

FLAC

Context-Sensitive Grammars

Klaus Sutner
Carnegie Mellon University

Fall 2019



1 Context-Sensitive Grammars

- Linear Bounded Automata

- LBA and Counting

As we have seen, r/w-systems are fairly complicated.

Is there a way to restrict the systems a bit, and get them to behave a little better?

One useful idea taken from the study of grammars of ordinary languages is to partition the alphabet into **terminals** and **syntactic variables**: in the end, we are only interested in strings of terminals, but, in order to organize derivations, we also introduce some syntactic variables.

For example, concepts such as *sentence*, *verb*, *object* and so on are useful to describe the structure of an English sentence.

Using r/w-systems to explain the structure of natural languages is quite a challenge—one might suspect that the approach is ultimately doomed to failure, it just does not seem to handle semantics sufficiently well.

But nowadays there is another class of languages that is highly important, and nowhere near as complicated as their natural counterparts: **programming languages**.

Note that semantics are critical here: there cannot be any ambiguity as to what computation a program describes. Well, at least there should not . . .

The old Unix warhorse `dc` also uses RPN.

n

10 20 30

f

30

20

10

^

n

1073741824000000000000000000000000000000

Many programming languages have a block structure like so:

```
begin
  begin
    end
  begin
    begin
      end
    begin
      end
    end
  end
end
```

Here is a more complicated property of some programming languages: variables must be declared before they can be used.

begin

int x;

...

x = 17;

...

end

This will turn out to be much trickier.

We could build general r/w-systems for programming languages, but then it would be difficult (of impossible) to produce parsers. Instead, we introduce a number of restricted r/w-systems called a **grammars** that will provide useful descriptions for programming languages—and lots of other combinatorial objects.

Again: the main idea will be that replacements can only be made on syntactic variables, but the final result will be a string of terminals.

For all our grammars, we will make sure to understand the corresponding machines that solve the recognition problem.

Definition

A (formal) grammar is a quadruple

$$G = \langle V, \Sigma, \mathcal{P}, S \rangle$$

where V and Σ are disjoint alphabets, $S \in V$, and $\mathcal{P}$ is a finite set of productions or rules (to be explained shortly).

- the symbols of V are (syntactic) variables,
- the symbols of Σ are terminals,
- S is called the start symbol (or axiom).

We often write $\Gamma = V \cup \Sigma$ for the complete alphabet of G .

Note that the derivability relations $\rightarrow$ and $\xrightarrow{*}$ still make sense in this setting, we can think of a formal grammar as a particular type of r/w-system.

A string $\alpha \in \Gamma^*$ is a **sentential form** if $S \xrightarrow{*} \alpha$.

The language of G is the collection of sentential forms that consist only of terminals:

$$\mathcal{L}(G) = \{ x \in \Sigma^* \mid S \xrightarrow{*} x \}$$

Note that $\mathcal{L}(G)$ is automatically semidecidable: we can systematically enumerate all possible derivations and check if one of them ends in x .

So the first real question is: is $\mathcal{L}(G)$ decidable?

- $A, B, C \dots$ represent elements of V ,
- $S \in V$ is the start symbol,
- $a, b, c \dots$ represent elements of Σ ,
- $X, Y, Z \dots$ represent elements of Γ ,
- $w, x, y \dots$ represent elements of Σ^* ,
- $\alpha, \beta, \gamma \dots$ represent elements of Γ^* .

Without further restrictions, a grammar is really no better than a rewrite system. In a **phrase-structure grammar** we impose the following rather mild restriction on productions:

$$\pi : \alpha \rightarrow \beta \quad \text{where } \alpha \in \Gamma^* V \Gamma^*, \beta \in \Gamma^*$$

So every replacement handle must contain a variable, but we clobber the whole string and replace it by β .

This is a classical case of lipstick on a pig, and no real progress over general r/w-systems.

Theorem

A language is phrase-structure if, and only if, it is semidecidable.

Proof.

Suppose G is a phrase-structure grammar for L .

Clearly we can construct a Turing machine that on input $x \in \Sigma^*$ starts to enumerate all possible derivations in G (in some order, say, length-lex).

But then the language of G is recursively enumerable and thus semidecidable.

For the opposite direction consider some deterministic one-tape Turing machine, say, $M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_H \rangle$, that accepts L .

Let

$$V = \Sigma_{\varepsilon} \times \Gamma \cup Q \cup \{S, T, U\}$$

Here S , T and U are new symbols and everything is disjoint.

We may safely assume that all transitions are of the form $\delta(p, C) = (q, D, \pm 1)$ (i.e., the tape head always moves). Also, we may assume that the TM never moves to the left beyond the original input (i.e., uses a one-way infinite tape).

Batten down the hatches.

$$S \rightarrow q_0 T$$

$$T \rightarrow \begin{pmatrix} a \\ a \end{pmatrix} T \quad a \in \Sigma$$

$$T \rightarrow U$$

$$U \rightarrow \begin{pmatrix} \varepsilon \\ \underline{b} \end{pmatrix} U \quad \underline{b} \text{ blank symbol}$$

$$U \rightarrow \varepsilon$$

$$p \begin{pmatrix} a \\ C \end{pmatrix} \rightarrow \begin{pmatrix} a \\ D \end{pmatrix} q \quad \delta(p, C) = (q, D, 1)$$

$$\begin{pmatrix} b \\ E \end{pmatrix} p \begin{pmatrix} a \\ C \end{pmatrix} \rightarrow q \begin{pmatrix} b \\ E \end{pmatrix} \begin{pmatrix} a \\ D \end{pmatrix} \quad \delta(p, C) = (q, D, -1)$$

$$\begin{pmatrix} a \\ C \end{pmatrix} q_H \rightarrow q_H a q_H \quad \text{halting state}$$

$$q_H \begin{pmatrix} a \\ C \end{pmatrix} \rightarrow q_H a q_H$$

$$q_H \rightarrow \varepsilon$$

One should think of this as a two-track alphabet $\left(\begin{smallmatrix} a \\ b \end{smallmatrix}\right)$ plus some big symbols that take up both tracks.

Block 1: produce a string of the form

$$\begin{array}{cccccccc} & a_1 & a_2 & \dots & a_n & \varepsilon & \varepsilon & \dots & \varepsilon \\ q_0 & a_1 & a_2 & \dots & a_n & \underline{b} & \underline{b} & \dots & \underline{b} \end{array}$$

Block 2: Run the computation of the Turing machine on the second track; if the padding is long enough, we can accommodate any convergent computation.

Block 3: If the accepting state appears, erase the bottom track and leave the top track (the original input). Done.



A slight less chaotic type of grammar forbids the use of terminals on the left hand side of a production.

A **type-0 grammar** has productions of the form

$$\pi : \alpha \rightarrow \beta \quad \text{where } \alpha \in V^+, \beta \in \Gamma^*$$

Type-0 grammars and phrase-structure grammars are essentially the same: just replace terminal a on the LHS by A_a and add rules $A_a \rightarrow a$.

Definition (CSG)

A **context-sensitive grammar (CSG)** or a **type-1 grammar** is a grammar where all productions are of the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta \quad \text{where } \gamma \neq \varepsilon$$

Some authors also allow $S \rightarrow \varepsilon$ in which case S may not appear on the righthand side of any production. A language is context-sensitive if it can be generated by a context-sensitive grammar.

Note the constraint that the replacement string $\gamma \neq \varepsilon$; as a consequence we have

$$\alpha \xrightarrow{*} \beta \quad \text{implies} \quad |\alpha| \leq |\beta|$$

This will be important for the decision algorithm.

Lemma

Every context-sensitive language is decidable.

Proof.

Suppose $w \in \Sigma^*$ and $n = |w|$. In any potential derivation $(\alpha_i)_{i < N}$ we have $|\alpha_i| \leq n$.

So consider the digraph D with vertices $\Gamma^{\leq n}$ and edges $\alpha \rightarrow_P \beta$.

Then w is in L iff w is reachable from S in D .

□

Of course, the size of D is exponential, so this method won't work in the RealWorld.

Claim

Not all decidable languages are context-sensitive.

Here is a cute diagonalization argument for this claim.

Let $(x_i)_i$ be an effective enumeration of Σ^* and $(G_i)_i$ an effective enumeration of all CSG over Σ (say, both in length-lex order). Set

$$L = \{x_i \mid x_i \notin \mathcal{L}(G_i)\}$$

Clearly, L is decidable.

But L cannot be context-sensitive by the usual diagonal mumbo-jumbo.

Here is one typical example of a context-sensitive language.

$$L = \{ a^n b^n c^n \mid n \geq 1 \}$$

A context-sensitive grammar G for L :

let $V = \{S, B\}$ and set

$$S \rightarrow aSBc \mid abc$$

$$cB \rightarrow Bc$$

$$bB \rightarrow bb$$

A typical derivation look like

$$S \xrightarrow{*} a^{n+1}bc(Bc)^n \xrightarrow{*} a^{n+1}bB^nc^{n+1} \xrightarrow{*} a^{n+1}b^{n+1}c^{n+1}$$

Right?

The “typical” derivation easily produces a proof that $L \subseteq \mathcal{L}(G)$.

But we also need to show that $\mathcal{L}(G) \subseteq L$.

This is a bit harder: we need to show that the productions cannot be abused in some unintended way to generate other strings.

E.g., the canonical order is not necessary:

$$S \xrightarrow{*} aaSBcBc \xrightarrow{*} aaSBBcc$$

Exercise

Figure out the details.

A slightly more complicated example:

$$L = \{ x \in \{a, b, c\}^* \mid \#_a x = \#_b x = \#_c x \}$$

And a suitable grammar:

let $V = \{S, A, B, C\}$ and set

$$S \rightarrow S' \mid \varepsilon$$

$$S' \rightarrow S'ABC \mid ABC$$

$$XY \rightarrow YX \quad \text{for all } X, Y \in \{A, B, C\}$$

$$A \rightarrow a$$

$$B \rightarrow b$$

$$C \rightarrow c$$

Note that most productions only have one syntactic variable on the left (more on these so-called context-free productions next time). The critical part is the commutation productions for $\{A, B, C\}$.

Note that in the proof of “semidecidable is phrase-structure” it is critical that the grammar can erase all the extra space used during the actual computation of the Turing machine, things are wildly non-increasing here.

By contrast, define a grammar to be **monotonic** if all productions are of the form

$$\pi : \alpha \rightarrow \beta \quad \text{where } \alpha \in \Gamma^* V \Gamma^*, \beta \in \Gamma^*, |\alpha| \leq |\beta|$$

As we have seen already, a monotonic grammar can only generate a decidable language.

In fact, these look rather similar to context-sensitive grammars except that we are now allowed to manipulate the context (but see 2 slides down). In particular, every CSG is monotonic.

Theorem

A language is context-sensitive iff it is generated by a monotonic grammar.

Proof.

We cheat and borrow the machine model from the next section: we show that every monotonic language can be accepted by an LBA. Initially, some terminal string $a_1, a_2, \dots, a_n$ is written on the tape.

- Search handle: the head moves to the right a random number of places, say, to a_i .
- Check righthand side: the machine verifies that $a_i, \dots, a_j = \beta$ where $\alpha \rightarrow \beta$ is a production.
- Replace by lefthand side: the block $a_i, \dots, a_j$ is replaced by α , possibly leaving some blanks.
- Collect: remove possible blanks by shifting the rest of the tape left.

This loop repeats until the tape is reduced to S and we accept. If any of the guesses goes wrong we reject.

One can also directly convert monotonic productions to context-sensitive ones. As a simple example, consider a commutativity rule

$$AB \rightarrow BA$$

Here are equivalent context-sensitive rules:

$$AB \rightarrow XB$$

$$XB \rightarrow XY$$

$$XY \rightarrow BY$$

$$BY \rightarrow BA$$

Here X and Y are new variables and the green variable is the one that is being replaced. Note how the left/right context is duly preserved.

Theorem

Context-sensitive languages are closed under union, concatenation, Kleene star and reversal.

They are also closed under ε -free homomorphisms.

Proof is straightforward by manipulating the grammar.

Note that arbitrary homomorphisms do not work in this case: they erase too much information.

- Are CSL closed under intersection?
- Are CSL closed under complement?

The answer is Yes in both cases.

The proof for intersection has to wait until we have a machine model, and the proof for complement requires a special and very surprising counting technique (later).

Theorem (Kuroda)

Every context-sensitive grammar can be written with productions of the form

$$A \rightarrow BC \quad AB \rightarrow CD \quad A \rightarrow a$$

Sketch of proof. Introduce new variables and productions $A \rightarrow a$, and use them to remove terminals from old productions. Then introduce more new variables and productions, and use them to break up long blocks of syntactic variables. □

Note that the recognition algorithm becomes particularly simple when the CSG is given in Kuroda normal form: we first get rid of all terminals and then operate only on pairs of consecutive variables.

CSGs are fairly powerful, in particular it is straightforward to encode configurations and computations of a Turing machine, along the lines of

$$\# C_0 \# C_1 \# \dots \# C_n \#$$

As a consequence even the most basic question about CSG is undecidable.

Theorem

It is undecidable whether a CSG generates the empty language.

- Context-Sensitive Grammars

2 Linear Bounded Automata

- LBA and Counting

So far we have managed to associate language classes with machines:

- semidecidable — Turing machines
- regular — finite state machines

The question arises whether a context-sensitive language can be similarly accepted by some suitable type of machine.

It is clear that we need more memory than for pushdown automata (stacks), but we cannot allow arbitrary memory (as in Turing machines).

Definition

A **linear bounded automaton (LBA)** is a type of one-tape, nondeterministic Turing machine acceptor where the input is written between special end-markers and the computation can never leave the space between these markers (nor overwrite them).

Thus the initial configuration looks like

$$\#q_0x_1x_2\ldots x_n\#$$

and the tape head can never leave the block of n consecutive cells.

It may seem that there is not enough space to perform any interesting computations on an LBA, but note that we can manipulate the tape alphabet to make room.

use a sufficiently large tape alphabet to “compress” the input to a fraction of its original size and make room.

Suppose Σ is the input alphabet, $x \in \Sigma^*$. Then we can use a tape alphabet of the form $\Sigma \times \Gamma$ and use the bottom track for computation.

Alternatively, we could use an alphabet that includes Σ^k and use it to shrink the input to size $|x|/k$: the remaining $n(k-1)/k$ cells can be used for computation.

The last trick can also be used to speed up the computation of any Turing machine by a linear amount.

As an aside: this type of compression argument flies in the face of physical reality. Even product alphabets $\Sigma \times \Sigma \times \dots \times \Sigma$ can become quite problematic.

For example, it was recently demonstrated how to store bits by placing chlorine atoms on a copper substrate (using a scanning electron microscope). This produces fantastic densities, the whole Library of Congress would fit into a few cubic millimeters.

But one cannot store an ASCII symbols instead of a single bit.

Of course, there are lots of other problems: time, space, energy requirements.

The concept of LBA was developed in stages:

- Myhill 1960 considered deterministic LBAs.
- Landweber 1963 showed that they produce only context-sensitive languages.
- Kuroda 1964 generalized to nondeterministic LBAs and showed that this produces precisely the context-sensitive languages.

Theorem

A language is accepted by a (nondeterministic) LBA iff it is context-sensitive.

First suppose L is context-sensitive via G . The idea is to run a derivation of G backwards, starting at a string of terminals.

To this end we can use a nondeterministic Turing machine that “randomly” tries to apply the productions $\alpha A \beta \rightarrow \alpha \gamma \beta$ in G backwards.

Since $\gamma \neq \varepsilon$, we do not need a full Turing machine: a LBA can handle this type of computation. Essentially, we replace γ by A (given the right context).

We will ultimately reach S iff the original word was in L .

For the opposite direction it is a labor of love to check in great gory detail that the workings of an LBA can be described by a context-sensitive grammar.

The critical point is that an LBA can overwrite a tape symbol but can not append new cells at the end (the head never moves beyond the end markers).



Note that nondeterminism is critical here: grammars are naturally nondeterministic and a deterministic machine would have to search through all possible choices. That seems to require more than just linear space (open problem, see below).

It was recognized already in the 1950s that Turing machines are, in many ways, too general to describe anything resembling the type of computation that was possible on emerging digital computers.

The Rabin-Scott paper was one radical attempt to impose a radical constraint on Turing machines that brings them into the realm of “feasible” computation.

Myhill's introduction of LBA is another attempt at constructive restrictions. As we now know, LBAs are still a rather generous interpretation of the notion of feasible computation; real, practical algorithms need further constraints.

Still, it is a perfectly good model and there are many interesting problems that fit perfectly into this framework.

Theorem

CSL are closed under intersection.

Proof.

Given two LBAs M_i for L_i . we can construct a new LBA for $L_1 \cap L_2$ by using a 2-track tape alphabet (recall our synchronous transducers).

The upper track is used to simulate M_1 , the lower track is used to simulate M_2 .

It is easy to check that the simulating machine is again a LBA (it will sweep back and forth over the whole tape, updating both tracks by one step on the way).

□

Definition

A **quantified Boolean formula (QBF)** is a formula consisting of propositional connectives “and,” “or” and “not,” as well as existential and universal quantifiers.

If all variables in a QBF are bounded by a quantifier then the formula has a truth value: we can simply expand it out as in

$$\exists x \varphi(x) \mapsto \varphi(0) \vee \varphi(1)$$

$$\forall x \varphi(x) \mapsto \varphi(0) \wedge \varphi(1)$$

In the end we are left with an exponential size propositional formula without variables which can simply be evaluated.

Note that many properties of propositional formulae can easily be expressed this way. For example, $\varphi(x_1, x_2, \dots, x_n)$ is satisfiable iff

$$\exists x_1, \dots, x_n \varphi(x_1, x_2, \dots, x_n) \text{ is valid}$$

Likewise, the formula is a tautology iff

$$\forall x_1, \dots, x_n \varphi(x_1, x_2, \dots, x_n) \text{ is valid}$$

Claim

Validity of QBF can be checked by a deterministic LBA.

Consider, say,

$$\forall x_1, \dots, x_n \exists y_1, \dots, y_m \varphi(\mathbf{x}, \mathbf{y})$$

Write this into the top track. In the bottom track, implement two loops, one counting from 0 to $2^n - 1$ for $\mathbf{x}$ and another counting from 0 to $2^m - 1$ for $\mathbf{y}$.

The machine checks that for each value of $\mathbf{x}$ there exists a $\mathbf{y}$ that satisfies the quantifier-free part.

Of course, the running time is exponential—but not much space is required.

`http://qbf.satisfiability.org/gallery/`

There is a lot of current research on building powerful QBF solvers.

- Context-Sensitive Grammars

- Linear Bounded Automata

- ③ LBA and Counting

Kuroda stated two open problems in his 1964 paper:

- Is nondeterminism in LBA really needed?
- Are context-sensitive languages closed under complements?

The first problem is still open and seems very difficult.

Note: a positive answer to the first question is sufficient for a positive answer for the second. But, it is not necessary.

Consider only decision problems, using nondeterministic machines for function problems is much more tricky (how do you interpret different answers?)

For Turing machines, nondeterminism can be eliminated: any nondeterministic TM $\mathcal{M}$ can be simulated by a deterministic machine $\mathcal{M}'$ that constructs the computation graph of $\mathcal{M}$.

Similarly, for finite state machines, we also know how to eliminate nondeterminism (though possibly with an exponential blow-up in the number of states).

But for LBA we currently don't know.

Kuroda's second problem was solved independently by two researchers in the late 1980s.

Theorem (Immerman-Szelepcsényi, 1988)

Context-sensitive languages are closed under complement.

The easiest way to prove this is to consider a seemingly unrelated and harmless graph problem first.

Problem:	Graph Reachability
Instance:	A digraph G , two nodes s and t .
Question:	Is there a path from s to t ?

Of course, this can easily be solved in linear time using standard graph algorithms.

The strategy is clear, but there is a problem. We build the set $R \subseteq V$ of vertices reachable from s in G in stages.

Let's say $(u, v) \in E$ **requires attention** if $u \in R$ but $v \notin R$.

```
// reachability
R = {s}
while some edge  $(u, v)$  requires attention
    add  $v$  to  $R$ 
return  $t \in R$ 
```

DFS and BFS are both instances of this strategy. Alas, these algorithms require linear space: we have to keep track of R .

Any algorithm using logarithmic space cannot in general keep track of the set of all reachable vertices, so this seems tricky.

It works, though, if we allow nondeterminism. Let $n = |V|$.

```
// path guessing
 $\ell = 0$ 
 $x = s$ 
while  $\ell < n - 1$  do
    if  $x = t$  then
        return Yes
    guess an edge  $(x, y)$ 
     $\ell++$ 
     $x = y$ 
return No
```

This works in the usual nondeterministic sense:

- If there is a path s to t , then, making the right guesses, the algorithm can return Yes.
- If there is no path, then the algorithm always returns No.

So there may be false negatives, but there can never be false positives.

So, the symmetry between true and false is broken. But we already know that this turns out to be a good thing: nondeterministic finite state machines work that way. They can be made deterministic, but at a potentially exponential cost.

And general semidecision algorithms cannot be fixed at any cost.

How about the opposite problem: t is not reachable from s ?

Problem:	Graph Non-Reachability
Instance:	A digraph G , two nodes s and t .
Question:	Is there no path from s to t ?

Note that for this version nondeterminism seems totally useless: what exactly would we guess? A non-path? A rainbow?? A unicorn???

It is a major surprise that Non-Reachability can also be handled in nondeterministic logarithmic space, though the logical complexity of the algorithm is substantially higher.

Let

$$R = \{ x \in V \mid \text{exists path } s \rightarrow x \text{ in } G \}$$

be the set of vertices reachable from s (the weakly connected component of s).

From the perspective of nondeterministic logarithmic space, R can be exponentially large (it may have cardinality up to n). But, it suffices to know just its cardinality to determine non-reachability.

Claim

Given the cardinality $r = |R|$, we can solve Non-Reachability in nondeterministic logarithmic space.

```
// non-reachability algorithm
 $\ell = 0$ 
forall  $x \neq t$  in  $V$  do           // length-lex order
    guess path  $s \rightarrow x$ 
    if path found
    then  $\ell++$ 
return  $\ell == r$ 
```

This works since $i = r$ means that all reachable vertices are different from t .

Nice, but useless unless we can actually compute r .

Instead of trying to compute r directly, let

$$R_\ell = \{ x \in V \mid \text{exists path } s \rightarrow x \text{ of length } \leq \ell \}$$

and set $r_\ell = |R_\ell|$.

Obviously $r_0 = 1$ and $r_{n-1} = r$.

So we only have to figure out how to compute r_ℓ , given $r_{\ell-1}$.

```
// inductive counting algorithm
// inputs  $\ell, \rho$ ; output  $\rho'$ 

 $\rho' = 0$ 
forall  $x$  in  $V$  do
     $b = 0$                                 // increment 0 or 1
     $c = 0$                                 // path counter
    forall  $y$  in  $V$  do
        guess path  $s \rightarrow y$  of length  $< \ell$ 
        if path found
            then  $c++$ 
            if  $y = x$  or  $(y, x) \in E$ 
                then  $b = 1$ 
    assert  $c == \rho$                         // all paths found
     $\rho' += b$ 

return  $\rho'$ 
```

Suppose ρ is the correct value of $r_{\ell-1}$.

Since we check that $c = \rho$, we know that all paths $s \rightarrow y$ of length less than ℓ have been guessed correctly.

But then we have properly found and counted all paths of length at most ℓ : we just have to add one more edge. Hence, the output ρ' is indeed r_ℓ .

Note that the algorithm requires only storage for a constant number of vertices, so logarithmic space suffices, plus nondeterminism to guess the right path from s to y .

The counting algorithm is heavily nondeterministic: we have to guess multiple paths $s \rightarrow y$ (which would never work out if we were to flip a coin to make nondeterministic decisions).

But note the **assert** statement: if we make a mistake, the whole computation crashes. This means in particular that no function value is produced on any of these branches.

The branches where all the guesses are correct all produce the exactly same value ρ .

The theorem now follows easily using a computation graph argument.

For suppose $\mathcal{M}$ is some LBA (nondeterministic!) which accepts some CSL L . We would like to build another LBA $\mathcal{N}$ that accepts $\Sigma^* - L$.

Consider some input $x \in \Sigma^\ell$. Clearly the number of configurations of $\mathcal{M}$ on x is bounded by $n = c^\ell$ for some constant $c > 1$. Note that n also bounds the running time of $\mathcal{M}$ (remove loops).

Now consider the computation graph of $\mathcal{M}$ on x : vertices are all such configurations, the source node s is the initial configuration for input x , and terminal node t is the accepting configuration. Edges are given by the one-step relation for $\mathcal{M}$.

The construction for Non-Reachability from above can then be handled by another LBA $\mathcal{N}$, done.

In a while, we will introduce **space complexity classes**: things that can be done using some amount $s(n)$ of space. Then we have:

Theorem

For any reasonable function $s(n) \geq \log n$ we have

$\text{NSPACE}(s(n)) = \text{co-NSPACE}(s(n))$.

In particular $\text{NL} = \text{co-NL}$ and $\text{NSPACE}(n) = \text{co-NSPACE}(n)$.

For this version one considers graphs determined by the computations of a nondeterministic Turing machine (vertices are instantaneous descriptions, edges correspond to the one-step relation).

Yet another Boolean algebra: the context-sensitive languages form a Boolean algebra, just like the regular and decidable ones.

But the following is somewhat surprising:

Theorem

The Boolean algebras of regular, context-sensitive and decidable languages are all isomorphic.

The proof requires a bit of lattice theory; we'll skip.

For an NFA $\mathcal{A} = \langle Q, \Sigma, \delta; I, F \rangle$, recall that $\text{pow}(\mathcal{A})$ denotes the accessible part of the power automaton produced by the Rabin-Scott determinization procedure.

We have $1 \leq |\text{pow}(\mathcal{A})| \leq 2^n$ where $n = |Q|$, and we have examples that show that the upper bound is hard, even when $Q = I = F$.

Now consider the following problem:

Problem:	Power-Automaton Reachability
Instance:	An NFA $\mathcal{A}$ and a state set $P \subseteq Q$.
Question:	Is P in $\text{pow}(\mathcal{A})$?

Power-Automaton Reachability can be “solved” by a nondeterministic LBA:
The LBA can guess a string $x \in \Sigma^*$ and verify that $\delta(I, x) = P$.

Note that we have to guess the witness $x = x_1x_2 \dots x_m$ one symbol at a time:
 m may be exponentially larger than n .

Surprisingly, by Immerman-Szelepsényi, the corresponding non-reachability problem can also be solved by an LBA: we can “guess” in linear space that $P \subseteq Q$ is not part of $\text{pow}(\mathcal{A})$.

This is another case where your intuition might suggest otherwise: what's there to guess?

Similarly the following problem can be solved by a nondeterministic LBA:

Problem: **Power-Automaton Size**
Instance: An NFA $\mathcal{A}$ and a bound B .
Question: Does $\text{pow}(\mathcal{A})$ have size at least B ?

The LBA can

- guess B -many subsets $P \subseteq Q$, say, in lexicographic order, and
- guess a string $x \in \Sigma^*$ and verify that $\delta(I, x) = P$.

Again by Immerman-Szelepsényi the corresponding problems $|\text{pow}(\mathcal{A})| \leq B$, $|\text{pow}(\mathcal{A})| = B$, $|\text{pow}(\mathcal{A})| \neq B$ can all be solved by an LBA.

All these problems are PSPACE-complete.