UCT

More on Space

Klaus Sutner

Carnegie Mellon University

Spring 2022

1 **Small Space**

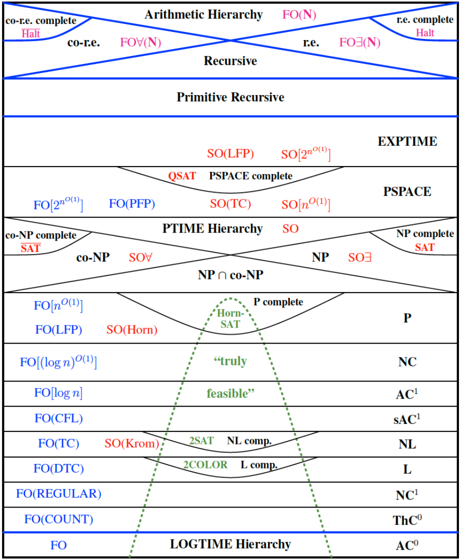
2 **Crash Course: Grammars**

3 **Properties of CFLs**

Where Are We?

- $\mathrm{SPACE}(1)$, constant space,
- $\mathbb{L} = \mathrm{SPACE}(\mathsf{log})$, logarithmic space,
- $\mathbb{NL} = \mathrm{NSPACE}(\mathsf{log})$, nondeterministic logarithmic space,
- $\mathbb{P} = \mathrm{TIME}(\mathsf{poly})$, polynomial time,
- $\mathbb{NP} = \mathrm{NTIME}(\mathsf{poly})$, nondeterministic polynomial time,
- $\mathrm{SPACE}(n)$, linear space,
- $\mathrm{NSPACE}(n)$, nondeterministic linear space,
- $\mathrm{PSPACE} = \mathrm{SPACE}(\mathsf{poly})$, polynomial space.
- All the old stuff: regular, CFL, CSL, primitive recursive, decidable, semidecidable.

Immerman's Zoo



Arithmetic Hierarchy FO(**N**)

co-r.e. complete $\overline{\text{Halt}}$

r.e. complete Halt

co-r.e. FO∀(**N**)

r.e. FO∃(**N**)

Recursive

Primitive Recursive

EXPTIME

SO(LFP) $\mathrm{SO}[2^{n^{O(1)}}]$

QSAT PSPACE complete

PSPACE

$\mathrm{FO}[2^{n^{O(1)}}]$ FO(PFP) SO(TC) $\mathrm{SO}[n^{O(1)}]$

PTIME Hierarchy SO

co-NP complete $\overline{\text{SAT}}$

NP complete SAT

co-NP SO∀

NP SO∃

NP ∩ co-NP

$\mathrm{FO}[n^{O(1)}]$

P complete

Horn-SAT

P

FO(LFP) SO(Horn)

$\mathrm{FO}[(\log n)^{O(1)}]$ "truly NC

$\mathrm{FO}[\log n]$ feasible" AC^1

FO(CFL) sAC^1

FO(TC) SO(Krom) 2SAT NL comp. NL

FO(DTC) 2COLOR L comp. L

FO(REGULAR) NC^1

FO(COUNT) ThC^0

FO LOGTIME Hierarchy AC^0

Recall: Computation Graphs

Lemma

$\mathbb{L} \subseteq \mathbb{NL} \subseteq \mathbb{P}$, *but no separation known.*

The proof uses the computation graph $\mathfrak{C}(\mathcal{M}, x)$: the graph has polynomial size, so we can use any vanilla graph reachability algorithm to check acceptance.

Same Old

So this is really our old friend

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

Clearly, Reachability is in deterministic linear time **and** space: any standard graph exploration algorithm such as DFS and BFS works just fine.

Question: Can we squeeze the space requirement?

Prototype Reachability Algorithm

In all standard algorithms, 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 $v \notin R$.

```
$R = \{s\}$;
while some edge $(u, v) \in E$ requires attention do
    add $v$ to $R$;
return $R$;
```

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

How About Logarithmic Space?

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 $(x, y) \in E$
    $\ell$++
    $x = y$
return No
```

Works!

This “algorithm” works in the usual nondeterministic way:

- 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, the symmetry between true and false is broken; there may be false negatives, but there can never by false positives.

We already know that this is a good idea from $\mathbb{NP}$, and even in the world of practical algorithms: nondeterministic finite state machines work that way.

2-Unsatisfiability

2-UNSAT is the problem of determining whether a 2-CNF formula fails to be satisfiable.

This problem is in $\mathbb{P}$: consider the graph on all literals with edges $\overline{x} \to y$ and $y \to \overline{x}$ for any clause $\{x, y\}$. The formula fails to be satisfiable iff there is a path $x \rightsquigarrow \overline{x} \rightsquigarrow x$ for some variable x. Of course, for $\mathbb{P}$ we have true/false symmetry, so this is underwhelming.

More interestingly, 2-UNSAT is in $\mathbb{NL}$.

Non-Reachability

Since symmetry is broken, how about the opposite problem: target t is not reachable from source 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 proof?? 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. More later.

Guess & Verify

For those who prefer the polynomial projection definition of $\mathbb{NP}$, one can define $\mathbb{NL}$ in a similar manner. To wit, $A \in \mathbb{NL}$ iff there is a deterministic log-space Turing machine $\mathcal{M}$ and a polynomial p such that

$$x \in A \iff \exists\, w \in \Sigma^{p(|x|)} \big(\mathcal{M} \text{ accepts } (w, x)\big)$$

Moreover, the witness is given to $\mathcal{M}$ on a separate, read-once input tape.

Note the slightly strange read-once condition.

Exercise

Figure out why the read-once condition is necessary.

The Right Reduction

As usual, to deal with classes $\mathbb{L}$ and $\mathbb{NL}$ we need a suitable notion of reduction: sufficiently fine-grained and compatible.

We have already encountered the right choice: logarithmic space works, it entails polynomial time, but there is no reason why a polynomial time computation should require only logarithmic space in general (just think about depth-first-search).

As always, the underlying machine model separates input/output tapes from the work tape.

Log-Space Reductions

Definition

Language A is log-space reducible to B if there is a function $f : \Sigma^\star \to \Sigma^\star$ that is computable by a log-space Turing machine such that

$$x \in A \iff f(x) \in B$$

We will write $A \leq_l B$.

As already pointed out, many of the polynomial time reductions showing $\mathbb{NP}$-hardness are in fact log-space reductions. For example, SAT is $\mathbb{NP}$-complete with respect to log-space reductions.

Key Properties

Lemma

- *$A \leq_l B$ implies $A \leq_m^p B$.*
- *$\leq_l$ is a pre-order (reflexive and transitive).*

Proof.

The first part is clear: the log-space reduction can be computed in time $O(2^{c \log n}) = O(n^c)$.

Reflexivity is trivial, as always.

Transitivity

This is more problematic: one cannot simply combine two log-space transducers by identifying the output tape of the first machine with the input tape of the second machine: this space would count towards the work space and may be too large.

Therefore we do not compute the whole output, rather we keep a pointer to the current position of the input head of the second machine.

Whenever the head moves, we compute the corresponding symbol from scratch using the first machine. The pointer needs only $\log(n^c) = O(\log n)$ bits. For this it is critical that we have a two-way input tape.

□

Closure under Log-Space Reductions

The following lemma shows that space complexity classes are closed under log-space reductions (under some mild technical conditions).

Lemma

Let $A \leq_l B$ *via a log-space reduction* $f : \Sigma^\star \to \Sigma^\star$ *and* $|f(x)| \leq \ell(|x|)$ *a polynomial bound on the output length.*

Then $B \in \mathrm{SPACE}(s(n))$ *implies* $A \in \mathrm{SPACE}(s(\ell(n)) + \log n)$.

The same holds for non-deterministic space.

Again, one cannot use the same argument as for $\leq_m^p$ and time classes since ℓ will not be logarithmic in general.

Proof

Therefore we use the same trick as in the transitivity lemma.

Let $\mathcal{M}$ be an acceptor for B in $s(n)$ space, with separate input tape. Here is an algorithm to test whether $x \in A$:

Instead of writing $y = f(x)$ on the input tape, we compute y symbol by symbol and input the symbols directly to $\mathcal{M}$, according to the movements of its input head. We keep track of the position of the current symbol.

Now $|y| \leq \ell(|x|)$, hence $\mathcal{M}$ will use no more than $s(\ell(|x|))$ space on its worktape.

Together with the position pointer, the total space requirement is $O((\ell(n)) + \log n)$.

□

Compatibility

Lemma

- *If B is in $\mathbb{L}$ and $A \leq_l B$ then A is also in $\mathbb{L}$.*
- *If B is in $\mathbb{NL}$ and $A \leq_l B$ then A is also in $\mathbb{NL}$.*

Proof.

Again, we cannot simply compute $y = f(x)$ in log-space and then use the log-space algorithm for B.

Instead we recompute every symbol y_i on demand, just like above. We are essentially trading time for space.

The nondeterministic case is entirely similar.

□

Hardness and Completeness

Definition

B is $\mathbb{NL}$-hard if for all A in $\mathbb{NL}$: $A \leq_l B$.

B is $\mathbb{NL}$-complete if B is in $\mathbb{NL}$ and is also $\mathbb{NL}$-hard.

As before with $\mathbb{NP}$ and PSPACE, the key is to produce natural examples of $\mathbb{NL}$-complete problems.

Ideally these problems should have been studied before anyone even thought about $\mathbb{NL}$.

Reachability

Theorem

Graph Reachability $\mathbb{NL}$-complete with respect to log-space reductions.

Proof.

Suppose we have a nondeterministic log-space machine $\mathcal{M}$ and some input x.

The computation graph $\mathfrak{C}(\mathcal{M}, x)$ has nodes of size $2^{c \log n}$ (and thus polynomial size).

The construction of $\mathfrak{C}(\mathcal{M}, x)$ can be carried out in logarithmic space: we can write down the adjacency matrix using just a handful of counters.

But then acceptance of $\mathcal{M}$ translates into a graph reachability problem in $\mathfrak{C}(\mathcal{M}, x)$, as usual.

□

1 Small Space

2 **Crash Course: Grammars**

3 Properties of CFLs

RealWorld

Inspecting actual algorithms, it seems that

- logarithmic space, and
- linear space

are far and away the most important space complexity classes.

Perhaps surprisingly, nondeterministic linear space is also quite natural. To see why, let's take a quick detour and talk a little bit about the old-fashioned complexity theory from the 1950s.

Generation vs. Recognition

Turing machines can be used to check membership in decidable sets. They can also be used to enumerate semidecidable sets, whence the classical notion of recursively enumerable sets.

For languages $L \subseteq \Sigma^\star$ there is a similar notion of generation.

The idea is to set up a system of simple rules that can be used to derive all words in a particular formal language. These systems are typically highly nondeterministic and it is not clear how to find (efficient) recognition algorithms for the corresponding languages.

Noam Chomsky

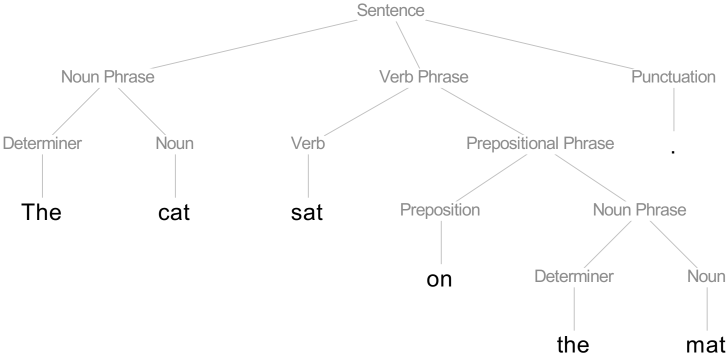
Historically, these ideas go back to work by Chomsky in the 1950s. Chomsky was mostly interested natural languages: the goal is to develop grammars that differentiate between grammatical and and ungrammatical sentences.

1. The cat sat on the mat.
2. The mat on the sat cat.

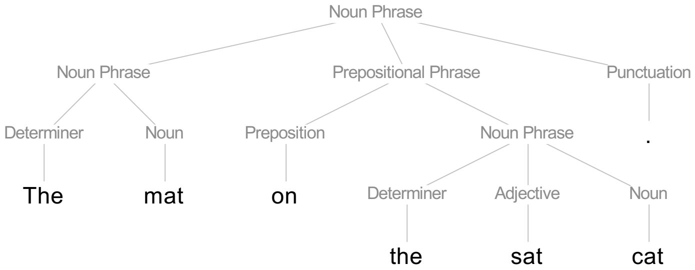
Alas, this turns out to be inordinately difficult, syntax and semantics of natural languages are closely connected and very complicated.

But for artificial languages such as programming languages, Chomsky's approach turned out be perfectly suited.

Cat-Mat Example



Mat-Cat Example



Killer App: Programming Languages

Many programming languages have a block structure like so:

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

What is a good description of such structures, and how do we build appropriate algorithms?

Grammars

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.

- 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. Then $\mathcal{P}$ is a finite subset of $\Gamma^\star \times \Gamma^\star$.

Preserving Sanity

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

There are many types floating around here, it's a good idea to build a suitable notation system.

Derivations

Given a grammar G, we define a one-step relation $\stackrel{1}{\Longrightarrow}$ on $\varGamma^\star$

$$\alpha\beta\gamma \stackrel{1}{\Longrightarrow} \alpha\delta\gamma \quad \text{if} \quad \beta \to \delta \in \mathcal{P}$$

Think of β as a handle and δ as the replacement string. As usual, by induction define

$$\alpha \stackrel{k+1}{\Longrightarrow} \beta \quad \text{if} \quad \exists\, \gamma\, (\alpha \stackrel{k}{\Longrightarrow} \gamma \wedge \gamma \stackrel{1}{\Longrightarrow} \beta)$$

and

$$\alpha \stackrel{*}{\Longrightarrow} \beta \quad \text{if} \quad \exists\, k \;\; \alpha \stackrel{k}{\Longrightarrow} \beta$$

in which case one says that α derives or yields β. α is a sentential form if it can be derived from the start symbol S.

To keep notation simple we'll often just write $\alpha \Longrightarrow \beta$.

Generated Languages

Definition

The language of a grammar G is defined to be

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

We also say that G generates $\mathcal{L}(G)$.

Thus $\mathcal{L}(G)$ is the set of all sentential forms in $\Sigma^\star$.

In modern parlance one often speaks of rewrite systems. All of this really comes down to plain word processing: find a handle, and replace it. Rinse and repeat.

Characterization

The following theorem is not hard to show.

Theorem

The languages generated by arbitrary formal grammars are precisely the semidecidable ones.

So, we have yet another characterization of semidecidable sets, based on a notion of effective enumeration.

That's nice, but a catastrophe from the perspective of using grammars as a way to specify programming languages: clearly, they should all be decidable. We need to impose restrictions on the type of productions allowed.

Context Free Grammars

Definition (CFG)

A context free grammar is a grammar where the productions have the form

$$\mathcal{P} \subseteq V \times \Gamma^\star$$

Thus we can only replace syntactic variables and the productions take the particularly simple form

$$\pi : A \to \alpha$$

where $A \in V$ and $\alpha \in \Gamma^\star$.

Note that this makes is very easy to find a handle.

Context Free Languages

Definition

A language is context free (CFL) if there exists a context free grammar that generates it.

Note that in a CFG one can replace a single syntactic variable A by strings over Γ independently of were A occurs; whence the name "context free."

Later on we will generalize to replacement rules that operate on a whole block of symbols (context sensitive grammars).

Machines

General grammars produce arbitrary semidecidable (i.e. recursively enumerable) sets and thus correspond closely to Turing machines.

For a context-free grammar we don't need the full power of Turing machines. Instead, push-down automata (essentially finite state machines with an additional stack) correspond precisely to context-free languages.

Example: Regular

Let $G = \langle \{S, A, B\}, \{a, b\}, \mathcal{P}, S \rangle$ where the set $\mathcal{P}$ of productions is defined by:

$$
\begin{aligned}
S &\to aA \mid aB \\
A &\to aA \mid aB \\
B &\to bB \mid b.
\end{aligned}
$$

A typical derivation is:

$$S \Rightarrow aA \Rightarrow aaA \Rightarrow aaaB \Rightarrow aaabB \Rightarrow aaabb$$

It is not hard to see that

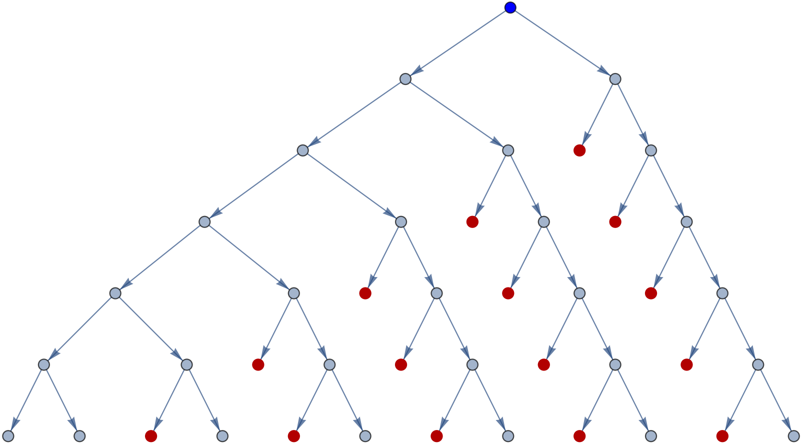
$$\mathcal{L}(G) = a^+ b^+$$

Not too interesting, we already know how to deal with regular languages.

Exercise

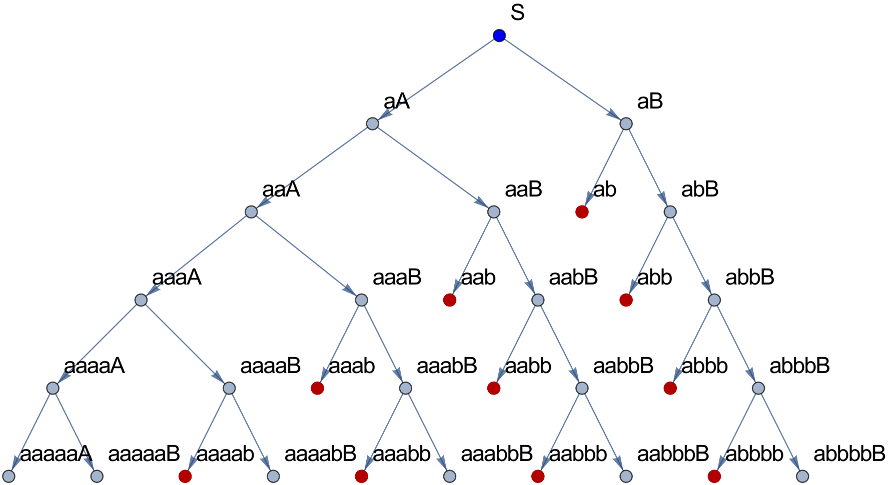
There is a finite state machine hiding in the grammar; which one?

Derivation Graph



Derivations of length at most 6 in this grammar.

Labeled


S
aA
aB
aaA
aaB
ab
abB
aaaA
aaaB
aab
aabB
abb
abbB
aaaaA
aaaaB
aaab
aaabB
aabb
aabbB
abbb
abbbB
aaaaaA
aaaaaB
aaaab
aaaabB
aaabb
aaabbB
aabbb
aabbbB
abbbb
abbbbB

Example: Mystery

Let $G = \langle \{A, B\}, \{a, b\}, \mathcal{P}, A \rangle$ where the set $\mathcal{P}$ of productions is defined by:

$$\begin{aligned} A &\to AA \mid AB \mid a \\ B &\to AA \mid BB \mid b. \end{aligned}$$

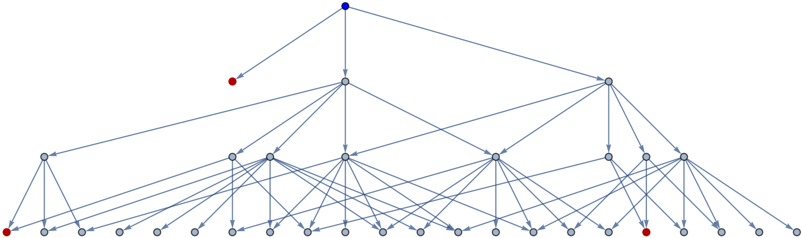
A typical derivation is:

$$A \Rightarrow AA \Rightarrow AAB \Rightarrow AABB \Rightarrow AABAA \Rightarrow aabaa$$

Exercise

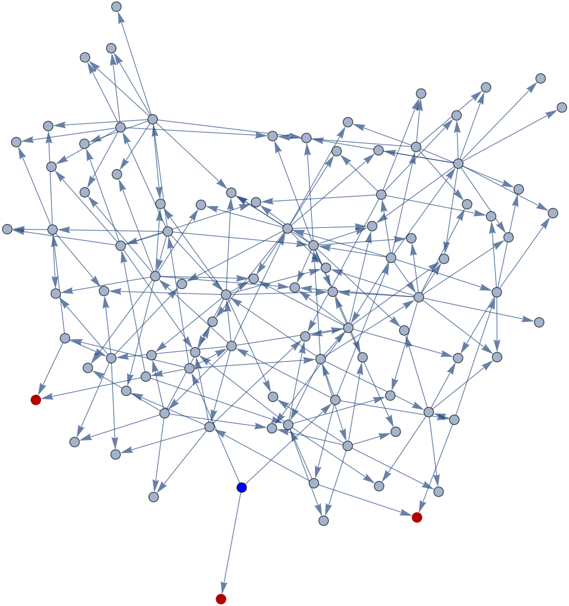
Find a simple description of the language generated by this grammar.

Derivation Graph



Derivations of length at most 3 in this grammar. Three terminal strings appear at this point.

Depth 4



Example: Counting

Let $G = \langle \{S\}, \{a, b\}, \mathcal{P}, S \rangle$ where the set $\mathcal{P}$ of productions is defined by:

$$S \rightarrow aSb \mid \varepsilon$$

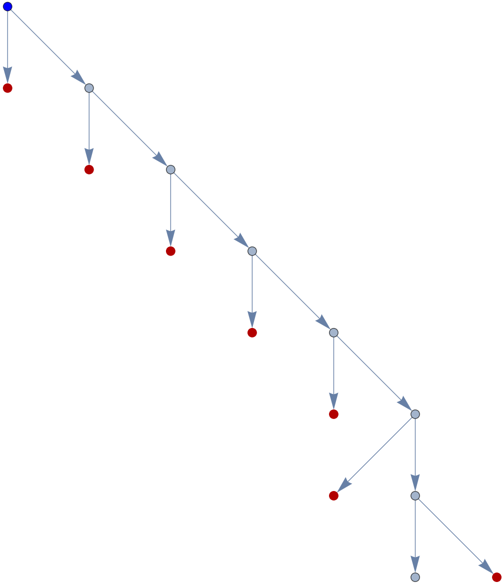
A typical derivation is:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

Clearly, this grammar generates the language $\{\, a^i b^i \mid i \geq 0 \,\}$

It is easy to see that this language is not regular.

Derivation Graph



Example: Palindromes

Let $G = \langle \{S\}, \{a, b\}, \mathcal{P}, S \rangle$ where the set $\mathcal{P}$ of productions is defined by:

$$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$$

A typical derivation is:

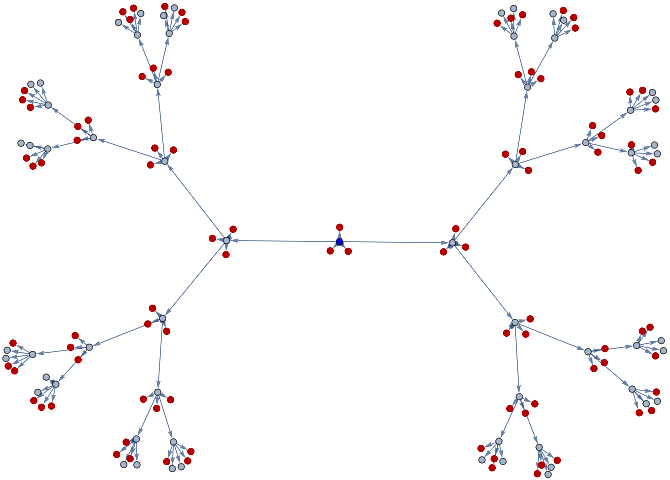
$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aababaa$$

This grammar generates the language of palindromes.

Exercise

Give a careful proof of this claim.

Derivation Graph



Example: Parens

Let $G = \langle \{S\}, \{(,)\}, \mathcal{P}, S \rangle$ where the set $\mathcal{P}$ of productions is defined by:

$$S \to SS \mid (S) \mid \varepsilon$$

A typical derivation is:

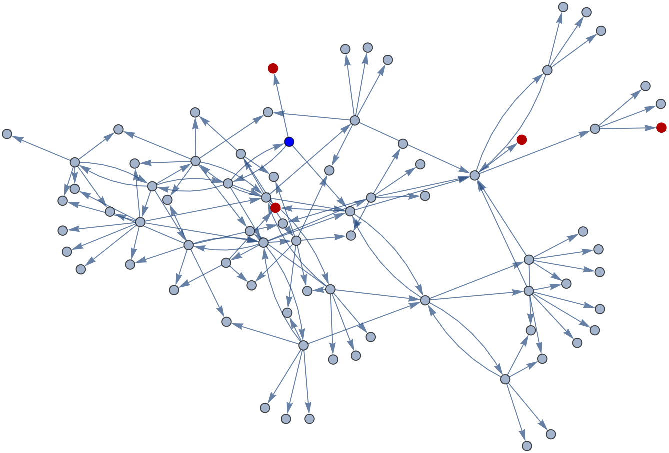
$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow (S)((S)) \Rightarrow ()(())$$

This grammar generates the language of well-formed parenthesized expressions.

Exercise

Give a careful proof of this claim.

Derivation Graph



Example: Expressions of Arithmetic

Let $G = \langle \{E\}, \{+, *, (,), v\}, \mathcal{P}, E \rangle$ where the set $\mathcal{P}$ of productions is defined by:

$$E \to E + E \mid E * E \mid (E) \mid v$$

A typical derivation is:

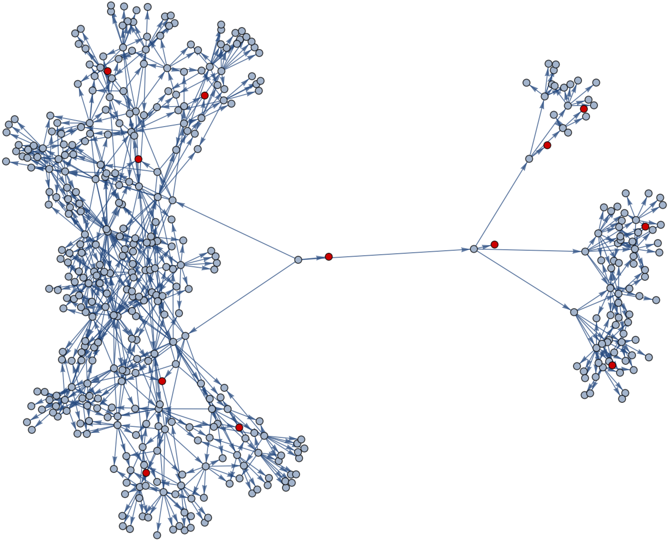
$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow v * (v + v)$$

This grammar generates a language of arithmetical expressions with plus and times. Alas, there are problems: the following derivation is slightly awkward.

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

Our grammar is symmetric in $+$ and $*$, it knows nothing about precedence.

Derivation Graph



1 Small Space

2 Crash Course: Grammars

3 **Properties of CFLs**

Regular Implies Context Free

Lemma

Every regular language is context free.

Proof. Suppose $\mathcal{M} = \langle Q, \Sigma, \delta; q_0, F \rangle$ is a DFA for L. Consider a CFG with $V = Q$ and productions

$$\begin{array}{lcll} p & \to & a\,q & \text{if } \delta(p,a) = q \\ p & \to & \varepsilon & \text{if } p \in F \end{array}$$

Let q_0 be the start symbol.

□

Note how the productions generate the symbol consumed by the transitions.

Substitutions

Definition

A substitution is a map $\sigma : \Sigma \to \mathfrak{P}(\Gamma^\star)$.

The idea is that for any word $x \in \Sigma^\star$ we can define its image under σ to be language

$$\sigma(x_1) \cdot \sigma(x_2) \cdot \ldots \cdot \sigma(x_n)$$

Likewise, $\sigma(L) = \bigcup_{x \in L} \sigma(x)$.

If $\sigma(a) = \{w\}$ then we have essentially a homomorphism.

The Substitution Lemma

Lemma

Let $L \subseteq \Sigma^\star$ be a CFL and suppose $\sigma : \Sigma \to \mathfrak{P}(\Gamma^\star)$ is a substitution such that $\sigma(a)$ is context free for every $a \in \Sigma$. Then the language $\sigma(L)$ is also context free.

Proof.

Let $G = \langle V, \Sigma, \mathcal{P}, S \rangle$ and $G_a = \langle V_a, \Gamma, \mathcal{P}_a, S_a \rangle$ be CFGs for the languages L and $L_a = \sigma(a)$ respectively. We may safely assume that the corresponding sets of syntactic variables are pairwise disjoint.

Define G' as follows. Replace all terminals a on the right hand side of a production in G by the corresponding variable S_a.

It is obvious that $f(\mathcal{L}(G')) = L$ where f is the homomorphism defined by $f(S_a) = a$.

Proof, cont'd

Now define a new grammar H as follows.

The variables of H are $V \cup \bigcup_{a \in \Sigma} V_a$, the terminals are Σ, the start symbol is S and the productions are given by

$$\mathcal{P}' \cup \bigcup_{a \in \Sigma} \mathcal{P}_a$$

Then the language generated by H is $\sigma(L)$.

It is clear that H derives every word in $\sigma(L)$.

For the opposite direction consider the parse trees in H.

□

Closure Properties

Corollary

Suppose $L, L_1, L_2 \subseteq \Sigma^\star$ *are CFLs. Then the following languages are also context free:* $L_1 \cup L_2$, $L_1 \cdot L_2$ *and* L^**: context free languages are closed under union, concatenation and Kleene star.*

Proof.

This follows immediately from the substitution lemma and the fact that the languages $\{a, b\}$, $\{ab\}$ and $\{a\}^*$ are trivially context free.

□

Non Closure

Proposition

CFLs are not closed under intersection and complement.

Consider

$$L_1 = \{\, a^i b^i c^j \mid i, j \geq 0 \,\} \qquad L_2 = \{\, a^i b^j c^j \mid i, j \geq 0 \,\}$$

One can show that $L_1 \cap L_2 = \{\, a^i b^i c^i \mid i \geq 0 \,\}$ fails to be context free.

More Closure

Lemma

Suppose L is a CFL and R is regular. Then $L \cap R$ is also context free.

Proof.

This requires a machine model for CFLs (push-down automata), we'll skip.

□

Dyck Languages

One can generalize strings of balanced parentheses to strings involving multiple types of parens.

To this end one uses special alphabets with paired symbols:

$$\varGamma = \varSigma \cup \{\, \overline{a} \mid a \in \varSigma \,\}$$

The Dyck language D_k is generated by the grammar

$$S \to SS \mid a\,S\,\overline{a} \mid \varepsilon$$

A typical derivation looks like so:

$$S \Rightarrow SS \Rightarrow aS\overline{a}S \Rightarrow aaS\overline{a}\,\overline{a}S \Rightarrow aaS\overline{a}\,\overline{a}aS\overline{a} \Rightarrow aa\overline{a}\,\overline{a}a\overline{a}$$

Exercise

Find an alternative definition of a Dyck language.

Dyck vs. CF

In a strong sense, Dyck languages are the "most general" context free languages: all context free languages are built around the notion of matching parens, though this may not at all be obvious from their definitions (and, actually, not even from their grammars).

Theorem (Chomsky-Schützenberger 1963)

Every context free language $L \subseteq \Sigma^\star$ *has the form* $L = h(D \cap R)$ *where* D *is a Dyck language,* R *is regular and* h *is a homomorphism.*

The proof also relies on push-down automata, so we skip.

The Recognition Problem

Problem:	**Context Free Recognition**
Instance:	A CFG G and a word $x \in \Sigma^\star$.
Question:	Is $x \in \mathcal{L}(G)$?

Of course, in applications to programming languages, just checking membership is nowhere near enough:

- If x is in the language, we want a parse tree.
- If x is not in the language, we want a reason; e.g., a place in x where there might be a typo.

Cocke-Younger-Kasami

Theorem

There is a cubic time, quadratic space algorithm that checks whether a word is generated by a context-free grammar.

The general algorithm requires a special normal form (Chomsky normal form) and is mostly of academic interest, there are much better methods for the more restricted types of CFGs relevant for programming languages.

Discovered and rediscovered by Sakai, Cocke, Younger, Kasami and Schwartz.

CYK Algorithm

```
for $i = 1, \ldots, n$ do
        $t_{i,i} = \{ A \mid A \rightarrow w_i \}$;

for $d = 1, \ldots, n-1$ do
for $i = 1, \ldots, n-d$ do
        $j = d + i$;
        for $k = i, \ldots, j-1$ do
                if exists $A \rightarrow BC$, $B \in t_{i,k}$, $C \in t_{k+1,j}$
                then add $A$ to $t_{i,j}$;
check $S \in t_{1,n}$
```

A classical example of bottom-up parsing and dynamic programming,

Example I

Consider the following grammar G in CNF, A being the start symbol.

$$\begin{aligned} A &\to AA \mid AB \mid a \\ B &\to AA \mid BB \mid b \end{aligned}$$

Again, it is not entirely clear what language this grammar describes.

The CYK algorithm on input $u = abababaaa$ and $v = bababbb$ produces the following two recognition matrices, showing that u is in the language generated by G whereas v is not.

Recognition Matrices

$$\begin{array}{rrrrrrr}
A & A & AB & AB & AB & AB & AB \\
 & B & - & - & B & B & B \\
 & & A & A & AB & AB & AB \\
 & & & B & - & B & B \\
 & & & & A & AB & AB \\
 & & & & & A & AB \\
 & & & & & & A
\end{array}$$

$$\begin{array}{rrrrrrr}
B & - & - & B & B & B & B \\
 & A & A & AB & AB & AB & AB \\
 & & B & - & - & - & - \\
 & & & A & A & A & A \\
 & & & & B & B & B \\
 & & & & & B & B \\
 & & & & & & B
\end{array}$$