lemma gives a general technique for showing that certain languages are not regular. The proof of the pumping lemma will be discussed in class.

Pumping lemma. For every regular language L there exists a constant n such that any string $x \in L$ of length at least n can be written in three parts

$$x = p \cdot q \cdot r$$

where

(P1) $q \neq \varepsilon$

(P2) $|p \cdot q| \leq n$

(P3) $pq^k r \in L$ for all $k \geq 0$.

Note: In the proof of the pumping lemma we can use as n the number of states of a state diagram accepting L. We know that n is a constant, but n can be arbitrarily large.

Examples. The following languages are not regular:

$\{0^i 1^i \mid i \geq 0\}$

$\{0^{2^i} \mid i \geq 0\}$ (this language consists of all strings of 0's having a length that is a power of 2)

How would you use the pumping lemma to show that the above languages are not regular? (Will be done in class.)

Note 2. If L_f is a finite language, no string $x \in L_f$ can be written in three parts $p \cdot q \cdot r$ such that conditions (P1) and (P3) hold because, for large enough k, the strings $pq^q r$ will be longer than the longest string of L_f. Do finite languages satisfy the conditions of the pumping lemma? Why or why not? Are all finite languages regular?

The general method of using the pumping lemma to show that a given language L is not regular can be described as follows. The method uses *proof by contradiction.*

1. For the sake of contradiction we assume that L is regular. Then the pumping lemma gives us the pumping length n. We don't know what n is (it can be arbitrarily large), we only know that it is a constant (positive) integer.

2. Choose a string $x \in L$ of length at least n.

3. Consider all the possible decompositions of x into three parts (as specified in the pumping lemma). If no decomposition of x into three parts satisfies the conditions (P1), (P2), (P3) from the pumping lemma simultaneously, we obtain a *contradiction* and can conclude that L cannot be regular.

The crucial part in using the pumping lemma is usually to select a suitable string $x \in L$ in 2. above. The string x should be selected so that it cannot be pumped (without going "out" of the language L). Note that $x \in L$ has to be specified using the "unknown" constant n.

Above in stage 2., we can choose the string $x \in L$ in a way that we expect to cause problems with the "pumping property". Note, however, that in the following stage 3. we need to show that no decomposition of x into three parts does satisfies the conditions (P1), (P2), (P3). That is, it is *not* sufficient to consider one particular decomposition.

Sometimes in order to show that given languages are not regular, together with the pumping lemma, we can rely on *closure properties* of the family of regular languages.

The class of regular languages has strong closure properties, in particular, regular languages are closed under the Boolean operations. This means that if S and T are regular languages then also the following languages are regular:

$S \cup T$

$S \cap T$

$\overline{S} = \{w \in \Sigma^* \mid w \notin S\}$

The regularity of $S \cup T$ follows directly from the definition regular expressions. How can you establish would you prove closure under intersection and complement? (Will be done in class.)

Example. Define S to consist of all strings over alphabet the $\{0, 1\}$ that have an equal number of occurrences of 0's and 1's. We show that S is not regular.

Proof by contradiction: If S is regular, then also the language

$$S \cap 0^*1^* = \{0^i1^i \mid i \geq 0\}$$

is regular. (Why?) We have in the earlier example shown that the language $\{0^i 1^i \mid i \geq 0\}$ is not regular. We conclude that S cannot be regular.

Example. Let $\Sigma = \{a, b, c\}$ and consider the language L_0 over Σ:

$$L_0 = \{a^k b^i c^i \mid i \geq 0, k \geq 1\} \cup \{b^r c^s \mid r \geq 0, s \geq 0\}.$$

Intuitively, L_0 should not be regular because after a positive number of symbols a, the numbers of b's and c's must match and clearly a state diagram cannot count arbitrarily large numbers. However, if we try to use the pumping lemma to prove non-regularity of L_0, we run into trouble because in a string of the form $a^k b^i c^i$ a non-empty prefix consisting of a's can always be repeated. Also if $k = 1$ and we try to "pump down" the result will be in the set $\{b^r c^s \mid r \geq 0, s \geq 0\}$.

In fact, directly using the pumping lemma it is not possible to prove the non-regularity of L_0.[1] That is, L_0 satisfies the pumping lemma but L_0 is not regular (as we show below).

Using closure properties we can easily establish non-regularity of L_0:

$$L_0 \cap \{ab^r c^s \mid r \geq 0, s \geq 0\} = \{ab^i c^i \mid i \geq 0\}.$$

Using the pumping lemma it is easy to establish that $\{ab^i c^i \mid i \geq 0\}$ is non-regular. Since $\{ab^r c^s \mid r \geq 0, s \geq 0\}$ is denoted by the regular expression $ab^* c^*$ and regular languages are closed under intersection we can conclude that L_0 is non-regular.

[1] If you are skeptical, you are welcome to try.

As the last topic on regular languages we consider an algorithm to minimize deterministic state diagrams. More on this topic can be found in the textbook by P. Linz [3]. A link to the textbook is given in CISC/CMPE-223 onQ pages.

Unreachable/useless states: A state diagram, whether deterministic or nondeterministic, may have states that cannot be reached in computations on any input word, such states are called *useless.* This includes states that cannot be reached from the start state and states that have no outgoing path reaching an accepting state. By viewing a state diagram as a directed graph, useless states can be found using a straigthforward graph reachability algorithm. How? Once the useless states are identified, they can be simply deleted from the state diagram.

Removing useless states is a first step in minimization, however, a DFA with no useless states need not be minimal. Consider the example given in Figure 1. Here states B and C are indistinguishable (as defined more precisely below) in the sense that any string w takes the state B to an accepting state if and only if w takes C to an accepting state. Indistinguishable states can be merged into one state. In the DFA of Figure 1 also states A and D are indistinguishable.

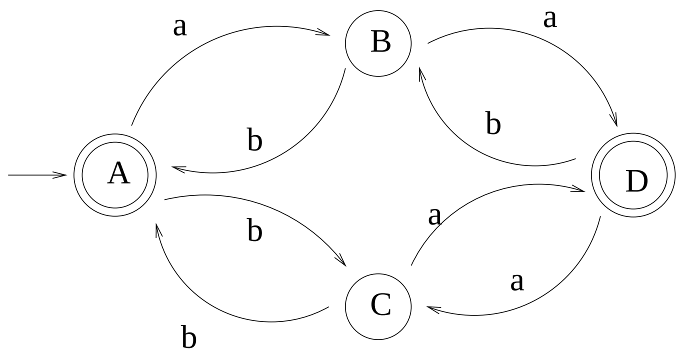

Figure 1: The states B and C can be merged into one state. Also states A and D can be merged.

Based on the above idea of merging indistinguishable states we present an algorithm to minimize an arbitrary DFA.[1] More on on minimization of DFAs can be found in the textbook by P. Linz [3]. A link to the text is given on the course web site.

For the algorithm we need to define a notion of distinguishability for states of a DFA and for this purpose we use some formal notation. Recall that a DFA was defined as a tuple $M = (Q, \Sigma, \delta, s, F)$ where Q is the set of states, Σ is the input alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $s \in Q$ is the start state and $F \subseteq Q$ is the set of accepting states. We extend δ as a function $\hat{\delta} : Q \times \Sigma^* \to Q$ by defining inductively

1. $\hat{\delta}(q, \varepsilon) = q$ for all $q \in Q$, and,
2. $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$ for all $q \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$.

In the following also the extended transition function $\hat{\delta}$ is denoted simply by δ. For a state $q \in Q$ and string $w \in \Sigma^*$, $\delta(q, w)$ is the unique state that the DFA M is in after reading the string w assuming the computation starts in state q. Note that here M is a *complete DFA,* that is, $\delta : Q \times \Sigma \to Q$ is a total function. Earlier in the course we have considered also *incomplete DFAs* where some transitions could be undefined. An incomplete DFA can be easily completed by adding a so called "sink state" that is the target of all previously undefined transitions.[2]

Definition. With the above notation we can now define that states q_1 and q_2 are *indistinguishable* if

$$(\forall w \in \Sigma^*) \;\; \delta(q_1, w) \in F \text{ iff } \delta(q_2, w) \in F.$$

The states q_1 and q_2 are *distinguishable (via string v)* if v violates the above condition, that is, if $\delta(q_1, v) \in F$ and $\delta(q_2, v) \notin F$ or vice versa.

[1]The minimization algorithm does not, in general, work for nondeterministic state diagrams.

[2]Adding the sink state may require adding a large number of transitions and this often makes drawing the state diagram messy. For this reason, often in examples the sink state is omitted. However, when dealing with the minimization algorithm we assume that all transitions are defined.

The idea behind the minimization algorithm is to find all pairs of distinguishable states. As the starting point of the algorithm we note that any accepting state (an element of F) is always distinguishable from a nonaccepting state (an element of $Q - F$). Why? Initially the algorithm marks all such pairs as distinguishable. After finding all pairs of distinguishable states, we know that the remaining pairs are indistinguishable and, consequently, they can be merged into one state.

We call the algorithm "mark distinguishable pairs of states." As preprocessing we assume that states not reachable from the start state have been removed.

Algorithm: *Mark distinguishable pairs of states.* The input for the algorithm is a DFA $M = (Q, \Sigma, \delta, s, F)$ where all states are reachable from the start state. The algorithm marks all pairs of states (q_1, q_2) such that q_1 and q_2 are distinguishable.

- Stage 0: Mark all pairs (q_1, q_2) where $q_1 \in Q - F$ and $q_2 \in F$, or vice versa.
- *Repeat* the following until at some ith stage no new pairs are marked:
  Stage i ($i \geq 1$): For each unmarked pair (q_1, q_2) and each $b \in \Sigma$ do the following. If the pair $(\delta(q_1, b), \delta(q_2, b))$ has been marked distinguishable at stage $i - 1$, then mark (q_1, q_2) as distinguishable.

Note that a pair (q_1, q_2) is distinguishable if and only if (q_2, q_1) is distinguishable. This means that when implementing the algorithm (or when tracing the algorithm "by hand") it is sufficient to consider unordered pairs of states.

Example 1. Consider the DFA M_1 of Figure 1. We have already observed that M_1 is not minimal.

When applying the marking algorithm to M_1, at stage 0 we mark pairs (A, B), (A, C), (B, D) and (C, D). As observed above, distinguishability is symmetric and hence marking (A, B) implies that also (B, A) is marked, that is, we are considering unordered pairs.

At stage one of the algorithm we now need to consider the unmarked pairs (A, D) and (B, C). We note that $(\delta(A, a), \delta(D, a)) = (B, C)$ and $(\delta(A, b), \delta(D, b)) = (C, B)$ where B and C are indistinguishable at previous stage 0. Hence the pair (A, D) is not marked at stage 1. Similarly it is observed that $(\delta(B, a), \delta(C, a)) = (D, D)$ and $(\delta(B, b), \delta(C, b)) = (A, A)$. Since a state is never distinguishable from itself, this verifies that the also the pair (B, C) will not be marked in stage 1. Since no new pairs were marked in stage 1, the algorithm terminates after stage 1.

The resulting minimized DFA is depicted in Figure 2.

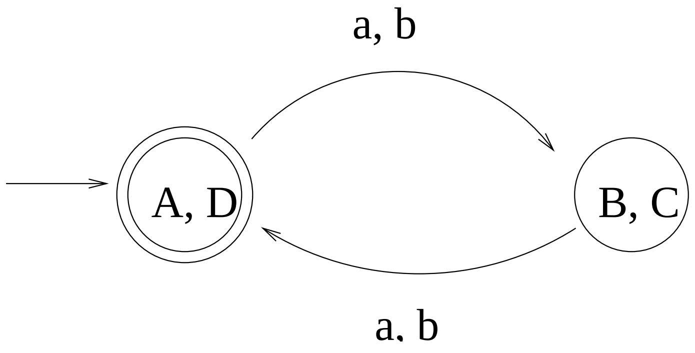

Figure 2: The minimized DFA equivalent to the DFA of Figure 1.

As indicated in the above example, the minimized DFA is obtained from the original DFA M by merging together into one class all states q_1, q_2 such that the pair (q_1, q_2) remains unmarked after the execution of the algorithm "*mark distinguishable pairs of states*". When merging q_1 and q_2, also the corresponding outgoing transitions are "merged" together. This is always possible because the pair (q_1, q_2) remaining unmarked means that the states q_1 and q_2 are indistinguishable, and hence, for any $b \in \Sigma$, the outgoing transitions from q_1 and q_2 on symbol b end up in a pair of indistinguishable states (that are also merged into one state).

More formally, assume that the original DFA M has a transition from state q to state p on input symbol b. Then the minimized DFA has a transition on input b from the "merged together class" containing q to the class containing p.[3] The start state of the minimized

[3]More formally, indistinguishability is an *equivalence relation* on the set of states and partitions the set

DFA is the class that contains the start state of the original DFA. All classes consisting of accepting states are accepting. Note that an accepting state can never belong to the same class as a non-accepting state.

Example 2. As a slightly bigger example, we use the algorithm to minimize the DFA M_2 of Figure 3. Here the alphabet is $\Sigma = \{a, b\}$. The details will be done in class.

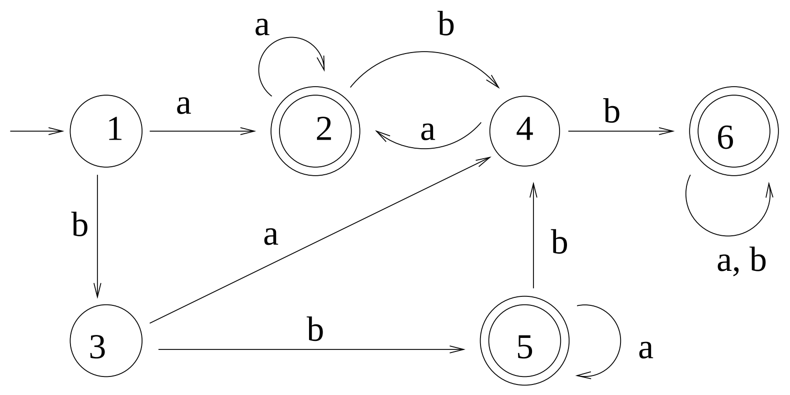

Figure 3: DFA M_2

It is easy to see that if a DFA M has n states the minimization algorithm always terminates after the $(n-1)$th stage.[4]

It can be shown that the algorithm produces a minimal DFA equivalent to the original DFA M and, furthermore, the minimal DFA is unique for any regular language [3]. That is, if M and M' are any DFAs recognizing the same language, by applying the minimization algorithm to M and to M', respectively, yields the same minimal DFA.

A naive implementation of the minimization algorithm runs in cubic time. A considerably more sophisticated variant can be made to run in $O(n \cdot \log n)$ time [2].

of states into equivalence classes – you may recall equivalence relations e.g. from the course CISC-203.

[4]At stage i the unmarked pairs define a partition of the state set into subsets where any two states in the same subset cannot be distinguished by any string of length at most i. At stage $i+1$ the partition is a refinement of the stage i partition and, if the total number of states is n, the refinement cannot be done more than $n-1$ times.

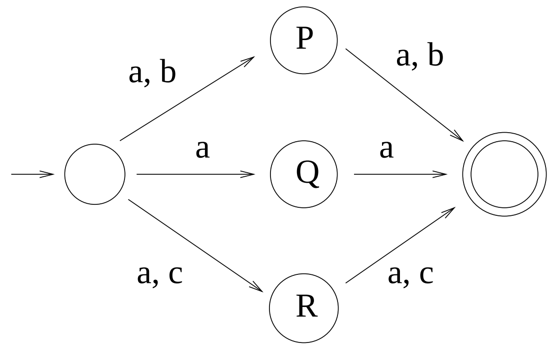

Figure 4: States P and Q can be merged or states Q and R can be merged. However, P, Q, R cannot all be merged.

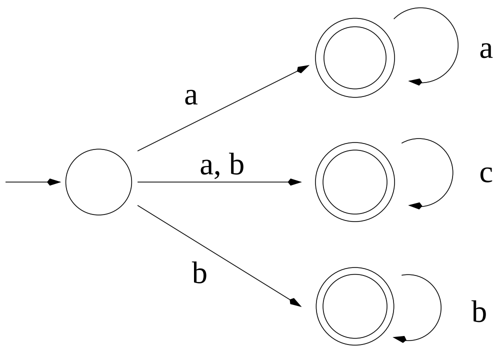

Figure 5: No two states can be merged, however, the NFA is not minimal.

Simplification of NFAs and regular expressions

Exactly as for deterministic state diagrams, in a given NFA we can identify and eliminate the useless states using a simple graph reachability algorithm. Here by useless states we mean states that cannot be reached from the start state on any string, or states from which an accept state cannot be reached on any string.

Analogously as was done for DFAs, we may attempt to reduce the number of states of an NFA by merging together "equivalent" states. However, the end result may depend on the order in which we choose to merge the states and, furthermore, a nonminimal NFA need not have any mergible states. This is illustrated by the two examples given in Figures 4 and 5.

The simple NFAs of Figures 4 and 5 can be easily minimized using exhaustive search.

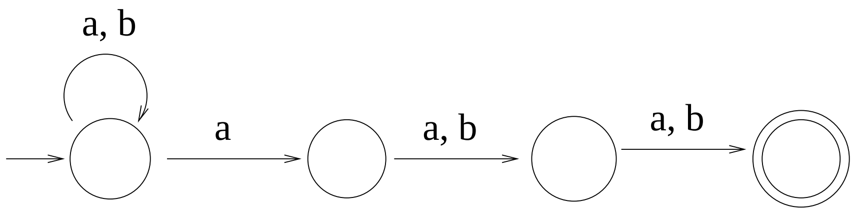

Figure 6: An NFA A_0 for which the equivalent minimal DFA has 8 states.

However, the phenomena illustrated by these examples imply that NFA minimization becomes a hard combinatorial problem and minimization, in general, is known to be intractable [1].[5]

Also the simplification of regular expressions is a hard computational problem. The regular expressions produced by the "state elimination algorithm" discussed in the previous section are often very large. The state elimination algorithm is implemented in the software package *Grail.* Using *Grail* we can determinize the NFA A_0 depicted in Figure 6 and the resulting DFA has 8 states. When, again using *Grail,* we apply the state elimination algorithm to the 8 state DFA, the regular expression has size over 32000 bytes. For this language the "obvious" regular expression is $(a + b)^*a(a + b)(a + b)$, however, when given the corresponding DFA as input, the state elimination algorithm cannot find any regular expression of reasonable size.

The size of the regular expressions produced by the state elimination algorithm can be reduced by various heuristic simplification techniques. There is no known general simplification algorithm and regular expression simplification is a current research topic.

References

[1] M. Holzer and M. Kutrib, Descriptional and computational complexity of finite automata — A survey. *Information and Computation* 209 (2011) 456–470.

[5]Using technical language, the algorithmic problem of NFA minimization is PSPACE-complete.

[2] J.E. Hopcroft, An $n \cdot \log n$ algorithm for minimizing the states in a finite automaton. In: Z. Zohavi, A. Paz (Eds.), *International Symposium on the Theory of Machines and Computations,* Academic Press, 1971, pp. 189–196.

[3] P. Linz, *An Introduction to Formal Languages and Automata,* (section 2.4). Jones and Bartlett Publishers. `A link to the on-line edition can be found on the course web page.`

Context-Free Languages

This material is described in Chapter 10 of the textbook.

We have observed that regular languages have nice properties, however, the specification capabilities of regular expressions and state diagrams are limited. Next we consider a rewriting mechanism, or grammar, that overcomes some of the limitations. Context-free grammars are more powerful than state diagrams, that is, they define a larger class of languages.

Context-free grammars are the most widely used specification tool for the syntactic structure of programming languages. The appendix of the textbook contains a grammar specification for a subset of the C language that covers features that are needed for the program verification part of the course.

In the programming language community grammars were originally specified using the Backus-Naur formalism (BNF):

- the nonterminals are indicated with angle brackets
- the left and right sides of rules (productions) are separated by `::=`
- `|` indicates alternative definitions

Example. `<expr> ::= <expr> + <expr> | <expr>` $\times$ `<expr> | (<expr>) | a`

Note that the string `a + a` $\times$ `a` has two essentially different derivations: the derivation can begin either with the "plus" rule or the "product" rule. Which derivation should we use?

Notational convention for grammars: In the following we will use $\rightarrow$ instead of `::=` for the productions (or rewriting rules). We normally denote nonterminals/variables by upper case letters and terminal symbols by lower case letters or digits.

Example. Design a context-free grammar for the language

$$\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$$

Next we will go through a more formal definition of a context-free grammar. Also we define formally the derivations of the grammar and the set of terminal strings (language) generated by the grammar.

Definition. A <u>context-free grammar</u> (CFG) is a 4-tuple (V, Σ, P, S) where

1. V is a finite set of nonterminals (also called variables)
2. Σ is a finite set of terminals; Σ and V are disjoint
3. P is a finite set of productions, the left side of each production is a nonterminal and the right side is a string of nonterminals and terminals
4. $S \in V$ is the start nonterminal

The definition means that the set of productions P is a finite subset of $V \times (\Sigma \cup V)^*$. The productions are applied only to variables, that is, the left side of a production is always a variable. The right side of a production is a string of variables and terminal symbols.

A derivation begins with the start variable S. The productions of the grammar are applied to any variable occurring in the "current string" (a.k.a. sentential form) and the process is continued as long a we get a string consisting only of terminal symbols. The language generated by the grammar consists of all terminal strings that are obtained in this way. Next we define formally the notion of *derivation.*

Let w_1 and w_2 be strings over $\Sigma \cup V$ (sets of terminals and nonterminals). We say that w_2 is immediately derivable from w_1 if w_2 can be obtained from w_1 by replacing one occurrence of a nonterminal by a string that appears on the right side of a production for that nonterminal.

Using notation the single step derivation is defined as follows:

- $w_1 = xNy$, $N \in V$, $x, y \in (V \cup \Sigma^*)$,
- $w_2 = xzy$ where $N \to z$ is a production of P.

If w_2 is obtained from w_1 in a single derivation step, we write $w_1 \Rightarrow w_2$.

The language generated by the grammar is the set of terminal strings that can be derived from the start nonterminal, that is,

$$\{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

The relation $\Rightarrow^*$ is the *reflexive-transitive closure* of the single-step derivation relation $\Rightarrow$. That is, $u \Rightarrow^* v$ if $u = v$ or there exists a sequence of strings $u_1, \ldots, u_k$, $k \geq 0$, such that

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$$

Example. Consider the grammar

$$G = (\{S\}, \{a, b\}, P, S)$$

where the productions of P are

$$S \to aSb \mid SS \mid \varepsilon$$

What is the language generated by this grammar? How can you describe the set of terminal strings generated by this grammar.

Regular grammars. A context-free grammar where all productions are of the forms

$B \to bC$

$B \to b \mid \varepsilon$

where B, C are nonterminals and b is a terminal, is called a <u>regular grammar</u>. It is not difficult to show that regular grammars generate exactly the regular languages. (Time permitting this can be discussed in class.) This means that every regular language is generated by some context-free grammar. We have already seen that there exist context-free languages that are not regular.

Parse Trees and Ambiguity

A context-free derivation can apply a production to any variable in a sentential form. This means that normally there are multiple ways to derive a given terminal string – exceptions can be, for example, right-linear grammars where a sentential form includes only one variable.

To begin with a very simple example, consider a grammar with productions:

$$S \to AB, \quad A \to a \quad B \to b$$

where S is the start variable, A, B are variables and a, b are terminals. Now the terminal string has two different derivations

$$S \Rightarrow AB \Rightarrow aB \Rightarrow ab \text{ and } S \Rightarrow AB \Rightarrow Ab \Rightarrow ab.$$

However, the derivations differ only in the order of production applications and both derivations have the *same parse tree.*

The notion of *ambiguity* refers to a situation where a given terminal string has more than one parse tree. Since compilers do program translation based on the parse tree it is desirable that each terminal string can be parsed in only one way - such grammars are called *unambiguous.*

Consider our earlier example with a simplified expression grammar[1]:

`<expr>` $\to$ `<expr> + <expr> | <expr>` $\times$ `<expr> | (<expr>) | a`

[1]For simplicity we use only one terminal value "a" for an expression. A more realistic grammar would have additional rules that can generate e.g. any integer value.

Since expressions would be evaluated based on the parse tree, ambiguity is an undesirable feature. For example, the expression `a + a` $\times$ `a` can be derived by first using the production `<expr>` $\rightarrow$ `<expr> + <expr>` or `<expr>` $\rightarrow$ `<expr>` $\times$ `<expr>` and these derivations yield distinct parse trees.

Here if the expression is evaluated according to the parse tree, the two parse trees may produce different values! If we consider how the expression `a + a` $\times$ `a` is evaluated and the usual precedence rules for addition and multiplication, the parse tree where we apply first the production `<expr>` $\rightarrow$ `<expr> + <expr>` yields the correct evaluation.

Ambiguity is a challenging problem and even determining whether a given grammar is unambiguous is a computationally unsolvable problem. There exist ad hoc techniques to remove ambiguity from a grammar. In our example, we can redesign the above grammar by introducing new nonterminals:

`<expr>` $\rightarrow$ `<term> | <expr> + <term>`

`<term>` $\rightarrow$ `<factor> | <term>` $\times$ `<factor>`

`<factor>` $\rightarrow$ `(<expr>) | a`

The intuitive idea is that, in the modified grammar, variable `<expr>` genereates a sum of "terms" and each variable `<term>` generates a product of "factors". The modified grammar generates the same terminal strings (expressions) as the original grammar and it is $\underline{\text{unambiguous}}$. (The example may be discussed in more detail in class.)

Pushdown automata (Section 10.5)

A grammar is a generative mechanism: the process begins with the start variables and applies rewriting rules to produce a terminal string. On the other hand, when a compiler receives a program (that is, a terminal string) as input and has to process the parse tree based on the program. For this purpose we use a machine model called *pushdown automaton* that is equivalent to a context-free grammar. The parser stage of a compiler will be, roughly speaking, a deterministic pushdown automaton.

A pushdown automaton (PDA) can write symbols on a stack and read them back later. Writing a symbol "pushes down" all the other symbols on the stack. The symbol on the top of the stack can be read and removed ("pop" operation), see Figure **??**.

A pushdown automaton can be depicted as a directed graph. Similarly as for state diagrams, the states are depicted by nodes (circle) in the graph. The arrows labeling the state transitions need to include both the input symbol read and the operation performed on the stack.

According to notational conventions used in our textbook, a transition from state q_1 to state q_2 labeled by

$$c, d \mapsto w$$

indicated that the computation can switch from state q_1 to state q_2 by reading c from the input, popping (removing) d from the top of the stack and pushing string w to the stack. Here c is an individual input symbol or the empty string, d is an individual stack symbol or the empty string, and, w is a string of stack symbols. That is, one operation pops at most on symbol from the stack but an operation can push a string of stack symbols.

A pushdown automaton begins processing the input in the initial state and with the stack contents empty. The computation accepts the input if at the right end of the input the machine is in an accepting state and the stack is empty.

More details of the definition of pushdown automata and examples are given on pages 216–219 in the textbook.

Note. While the definition of state diagrams (DFAs or NFAs) is fairly consistent in different sources, the precise details of a pushdown automaton definition vary depending on different textbooks.[2] In the lectures and assignments we will follow the notational conventions of our

[2]For example, some textbooks use different acceptance conditions for a computation.

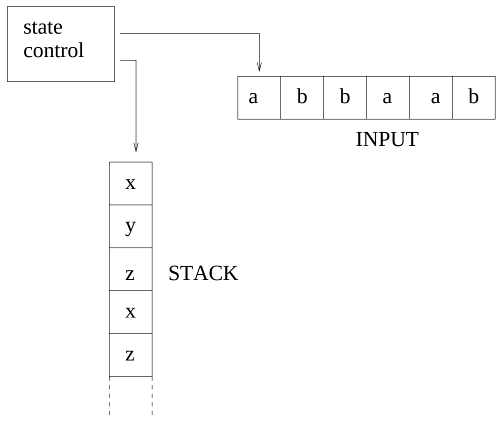

Figure 1: A pushdown automaton

textbook.

Example. Construct a pushdown automaton for the language

$$\{a^i b^i \mid i \geq 0\} \cup \{a^{2i} b^i \mid i \geq 0\}$$

(To be done in class.)

Pushdown automata recognize exactly the class of context-free languages. We will go through the construction showing how to construct a pushdown automaton recognizing the language generated by a grammar. The converse inclusion is omitted in this course.

Pumping lemma for context-free languages (Section 10.6)

Context-free languages form a larger class than regular languages. Context-free grammars have important applications in specifying the syntax of programming languages, however, context-free grammars cannot specify certain programming language feature, like the requirement that variable names need to be declared before their use.

To have an understanding of the limitations of context-free grammars we need a tool for proving that a language is not context-free. The *context-free pumping lemma* is based on similar ideas as our earlier pumping lemma for regular languages. Roughly speaking, the difference is that in the context-free pumping we have to repeat in parallel two substrings.

Pumping Lemma for Context-Free Languages. For every context-free language L there is a constant p such that every string $s \in L$ of length at least p can be written as

$$s = uvwxy$$

where

(i) $v \neq \varepsilon$ or $x \neq \varepsilon$

(ii) $|vwx| \leq p$

(iii) for each $i \geq 0$, $uv^iwx^iy \in L$

Similarly as with regular languages, the use of the context-free pumping lemma is a proof by contradiction. For the sake of contradiction we assume that the language is context-free. Then from the pumping lemma we get the constant p and select a string s of length at leastr p. To derive a contradiction we have to prove that no matter how we try to divide s into five parts $uvwxy$ the parts cannot satisfy the above conditions (i), (ii), (iii).

Because the context-free pumping lemma repeats two substrings and we consider decompositions into five parts, the details of the proof are some times more complicated and involve a case analysis to cover all possible ways to divide s into five parts. The following two examples will be covered in detail in class.

Examples. Show that the following languages are not context-free:

a) $\{a^ib^ic^i \mid i \geq 0\}$

b) $\{ww \mid w \in \{a,b\}^*\}$

Note that above the language $\{ww \mid w \in \{a,b\}^*\}$ abstracts the idea that context-free grammars cannot specify the requirement that variable names have to be declared before their use. Assuming there is no upper limit on the length of a variable name, a grammar specifying the requirement would need to generate strings of the form

$$\ldots w \ldots w \ldots, \text{ for } w \in \Sigma^*$$

where ... refer to other parts of the program that are omitted in the abstraction.

Applications of grammars: RNA secondary structure prediction using context-free grammars

Regular languages and finite state machines are often used in DNA and protein sequence matching problems (see notes for week 1). The use of regular languages is based on the assumption that the mutations that caused variations in the sequence occurred independently of each other.

The situation is different when we consider the secondary (or three dimensional) structure of an RNA molecule. RNA molecules with same secondary structure usually have the same function. Because of the way RNA molecules fold, in order to preserve the secondary structure, variations to nucleotides in one part of the sequence must be matched in the corresponding bonded subsequence (the bonded subsequences are called a *stem*.) This type of nested dependencies can be described using context-free grammars.

We consider a simple example. The RNA nucleotides contain four different bases: adenine (A), guanine (G), cytosine (C), uracil (U). C and G, and A and U, respectively, are complementary. In the below grammar we use lower case letters (a, g, c, u) since they are the terminals of the grammar.

To generate RNA structures we use a grammar (V, Σ, P, S) where $V = \{S\}$, $\Sigma = \{a, g, c, u\}$, and P consists of the productions:

(i) $S \longrightarrow aSu \mid uSa \mid cSg \mid gSc$

(ii) $S \longrightarrow aS \mid cS \mid gS \mid uS$

(iii) $S \longrightarrow Sa \mid Sc \mid Sg \mid Su$

(iv) $S \longrightarrow SS \mid \varepsilon$

A parse tree for the terminal string

$$ccugagaggcaaccuagaaggu \tag{1}$$

is given in Figure **??**. The parse tree corresponds to the RNA secondary structure with two stems (or bonded subsequences) that is illustrated in Figure **??**.

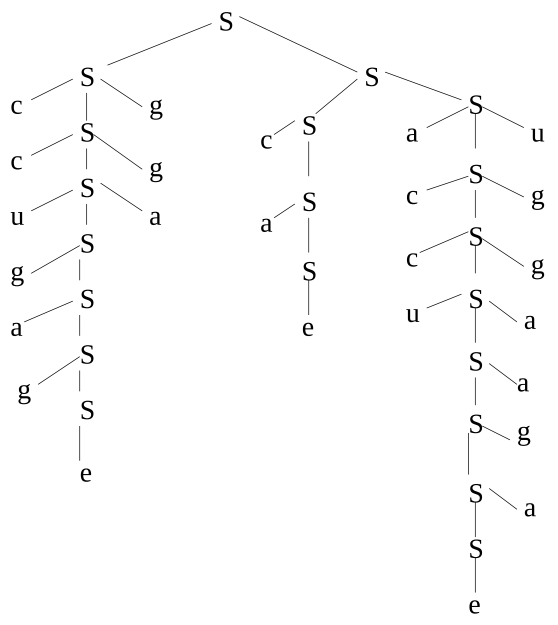

Figure 2: Parse tree for an example RNA structure. In the figure, “e” denotes the empty string ε.

From the parse tree we can read the secondary structure: the application of the grammar rules (i) produces a stem of bonded base pairs. Note that the grammar is *ambiguous*: the same RNA sequence (**??**) has many other parse trees. The ambiguity is, for the most part, caused by the possibility of applying the rules (ii), (iii) in different order. Derivations that differ only in the order of application of the rules (ii) and (iii) correspond to the same secondary structure.

Above we have outlined how context-free grammars can be used to model RNA folding. A more detailed study tells us that complementary pairs (C–G and A–U) are the likeliest

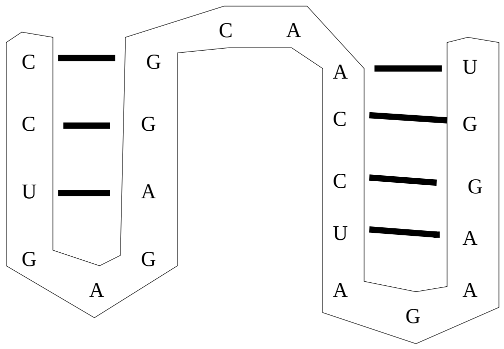

Figure 3: An RNA secondary structure. The thick lines indicate bonds between complementary bases in a stem.

candidates to form base pairs but, sometimes, also other pairs are formed. For more accurate secondary structure prediction the grammar rules need to be augmented with probabilities. The computerized methods used for RNA secondary structure prediction use probabilistic models called *stochastic context-free grammars.* An introductory survey can be found e.g. in [**?**].

References

[1] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, *Biological sequence analysis: Probabilistic models of proteins and nucleic acid.* Cambridge University Press, 1998.

Context-Free Languages

This material is described in Chapter 10 of the textbook.

We have observed that regular languages have nice properties, however, the specification capabilities of regular expressions and state diagrams are limited. Next we consider a rewriting mechanism, or grammar, that overcomes some of the limitations. Context-free grammars are more powerful than state diagrams, that is, they define a larger class of languages.

Context-free grammars are the most widely used specification tool for the syntactic structure of programming languages. The appendix of the textbook contains a grammar specification for a subset of the C language that covers features that are needed for the program verification part of the course.

In the programming language community grammars were originally specified using the Backus-Naur formalism (BNF):

- the nonterminals are indicated with angle brackets
- the left and right sides of rules (productions) are separated by `::=`
- `|` indicates alternative definitions

Example. `<expr> ::= <expr> + <expr> | <expr>` $\times$ `<expr> | (<expr>) | a`

Note that the string `a + a` $\times$ `a` has two essentially different derivations: the derivation can begin either with the "plus" rule or the "product" rule. Which derivation should we use?

Notational convention for grammars: In the following we will use $\to$ instead of `::=` for the productions (or rewriting rules). We normally denote nonterminals/variables by upper case letters and terminal symbols by lower case letters or digits.

Example. Design a context-free grammar for the language

$$\{0^n1^n \mid n \geq 0\} \cup \{1^n0^n \mid n \geq 0\}$$

Next we will go through a more formal definition of a context-free grammar. Also we define formally the derivations of the grammar and the set of terminal strings (language) generated by the grammar.

Definition. A context-free grammar (CFG) is a 4-tuple (V, Σ, P, S) where

1. V is a finite set of nonterminals (also called variables)
2. Σ is a finite set of terminals; Σ and V are disjoint
3. P is a finite set of productions, the left side of each production is a nonterminal and the right side is a string of nonterminals and terminals
4. $S \in V$ is the start nonterminal

The definition means that the set of productions P is a finite subset of $V \times (\Sigma \cup V)^*$. The productions are applied only to variables, that is, the left side of a production is always a variable. The right side of a production is a string of variables and terminal symbols.

A derivation begins with the start variable S. The productions of the grammar are applied to any variable occurring in the "current string" (a.k.a. sentential form) and the process is continued as long a we get a string consisting only of terminal symbols. The language generated by the grammar consists of all terminal strings that are obtained in this way. Next we define formally the notion of *derivation.*

Let w_1 and w_2 be strings over $\Sigma \cup V$ (sets of terminals and nonterminals). We say that w_2 is immediately derivable from w_1 if w_2 can be obtained from w_1 by replacing one occurrence of a nonterminal by a string that appears on the right side of a production for that nonterminal.

Using notation the single step derivation is defined as follows:

- $w_1 = xNy$, $N \in V$, $x, y \in (V \cup \Sigma^*)$,
- $w_2 = xzy$ where $N \to z$ is a production of P.

If w_2 is obtained from w_1 in a single derivation step, we write $w_1 \Rightarrow w_2$.

The language generated by the grammar is the set of terminal strings that can be derived from the start nonterminal, that is,

$$\{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

The relation $\Rightarrow^*$ is the *reflexive-transitive closure* of the single-step derivation relation $\Rightarrow$. That is, $u \Rightarrow^* v$ if $u = v$ or there exists a sequence of strings $u_1, \ldots, u_k$, $k \geq 0$, such that

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$$

Example. Consider the grammar

$$G = (\{S\}, \{a, b\}, P, S)$$

where the productions of P are

$$S \to aSb \mid SS \mid \varepsilon$$

What is the language generated by this grammar? How can you describe the set of terminal strings generated by this grammar.

Regular grammars. A context-free grammar where all productions are of the forms

$B \to bC$

$B \to b \mid \varepsilon$

where B, C are nonterminals and b is a terminal, is called a regular grammar. It is not difficult to show that regular grammars generate exactly the regular languages. (Time permitting this can be discussed in class.) This means that every regular language is generated by some context-free grammar. We have already seen that there exist context-free languages that are not regular.

Parse Trees and Ambiguity

A context-free derivation can apply a production to any variable in a sentential form. This means that normally there are multiple ways to derive a given terminal string – exceptions can be, for example, right-linear grammars where a sentential form includes only one variable.

To begin with a very simple example, consider a grammar with productions:

$$S \to AB, \quad A \to a \;\; B \to b$$

where S is the start variable, A, B are variables and a, b are terminals. Now the terminal string has two different derivations

$$S \Rightarrow AB \Rightarrow aB \Rightarrow ab \;\text{ and }\; S \Rightarrow AB \Rightarrow Ab \Rightarrow ab.$$

However, the derivations differ only in the order of production applications and both derivations have the *same parse tree.*

The notion of *ambiguity* refers to a situation where a given terminal string has more than one parse tree. Since compilers do program translation based on the parse tree it is desirable that each terminal string can be parsed in only one way - such grammars are called *unambiguous.*

Consider our earlier example with a simplified expression grammar[1]:

`<expr>` $\to$ `<expr> + <expr> | <expr>` $\times$ `<expr> | (<expr>) | a`

[1]For simplicity we use only one terminal value "a" for an expression. A more realistic grammar would have additional rules that can generate e.g. any integer value.

Since expressions would be evaluated based on the parse tree, ambiguity is an undesirable feature. For example, the expression `a + a` $\times$ `a` can be derived by first using the production `<expr>` $\rightarrow$ `<expr> + <expr>` or `<expr>` $\rightarrow$ `<expr>` $\times$ `<expr>` and these derivations yield distinct parse trees.

Here if the expression is evaluated according to the parse tree, the two parse trees may produce different values! If we consider how the expression `a + a` $\times$ `a` is evaluated and the usual precedence rules for addition and multiplication, the parse tree where we apply first the production `<expr>` $\rightarrow$ `<expr> + <expr>` yields the correct evaluation.

Ambiguity is a challenging problem and even determining whether a given grammar is unambiguous is a computationally unsolvable problem. There exist ad hoc techniques to remove ambiguity from a grammar. In our example, we can redesign the above grammar by introducing new nonterminals:

```
<expr> → <term> | <expr> + <term>

<term> → <factor> | <term> × <factor>

<factor> → (<expr>) | a
```

The intuitive idea is that, in the modified grammar, variable `<expr>` genereates a sum of "terms" and each variable `<term>` generates a product of "factors". The modified grammar generates the same terminal strings (expressions) as the original grammar and it is unambiguous. (The example may be discussed in more detail in class.)

Pushdown automata (Section 10.5)

A grammar is a generative mechanism: the process begins with the start variables and applies rewriting rules to produce a terminal string. On the other hand, when a compiler receives a program (that is, a terminal string) as input and has to process the parse tree based on the program. For this purpose we use a machine model called *pushdown automaton* that is equivalent to a context-free grammar. The parser stage of a compiler will be, roughly speaking, a deterministic pushdown automaton.

A pushdown automaton (PDA) can write symbols on a stack and read them back later. Writing a symbol "pushes down" all the other symbols on the stack. The symbol on the top of the stack can be read and removed ("pop" operation), see Figure **??**.

A pushdown automaton can be depicted as a directed graph. Similarly as for state diagrams, the states are depicted by nodes (circle) in the graph. The arrows labeling the state transitions need to include both the input symbol read and the operation performed on the stack.

According to notational conventions used in our textbook, a transition from state q_1 to state q_2 labeled by

$$c, d \mapsto w$$

indicated that the computation can switch from state q_1 to state q_2 by reading c from the input, popping (removing) d from the top of the stack and pushing string w to the stack. Here c is an individual input symbol or the empty string, d is an individual stack symbol or the empty string, and, w is a string of stack symbols. That is, one operation pops at most on symbol from the stack but an operation can push a string of stack symbols.

A pushdown automaton begins processing the input in the initial state and with the stack contents empty. The computation accepts the input if at the right end of the input the machine is in an accepting state and the stack is empty.

More details of the definition of pushdown automata and examples are given on pages 216–219 in the textbook.

Note. While the definition of state diagrams (DFAs or NFAs) is fairly consistent in different sources, the precise details of a pushdown automaton definition vary depending on different textbooks.[2] In the lectures and assignments we will follow the notational conventions of our

[2]For example, some textbooks use different acceptance conditions for a computation.

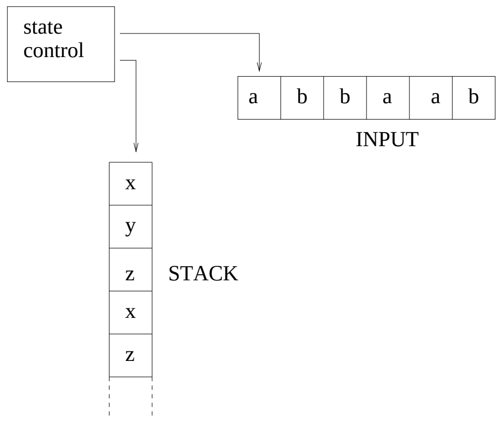

Figure 1: A pushdown automaton

textbook.

Example. Construct a pushdown automaton for the language

$$\{a^i b^i \mid i \geq 0\} \cup \{a^{2i} b^i \mid i \geq 0\}$$

(To be done in class.)

Pushdown automata recognize exactly the class of context-free languages. We will go through the construction showing how to construct a pushdown automaton recognizing the language generated by a grammar. The converse inclusion is omitted in this course.

Pumping lemma for context-free languages (Section 10.6)

Context-free languages form a larger class than regular languages. Context-free grammars have important applications in specifying the syntax of programming languages, however, context-free grammars cannot specify certain programming language feature, like the requirement that variable names need to be declared before their use.

To have an understanding of the limitations of context-free grammars we need a tool for proving that a language is not context-free. The *context-free pumping lemma* is based on similar ideas as our earlier pumping lemma for regular languages. Roughly speaking, the difference is that in the context-free pumping we have to repeat in parallel two substrings.

Pumping Lemma for Context-Free Languages. For every context-free language L there is a constant p such that every string $s \in L$ of length at least p can be written as

$$s = uvwxy$$

where

(i) $v \neq \varepsilon$ or $x \neq \varepsilon$

(ii) $|vwx| \leq p$

(iii) for each $i \geq 0$, $uv^iwx^iy \in L$

Similarly as with regular languages, the use of the context-free pumping lemma is a proof by contradiction. For the sake of contradiction we assume that the language is context-free. Then from the pumping lemma we get the constant p and select a string s of length at leastr p. To derive a contradiction we have to prove that no matter how we try to divide s into five parts $uvwxy$ the parts cannot satisfy the above conditions (i), (ii), (iii).

Because the context-free pumping lemma repeats two substrings and we consider decompositions into five parts, the details of the proof are some times more complicated and involve a case analysis to cover all possible ways to divide s into five parts. The following two examples will be covered in detail in class.

Examples. Show that the following languages are not context-free:

a) $\{a^ib^ic^i \mid i \geq 0\}$

b) $\{ww \mid w \in \{a,b\}^*\}$

Note that above the language $\{ww \mid w \in \{a,b\}^*\}$ abstracts the idea that context-free grammars cannot specify the requirement that variable names have to be declared before their use. Assuming there is no upper limit on the length of a variable name, a grammar specifying the requirement would need to generate strings of the form

$$\ldots w \ldots w \ldots, \text{ for } w \in \Sigma^*$$

where ... refer to other parts of the program that are omitted in the abstraction.

Applications of grammars: RNA secondary structure prediction using context-free grammars

Regular languages and finite state machines are often used in DNA and protein sequence matching problems (see notes for week 1). The use of regular languages is based on the assumption that the mutations that caused variations in the sequence occurred independently of each other.

The situation is different when we consider the secondary (or three dimensional) structure of an RNA molecule. RNA molecules with same secondary structure usually have the same function. Because of the way RNA molecules fold, in order to preserve the secondary structure, variations to nucleotides in one part of the sequence must be matched in the corresponding bonded subsequence (the bonded subsequences are called a *stem.*) This type of nested dependencies can be described using context-free grammars.

We consider a simple example. The RNA nucleotides contain four different bases: adenine (A), guanine (G), cytosine (C), uracil (U). C and G, and A and U, respectively, are complementary. In the below grammar we use lower case letters (a, g, c, u) since they are the terminals of the grammar.

To generate RNA structures we use a grammar (V, Σ, P, S) where $V = \{S\}$, $\Sigma = \{a, g, c, u\}$, and P consists of the productions:

(i) $S \longrightarrow aSu \mid uSa \mid cSg \mid gSc$

(ii) $S \longrightarrow aS \mid cS \mid gS \mid uS$

(iii) $S \longrightarrow Sa \mid Sc \mid Sg \mid Su$

(iv) $S \longrightarrow SS \mid \varepsilon$

A parse tree for the terminal string

$$ccugagaggcaaccuagaaggu \qquad (1)$$

is given in Figure **??**. The parse tree corresponds to the RNA secondary structure with two stems (or bonded subsequences) that is illustrated in Figure **??**.

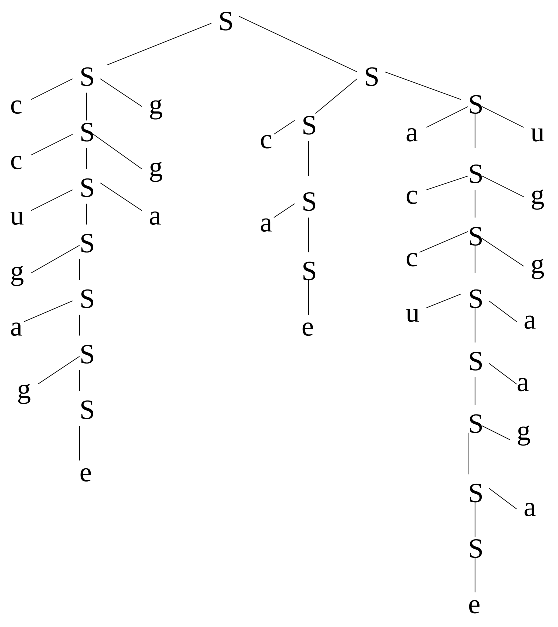

Figure 2: Parse tree for an example RNA structure. In the figure, “e” denotes the empty string ε.

From the parse tree we can read the secondary structure: the application of the grammar rules (i) produces a stem of bonded base pairs. Note that the grammar is *ambiguous*: the same RNA sequence (**??**) has many other parse trees. The ambiguity is, for the most part, caused by the possibility of applying the rules (ii), (iii) in different order. Derivations that differ only in the order of application of the rules (ii) and (iii) correspond to the same secondary structure.

Above we have outlined how context-free grammars can be used to model RNA folding. A more detailed study tells us that complementary pairs (C–G and A–U) are the likeliest

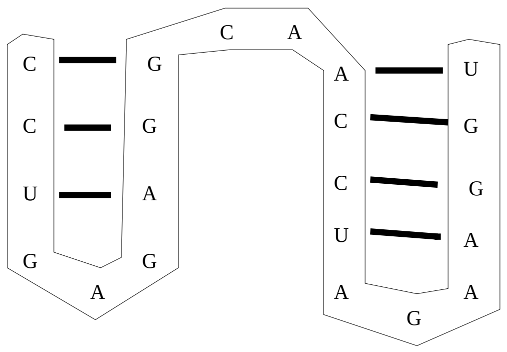

Figure 3: An RNA secondary structure. The thick lines indicate bonds between complementary bases in a stem.

candidates to form base pairs but, sometimes, also other pairs are formed. For more accurate secondary structure prediction the grammar rules need to be augmented with probabilities. The computerized methods used for RNA secondary structure prediction use probabilistic models called *stochastic context-free grammars.* An introductory survey can be found e.g. in [**?**].

References

[1] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, *Biological sequence analysis: Probabilistic models of proteins and nucleic acid.* Cambridge University Press, 1998.

Parsing

This material is covered in Chapter 11 of the textbook.

Parsing is the process of determining if a string of tokens can be generated by a grammar. As discussed in the previous weeks, context-free languages are recognized by pushdown automata. A parser, as discussed in this course, is a deterministic pushdown automaton. While a pushdown automaton just gives a "yes" or "no" answer of an input string, the parsing stage of a compiler actually constructs the parse tree and the parse tree is used in program translation.

Parsing in an important step in the compilation of programming languages. There are two general approaches to do this:

- Attempt to construct reasonably efficient parsers for general context-free languages.
- Define subclasses of context-free grammars which can be parsed more efficiently.

A straightforward "brute-force" parsing method for general context-free grammars systematically tries all possible derivations that could produce the given terminal string. This is *extremely* inefficient (and for grammars with ε-productions the straightforward parsing method does not even need to terminate).

Using a dynamic programming technique we can get a significantly improved parsing algorithm for general context-free grammars, with time complexity $O(n^3)$. However, this is still not good enough for compilers which need to handle very large size programs. Parsing based on a deterministic pushdown automaton can be done in linear time.[1]

In the following we consider a subclass of grammars for which an efficient parser can be constructed in a straightforward way. Roughly speaking, a recursive descent parser associates

[1] Recall that some context-free languages cannot be recognized by a deterministic pushdown automaton.

a procedure to each nonterminal of the grammar. The parse tree is constructed top-down by recursively calling the procedure of the current left-most nonterminal.

We consider a special case of recursive descent parsing, called predictive recursive descent. In predictive parsing the current input token (look-ahead symbol) uniquely determines the procedure chosen for the nonterminal, that is, the first token of the remaining input determines the production chosen for the nonterminal.

A recursive descent (predictive) parser does not explicitly construct the parse tree, although it does so implicitly.

Example. Grammar for balanced strings:

$$< \text{balanced} > \rightarrow < \text{empty} > \mid \ 0 < \text{balanced} > 1$$

$$< \text{empty} > \rightarrow \ \varepsilon$$

The recursive descent parser defines a function `Balanced()` that makes a recursive call:

MustBe(ZERO)

Balanced()

MustBe(ONE)

More details of the construction, including the code for the parser, can be found on page 229 in the textbook. Here “ZERO” and “ONE” are tokens corresponding to the input symbols 0 and 1, respectively. The function MustBe() advances to the next token if the argument matches the lookahead symbol.

Consider the operation of the parser on the input: 000111
Initially we call procedure Balanced(), gettoken() returns ZERO and the switch-statement determines that we execute the sequence

MustBe(ZERO); Balanced(); MustBe(ONE)

Now the symbol 0 is consumed by MustBe(ZERO) and then Balanced() is called again. Again gettoken() returns ZERO which means that we again execute the sequence

MustBe(ZERO); Balanced(); MustBe(ONE)

where MustBe(ZERO) consumes the second symbol 0. After this gettoken() returns ONE and the switch-statement in Balanced() does nothing (simulating the production $<$ balanced $> \rightarrow <$ empty $>$. Finally, the two recursive calls MustBe(ONE) consume the remainder of the input.

Predictive parsing may be performed using a pushdown stack, that is, a deterministic pushdown automaton:

- Initially the stack holds the start nonterminal.
- At each step in the parse, terminal symbols which appear on top of the stack are "popped" and matched with the next input symbols.

  Whenever a nonterminal appears on top of the stack, using lookahead on the input and a <u>parsing table</u>, the parser *predicts* the production that is used to replace the nonterminal.

A recursive-descent parser uses a stack implicitly to implement recursive calls of the functions.

Certain context-free grammars cannot be parsed using predictive recursive-descent. In some cases we may transform the grammar into an equivalent one for which we can use recursive-descent parsing but there exist context-free languages that do not have any grammar that can be parsed using recursive-descent.

Recall that some context-free grammars cannot be recognized by a deterministic pushdown automaton and predictive recursive descent parsing can be done only for a subclass of deterministic context-free languages. We want consider properties of grammars that may prevent the use of predictive recursive-descent parsing. In the following we develop "if-and-only-if" conditions that tell us whether a grammar can be parsed using predictive recursive descent.

The following example illustrates the two types of problems that prevent the use of recursive descent parsing for a given grammar.

Example. A string w is said to be a *palindrome* if w equals its reversal w^R (a.k.a. mirror image). Consider the language of palindromes over a binary alphabet:

$$\{w \in \{0,1\}^* \mid w = w^R\}$$

It is easy to write a grammar for the language of palindromes:

$$< \text{palindrome} > \;\rightarrow\; 0 \mid 1 \mid 0 < \text{palindrome} > 0 \mid 1 < \text{palindrome} > 1 \mid \varepsilon$$

If we try to construct a predictive parser for the grammar, we have the following *problem:* the right sides of different productions for the same variable begin with the same token. Consequently, the grammar cannot be parsed using predictive recursive descent.

The above is, perhaps, the obvious reason that prevents the use of predictive parsing. However, the grammar has another more subtle feature that would also cause problems with predictive parsing. Rules with right side $0 < \text{palindrome} > 0$ means that token 0 can occur directly after variable $< \text{palindrome} >$. Thus, when the next token is 0, the parser could not decide whether to use one of the productions

$$< \text{palindrome} > \;\rightarrow\; 0 \mid 0 < \text{palindrome} > 0$$

or the erasing production.

In fact, it can be shown that the language of palindromes cannot be generated by any context-free grammar for which we can use predictive recursive descent. The proof is beyond the scope of our course.

If we modify the language and instead consider the language of *centered palindromes*

$$\{w\$w^R \mid w \in \{0,1\}^*\}$$

we can avoid the problems. A grammar for centered palindromes avoids the above problems and has a predictive parser:

$$< \text{CenPal} > \rightarrow \ \$ \mid 0 < \text{CenPal} > 0 \mid 1 < \text{CenPal} > 1$$

Naturally modifying the language is not a good solution. What we would want to do is to modify the grammar into an equivalent grammar that is amenable for predictive parsing. In the case of the grammar generating the set of palindromes this is not possible, but later we will consider grammar transformations that, in special cases, allow us to eliminate problems that prevent the use of predictive recursive descent parsing.

As we have discussed, predictive recursive descent parsing cannot be used with all context-free grammars. Next we develop technical conditions that allow us to determine whether or not predictive parsing is possible.

A necessary condition is that the *director sets* associated with any two productions for the same nonterminal must be disjoint.[2] The director sets are defined using "first" and "follow" sets. The technical definition of the first- and follow- sets is below.

We consider a context-free grammar $G = (V, \Sigma, P, S)$ where V is the set of variables, Σ is the set of terminals, P is the set of productions and $S \in V$ is the start variable.

The first-sets are defined for a string of variables and terminals. For $\alpha \in (V \cup \Sigma)^*$ the set

$$\underline{\text{first}(\alpha)} \subseteq \Sigma \cup \{\varepsilon\}$$

[2]The description of these conditions on page 238 in the textbook contains minor inaccuracies. Please see the corrections posted on the textbook's web site (a link to the corrections can be found on CISC/CMPE-223 homepage). The below discussion follows the corrected version.

consists of all terminals b that can begin a string derived from α. Additionally, if α derives the empty string, ε is in first(α).

The follow-sets are defined only for individual variables. For a variable $N \in V$, the set

$$\underline{\text{follow}(N)} \subseteq \Sigma \cup \{\text{EOS}\}$$

consists of all terminals that can appear immediately to the right of N at some stage of any derivation. The pseudo-terminal EOS is used to denote the end of the input and EOS $\in$ follow(N) if N may appear as the rightmost symbol of some string that occurs in a derivation.

Note. The definitions of first-sets and follows-sets are not symmetric.

- To determine the first-set of a string α we consider all derivations beginning with α.
- To determine the follow-set of a variable N we consider derivations beginning with the start variable and check which terminals can occur directly to the right of N in some sentential form occurring in the derivation.

Example. To become familiar with first- and follow-sets we consider a simple example. Let $V = \{S, A\}$, $\Sigma = \{a, b, c\}$ and the grammar has the following productions:

$$S \rightarrow aAa \mid bAa \mid \varepsilon$$

$$A \rightarrow cA \mid bA \mid \varepsilon$$

Determine what are the following sets (to be done in class):

- first(S) $= \ldots?$
- first(A) $= \ldots?$
- first(Aa) $= \ldots?$
- follow(S) $= \ldots?$

- follow$(A) = \ldots ?$

Now we can define the *director sets* of productions. Let

$$N \rightarrow w_1 \mid w_2 \mid \ldots \mid w_n$$

be all the productions for a variable N. The *director set* of the production $N \rightarrow w_i$, $1 \leq i \leq n$, consists of the following:

- The set first(w_i).
- If the empty string can be derived from w_i, the director set additionally contains follow(N).

In order to be able to use recursive descent parsing[3], the grammar must satisfy the condition that the director sets for different productions for the same nonterminal must be disjoint.

The conditions characterizing grammars that allow recursive descent parsing can be formulated more directly as follows:

A grammar can be parsed using recursive descent if for any two productions having the same nonterminal on the left side

$$N \rightarrow \alpha \mid \beta$$

the following conditions hold:

(i) No terminal $b \in \Sigma$ can begin both a string w_1 derived from α and a string w_2 derived from β.

(ii) At most one of α and β can derive the empty string ε.

(iii) If $\beta \Rightarrow^* \varepsilon$ then first$(\alpha) \cap$ follow$(N) = \emptyset$.

[3]Here by recursive descent we always mean predictive recursive descent with "look-ahead one". In more advanced courses you may encounter parsing algorithms that use a longer look-ahead string.

Conditions (i) and (ii) together means just that first(α) $\cap$ first(β) $= \emptyset$.

Note that in condition (iii) the role of α and β is symmetric, that is, if α derives the empty string then first(β) and follow(N) must be disjoint.

In simple examples, like the one seen above, we can determine the "first" and "follow" sets by hand. In real-life situations we must use an algorithm to compute these sets, see for instance the text on compilers by Aho, Sethi and Ullman mentioned on page 241 in our textbook.

Example. Consider a grammar with the following productions. Here S is the start variable, A, B are variables and the set of terminals is $\{a, b, c, d\}$.

$$S \to BaA$$

$$A \to aA \mid Bc \mid \varepsilon$$

$$B \to bB \mid cBa \mid \varepsilon$$

Does this grammar allow the use of recursive descent parsing?

If the grammar does not satisfy the above criterion, i.e., predictive recursive descent parsing is not possible directly based on rules of the grammar, it may be possible to transform the grammar into an equivalent grammar for which we can use recursive descent. Below we describe some commonly used transformations.

Grammar transformations

Perhaps the most obvious situation preventing is where some variable has two or more productions where the right side has a common prefix. If the grammar contains productions

$$A \ \to \ \alpha\beta_1 \ \mid \ \alpha\beta_2$$

where $\alpha \neq \varepsilon$, then the sets first($\alpha\beta_1$) and first($\alpha\beta_2$) are (generally) not disjoint and consequently also the director sets of the productions $A \to \alpha\beta_1$ and $A \to \alpha\beta_2$ are not disjoint.

Left factoring is a transformation that attempts to fix the above problem. Consider productions

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_m \mid \gamma_1 \mid \ldots \mid \gamma_n$$

where $\alpha \neq \varepsilon$ is not a prefix of $\gamma_1, \ldots, \gamma_n$ and, furthermore, $\beta_1, \ldots, \beta_m$ do not have any common prefix.

Then we replace the above productions by

$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_n$

$A' \rightarrow \beta_1 \mid \ldots \mid \beta_m$

where A' is a new nonterminal. (Note that A' must not be used anywhere else in the grammar because otherwise the modified grammar might not be equivalent with the original grammar.) We can repeat the transformation until no two alternatives for a nonterminal have a common prefix.

Example 1. Consider the grammar with S as the only variable:

$S \rightarrow abSa \mid abcSc \mid bcc \mid abdc$

Apply left-factoring to this grammar.

Note: The above left factoring algorithm always terminates and produces a grammar where the "immediate problem" has been fixed, that is, no nonterminal has two productions where the right sides have a common nonempty prefix. However, the left factoring method *does not* always produce a grammar that can be used for predictive recursive descent parsing.

Example 2.

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

This grammar illustrates the ambiguity in if-statements with optional "else" parts. Here S is a nonterminal for "statement", E is a nonterminal for "expression", and i (respectively, t, e) stands for "if" (respectively, "then", "else"). Applying left-factoring to the above grammar removes the immediate problem but does not yield a grammar suitable for predictive recursive descent parsing.

If-statements with optional else-parts is a notoriously difficult programming language construct to handle during parsing. Typically parsers enforce some ad-hoc rule like "an 'else' should be matched with the closest unmatched 'if' ".

Left-recursive productions

Left-recursive productions have the form $A \to A\alpha$. These may cause a recursive-descent parser to go into an infinite loop. Consider our earlier example of a grammar for simple expressions:

```
<expr> → <term> | <expr> + <term>

<term> → <factor> | <term> × <factor>

<factor> → (<expr>) | a
```

On the basis of the next terminal symbol there is no way to determine the production to be used. The rules can be modified as follows:

```
<expr> → <term> <expTail>

<expTail> → + <term> <expTail> | ε
```

Here `<expTail>` is a new variable. We use a similar transformation for productions for `<term>`.

The above idea inspires a general method to eliminate left-recursion. Suppose we have productions

$$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n \tag{1}$$

where the strings β_i do not begin with A and $\alpha_j \neq \varepsilon$[4], $j = 1, \ldots, m$. We replace (1) by productions

$$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

Example 3. Consider a simple grammar with only one variable S:

$$S \rightarrow Sa \mid Sbc \mid cc \mid \varepsilon$$

The grammar is obviously left-recursive. Apply the transformation to eliminate left-recursion.

When the above transformation goes though all the different variables, this method eliminates all immediate left recursion in the grammar. However, the above method does not handle left recursion involving two or more derivation steps, that is, if we have a situation

$$A \Rightarrow B\beta \Rightarrow \ldots \Rightarrow A\alpha, \quad B \neq A.$$

There is a general algorithm to eliminate also above type of multi-step left recursion, but we do not discuss it here.

[4]What should we do is $\alpha_j = \varepsilon$ for some j?

Specifying Algorithms

We now continue with Part A of the textbook. Material in this document is from chapter 1 and sections 2.1 - 2.7.

Stages in software construction include requirements, specification and implementation. Verification consists of checking that the program follows the specification.

- Specifications describe transformations from input values to output values.
- A program transforms input values to output values in a particular way.

A specification consists of the following parts:

1. What the input will be.
2. What the outputs should be.
3. Environment in which the specification/program should work.

Inputs and outputs typically refer to things that can be observed:

- set of input variables
- set of output variables
- constraints on the values that the variables can take

Formalization of a specification consists of the following:

- declarative interface: static properties of the identifiers
- pre-conditions: assertion on input values that the program will be given

- post-condition: assertion on output values, possibly in relation to the same input values

The declarative interface can specify which values should not be changed by the code (these are declared as constants).

We can say that a specification is a contract: the software designer agrees to establish the post-condition if the program is started in a way that satisfies the pre-condition.

If the program is run in a context not covered by the pre-condition, it can run in any way without “breaking” the contract.

Note: In the following, we will develop logic-based techniques to verify correctness of small programs (algorithms), that is, we want to prove that a program does what a specification says it should do.

The techniques discussed here are, in general, too time consuming to be directly used with large software systems. Formal methods[1] used for verification encompass various software tools for reasoning about correctness. Such tools implement algorithms, typically, from theorem proving or model checking. The algorithms and their use require an understanding of foundations of program correctness discussed in Chapters 1–4 of the textbook.

We use logical formulas, called assertions, as comments in programs. We are asserting that the formula should be true when flow of control reaches it. The assertions are allowed to contain notation that cannot be used in Boolean expressions in programs, see Section 1.3 in the text. In particular, the assertions may include quantified formulas. It will be important to recall the notions of *free variables* and *bound variables* in quantified formulas.[2]

The assertions are not evaluated during program execution, but would be true if evaluated. The notation

[1]You will learn more about formal methods in 4th year.

[2]A very brief review will be done in class. If you do not recall the notions, please review material from CISC-204.

```
ASSERT(P)
S
ASSERT(Q)
```

where S is a sequence of program statements, has the following interpretation:

- If execution of S is begun in a state satisfying P, then it is guaranteed to terminate (in a finite amount of time) in a state satisfying Q.

The above condition is called total correctness.

The correctness statement

```
ASSERT(P)
S
ASSERT(Q)
```

is said to be partially correct if always when execution of S is begun in a state satisfying P it does not terminate (normally) in a state not satifying Q. Note that with partial correctness we do not exclude the possibility of nontermination, runtime errors, etc.

Consider assertions `P` and `Q`. If `P` implies `Q`, we say that "`P` is stronger than `Q`" or "`Q` is weaker than `P`". When the terms "stronger" and "weaker" have the above meaning, note that any assertion `P` is stronger (or weaker) than itself. A general rule is that the pre-condition may be strengthened and the post-condition may be weakened:

(`P {S} Q` and `[P' => P]`) implies `P' {S} Q`

(`P {S} Q` and `[Q => Q']`) implies `P {S} Q'`

Above `P {S} Q` is an abbreviation for the correctness statement

```
ASSERT(P)
S
ASSERT(Q)
```

Example. To illustrate the notion of correctness statements and their validity we consider the specification for array-searching code and some programs for array searching as presented in textbook Section 2.1.

We begin verification from simplest possible program fragments by considering simple assignment statements. The first idea could be, perhaps, to reason in the forward direction but this turns out to be inconvenient.

Axiom scheme:

```
V==I { V = E; } V == [E](V ↦ I)
```

Here `[E](V ↦ I)` is the expression obtained from E by replacing occurrences of V by I.

Required side condition above: identifier I must be distinct from variable V. Why is this needed?

Example.

```
x==7 { x = 6*x + 15; } x==57
```

The use of the above scheme is restricted because the precondition has to be an equality test for the left side of an assignment.

A simpler and more general approach does the reasoning backwards. Based on the post-condition determine the most general pre-condition that guarantees the post-condition holds after the assignment.

Hoare's axiom scheme for assignments:

```
[Q]( V ↦ E ) { V = E; } Q
```

See section 2.3 for a detailed explanation of the symbols. For post-condition Q, Hoare's axiom scheme gives the most general pre-condition that guarantees that post-condition holds after assignment.

Note that in Hoare's axiom scheme the post-condition can be any assertion.

Example.

```
x-y >= 0 { x = x-y; } x >= 0
```

Example. Verify, using also pre-condition strengthening,

```
ASSERT( y === 3 )
x = y-1;
ASSERT( x >= 2 )
```

Issues concerning substitutions

When substituting an expression E for an identifier I we should take care of the following:

1. Add parentheses when necessary.
2. Only free occurrences of I in the assertion are to be replaced by E.
3. As a result of the substitution free occurrences of an identifier in E should not become bound. To prevent this, the names of bound identifiers in the assertion can be changed, when necessary. (The names of free variables cannot be changed.)

Example. What is the result of the below substitution?

```
[ ForAll(x) Exists(y) x < z implies x < y < z ]( z ↦ x + 1 )
```

Note that bound occurrences of x should be replaced by some "fresh" identifier.

The inference rule for *statement sequencing* is:

$$\frac{\mathtt{P}\{\mathtt{C}_0\}\mathtt{Q} \quad \mathtt{Q}\{\mathtt{C}_1\}\mathtt{R}}{\mathtt{P}\{\mathtt{C}_0\mathtt{C}_1\}\mathtt{R}}$$

A limitation is that the post-condition of the first correctness statement has to be the same as the pre-condition of the second correctness statement.

Using the fact that a pre-condition may be strengthened (or post-condition weakened), we see that also the following more general inference rule is valid:

$$\frac{\texttt{P}\{\texttt{C}_0\}\texttt{Q} \quad \texttt{Q}'\{\texttt{C}_1\}\texttt{R} \quad \texttt{Q implies Q}'}{\texttt{P}\{\texttt{C}_0\texttt{C}_1\}\texttt{R}}$$

Proof tableaux

A formal proof with respect to a set of axioms and inference rules consists of a sequence of statements where each statement is an axiom or a conclusion of one of the inference rules all of whose premises have already been proved.

In program verification we use *proof tableaux* as short hand notation for formal proofs of correctness. A proof tableau consists of program code, pre- and post-conditions and *intermediate assertions.* Axioms are instances of the Hoare axiom scheme. The correctness of the intermediate assertions is implied by pre-condition strengthening (post-condition weakening), the inference rule for statement sequencing, and inference rules for conditional statements and loops (to be discussed in the following). Since Hoare axiom scheme works "backwards", a proof tableau is typically constructed starting from the bottom.

Example. (Exercise 2.26, p. 46) Construct proof tableau for

```
ASSERT( x == x0 && y == y0 )

x = x-y;

y = y+x;

x = y-x;

ASSERT( x == y0 && y == x0 )
```

Example. Construct proof tableau for

```
ASSERT( x >= 2 || x <= -2 )

y = 2*x;

z = x-y;

y = x+z;

ASSERT( x+y >= 1 || z >= 2 )
```

Verifying Algorithms

We continue the discussion how to validate correctness statements for the central programming language constructs. Next we consider conditional statements and loops, see Sections 2.8 and 2.9 in the textbook.

The inference rule for if-statements is as follows:

$$\frac{\mathtt{P\&\&B}\{\mathtt{C_0}\}\mathtt{Q} \quad \mathtt{P\&\&!B}\{\mathtt{C_1}\}\mathtt{Q}}{\mathtt{P}\{\text{if}(\mathtt{B})\mathtt{C_0} \text{ else } \mathtt{C_1}\}\mathtt{Q}}$$

Here we need the side condition that the evaluation of `P` should not change the state.

Example. Verify the validity of the following correctness statement:

```
ASSERT( y==y0 && z==z0)
if (y <= z) { y=z+1; z=z+2; } else { z=y+2; y=y+1;}
ASSERT( max( y0, z0) < y < z )
```

The inference rule of if-statements without an else-part is obtained by choosing C_1 to be an empty code fragment.

$$\frac{\mathtt{P\&\&B}\{\mathtt{C_0}\}\mathtt{Q} \quad \mathtt{P\&\&!B}\{\}\mathtt{Q}}{\mathtt{P}\{\text{if}(\mathtt{B})\mathtt{C_0}\ \}\mathtt{Q}}$$

How can we make the 2nd premise of the inference rule more understandable?

Example. Verify

```
ASSERT( z <= y )
if ( x > z || y > x ) { w = z-1; x = y; }
ASSERT( w <= z <= y <= x )
```

Handling loops is more difficult. At first sight it could seem that we have to treat each iteration separately and use large numbers of intermediate assertions. A single well-chosen assertion is normally sufficient.

The inference rule for while statements is:

$$\frac{\texttt{I\&\&B}\ \{\texttt{C}\}\ \texttt{I}}{\texttt{I}\ \{\texttt{while(B)C}\}\ \texttt{I\&\&!B}}$$

Above assertion I is called the loop invariant and it occurs four times in the inference rule. Below we discuss justification for this rule, for more details see Section 2.9.

```
/* other part of the program */
                                <-- (1)
while(B) {
              <-- (2)
           C
              <-- (3)
}
                           <-- (4)
/* rest of the program */
```

(1) Here the program is in a state satisfying the pre-condition `I`.

(2) Here the program is in a state satisfying the assertion `I&&B`.

(3) Here `B` may or may not be true.

(4) Here the program is in a state satisfying `I&&!B` (assuming the code C preserves the assertion `I`.

Hence a proof tableau schema can be written as:

```
/* other part of the program */


ASSERT(I)
while(B) {
         ASSERT(I && B)


         C


         ASSERT(I)
}
ASSERT(I && !B)


/* other part of the program */
```

Similarly as in the inference rule for conditional statements, above we need to assume that evaluating the expression `B` does not change the state.

The essential question in loop verification is choosing a suitable loop invariant. Roughly speaking, the invariant has to satisfy the following:

- the invariant is true before entering the loop,
- the loop body preserves the invariant, and,
- the invariant together with the negation of the while-condition should imply the post-condition.

Example. Choose a suitable loop invariant and complete the proof tableau for the following correctness statement:

```
ASSERT(true)
i = 0;
j = 100;
while( i <= 100 ) {
            i = i+1;
            j = j-1;
}
ASSERT( i == 101 && j == -1)
```

In class we will go through further examples of choosing a loop invariant and completing a proof tableau for program fragments involving loops.

Termination

Above we have discussed correctness statements of the form

```
P { while(B) C } Q
```

However, the corresponding inference rule establishes only that if the program is started in a state satifying `P`, then always when it terminates the state satisfies assertion `Q`. That is, we cannot exclude the possibility of non-termination and thus the inference rule establishes only *partial correctness.*

In order to prove *total correctness,* we need to prove that the execution of the loop always terminates. A typical way to do this is to use a suitable "variant" integer expression that is

- strictly decreased (respectively, increased) by each iteration of the loop, and
- must remain greater than a given lower bound (respectively, less than some upper bound).

Example. The following code to compute powers is partially correct. How should the pre-condition and the invariant be modified in order to guarantee total correctness.

```
ASSERT( n >= 0 )
i = 0; y = 1;
while ( i != n )
INVAR( i >= 0 && y == power(x, i))
{
  y = y*x*x;
  i = i+2;
}
ASSERT( y == power(x, n))
```

Uncomputability of termination. There is *no general algorithm* to determine whether or not a given (while) loop terminates. We will come back to this question at the end of the course when discussing computability. It may be interesting to note that there exist very innocent looking while-loops for which *no one knows* whether or not the loop terminates for all variable values, for example, see Program 2.7 on page 55 in the text.

Summary of steps needed to verify while-loops

1. Select an invariant and show that the loop preserves the invariant.
2. Show that the invariant holds before entering the loop.
3. Show that the invariant and the negation of the loop condition imply the post-condition (possibly after some finalizing assignments).
4. Show that the loop terminates. For well-designed while-loops[1] often can include bounds for the loop variable in the invariant.

Additional Verification Techniques.

Next we discuss verification of for-loops. This material is from Chapter 4 in the textbook.

In order to avoid "re-inventing the wheel", we will rewrite a for-loop

$$\texttt{for}(\texttt{A}_0, \texttt{B}, \texttt{A}_1)\texttt{C}$$

as a while-loop

$$\texttt{A}_0; \texttt{while}(\texttt{B})\texttt{CA}_1;$$

[1]In general, termination is uncomputable. This will be discussed next week.

Now the proof tableau schema looks as follows:

```
ASSERT( P )  //pre-condition

...

A_0 ;

ASSERT(I)   // I is the invariant

while (B) {

     ASSERT( I && B )

     CA_1;

     ASSERT(I)

     } //end while

ASSERT( I && !B )

     ...

ASSERT(Q)    // post-condition
```

As an example we verify the partial correctness of the following:

```
ASSERT(0 <= n <= max)

{ int i;

  for (i=0; A[i] != x && i < n; i++)

    {}

  present = i<n;

}

ASSERT(present iff x in A[0:n-1])
```

Note that the code may not terminate normally if x does not occur in `A[0:n-1]`. Why?

As the loop invariant (denoted as `I`) we choose:

```
0<=i<=n && ForAll(k=0; k<i) x != A[k]
```

Using the schema for for-loops, we must verify the following:

```
ASSERT(pre-condition)
i=0; /* initial assignment */
ASSERT(I) /*loop invariant*/
while( A[i] != x && i < n) {
   ASSERT(I && A[i] != x && i < n)
   /* for-loop has empty body*/
   i++;
   ASSERT(I)
}
ASSERT(I && !(A[i] != x && i < n))
present = i<n;  /*final assignment*/
ASSERT(post-condition)
```

The complete construction is given in class. Here we have to be a little careful in how the post-condition is established from the invariant and the negation of the loop condition.

Array component assignment rule

To deal with array component assignments we need to modify the Hoare axiom schema so that instead of substituting individual array components we modify the entire array.

The notation `(A | I` $\mapsto$ `E)` refers to an array obtained from `A` by replacing the value at position `I` by the value of the expression `E`.

More formally,

$$(\mathtt{A}|\mathtt{I} \mapsto \mathtt{E})[\mathtt{I}'] = \begin{cases} \mathtt{E} \text{ when } \mathtt{I}' = \mathtt{I}, \\ \mathtt{A}[\mathtt{I}'] \text{ when } \mathtt{I}' \neq \mathtt{I}. \end{cases}$$

Now the modified array component assignment rule can be written as:

$$[\mathtt{Q}](\mathtt{A} \mapsto \mathtt{A}') \;\; \{\mathtt{A}[\mathtt{I}] = \mathtt{E};\} \;\; \mathtt{Q}$$

where `A'` is `(A | I` $\mapsto$ `E)`.

It has to be verified separately that the value of `I` is within the subscript range of the array `A`.

Example. Use the array component assignment axiom (two times) to find the weakest sufficient pre-condition `P` for the following code fragment

```
ASSERT( P )
A[i] = x;
A[k] = 5;
ASSERT( A[j] == 0 )
```

Above `x` is an integer variable, `A` is an array of integers and we assume that all the subscripts are within the range of subscripts for `A`.

Example. We consider an array of even length.

The program should move elements from even numbered positions to a contiguous chunk at the beginning, see Figure 1.

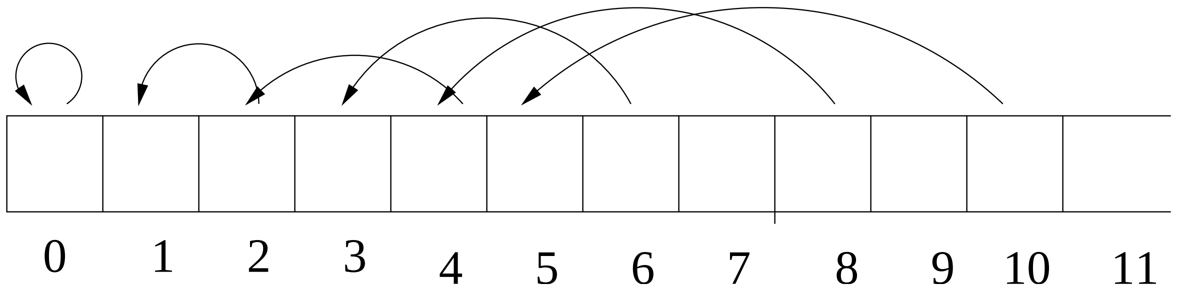

Figure 1: Moving of the array elements.

The specification for the program is as follows:

```
Interface:  const int n;
            Entry A[2n]; /*entries numbered 0,...,2n-1 */
```

```
Pre-condition:  n >= 1 && A == A0

Post-condition: ForAll(i=0; i<n) A[i] == A0[2i]
```

On the basis of the post-condition we can select a suitable loop invariant and using it "derive" the program. (To be done in class.)

UNIMPLEMENTABLE SPECIFICATIONS

This material is covered in Chapter 12 of the textbook.

There exist specifications that cannot be implemented because the specified function is *uncomputable.* This is possible even for well-defined and consistent specifications!

The Halting Problem

We want to implement a function HALTS that receives file parameters `func` and `arg` and

1. HALTS returns `true` if the file `func` contains a function definition with one file parameter, and the function terminates if applied to file `arg`.
2. HALTS returns `false` otherwise.

Theorem. HALTS cannot be implemented in C
(that is, the function HALTS is uncomputable).

The function HALTS cannot be implemented in any (currently used) programming language. More generally, it is not *effectively computable* (or algorithmically computable).

In a general setting, a computational problem consists of

1. a potentially infinite set of inputs, and
2. a function that associates an output to each input.[1]

Since computers are finite, we usually require that each individual input has a finite representation, that is, can be encoded as a finite string. Similarly, each individual output is

[1]In some computatitional problems, for a given input we may allow several acceptable otuputs. To formalize this kind of computational problems we need to use a relation instead of a function.

required to have a finite representation. This is not a restriction in practice since all natural computational problems can be encoded in this way.

A *solution* to a computational problem is then an algorithm that for each input finds (computes) the correct output.

The fact that the function HALTS is uncomputable is not caused by any "defect" in the programming language C. Exactly the same argument works with any other programming language!

- It has been shown that all the standard programming languages are computationally equivalent in the sense that the same class of functions can be implemented in each of them.
- Furthermore, the functions that can be implemented in some programming language (C, Java, ...) are exactly the functions that can be computed by so called Turing machines. Turing machines are a formal model of algorithms originally developed by Alan Turing in the 1930's.

Church-Turing thesis: The functions computable by Turing machines are exactly all the functions that can be effectively computed by some informal algorithm.

- It is generally believed that the Church-Turing thesis holds. However, it is impossible to prove that it is correct since the notion of "informal algorithm" is not precisely defined.

Note: The fact that Turing machines compute the same functions that can be implemented in C (or Java, Lisp, etc.) can be proved. This also establishes that the different programming languages can implement exactly the same class of functions.

The technique we used to show that HALTS is uncomputable[2] is called diagonalization.

The technique of diagonalization allows us to establish the uncomputability of specific algorithmic problems. Comparing the "numbers"[3] of all possible programs and of all algorithmic problems, we can establish that, in fact, "most" algorithmic problems are unsolvable. All possible programs can be listed as an infinite unending list P_1, P_2, ... On the other hand, the set of algorithmic problems is "as large" as the set of real numbers.

Reducibility

Since the diagonalization argument is fairly complicated, it would not be convenient to use it directly for each new computational problem we suspect to be unsolvable. Once we have some problem (function) that is known to be uncomputable, we can show other problems to be uncomputable using a technique called reduction.

An algorithmic problem A is said to be reducible to a problem B, if a solution (algorithm) for B can be used to solve A.

When problem a problem A is reducible to problem B, we can also say that

- A is easier than or as hard as B, or,
- A is not harder than B.

Now the following holds:

1. If A is uncomputable, and
2. A is reducible to B

[2]or unsolvable

[3]In more precise terminology, we compare the cardinality of the set of all programs and the cardinality of the set of algorithmic problems.

then also B is uncomputable.

Why?

Thus, if problem A is reducible to problem B, the following three situations are possible:

- A and B are are both computable,
- A is computable and B is uncomputable,
- both A and B are uncomputable.

Note that if A is reducible to B, it is not possible that B is computable and A is uncomputable.

The above general definition of reducibility can be used to establish algorithmic uncomputability. For more precise comparisons of the hardness (or complexity) of algorithmic problems we use *mapping reductions* that transform instances of one problem to instances of another problem.

Rice's theorem

For computing problems related to program testing, we can distinguish between *syntactic* and *semantic* problems. Syntactic problems are, roughly speaking, related to the correct representation of a program in the formalism of a given programming language, and these problems are usually algorithmically solvable. Semantic problems are related to the meaning of programs. Examples of semantic problems include:

- Does a program accept a given input? (that is, produce the answer "yes" for a given input)
- Does a program satisfy a given specification?
- Does a program halt on a given input, or does a program halt on all inputs?

By a *decision problem* we mean a problem where the answer is simply “yes” or “no”.

The so called *Rice’s theorem* says that all non-trivial semantic decision problems for programs are unsolvable.

A (semantic) decision problem is said to be non-trivial if there is an input program for which the answer is “yes” and another input program for which the answer is “no”. Note that the requirement is very weak, and any decision problems considered in practice are non-trivial in the above sense.

For trivial decision problems the answer is the same for all inputs. What would be an example of a trivial semantic problem?