Learning Objectives:

- See that Turing completeness is not always a good thing
- Two important examples of non-Turing-complete, always-halting formalisms: *regular expressions* and *context-free grammars*.
- The pumping lemmas for both these formalisms, and examples of non regular and non context-free functions.
- Unrestricted grammars, and another example of an uncomputable function.

11

Restricted computational models

> *"Happy families are all alike; every unhappy family is unhappy in its own way"*, Leo Tolstoy (opening of the book "Anna Karenina").

As we saw before, many natural computational models turn out to be *equivalent* to one another, in the sense that we can transform a "program" of one model (such as a λ expression, or a game-of-life configurations) into another model (such as a NAND++ program). This equivalence implies that we can translate the uncomputability of the Halting problem for NAND++ programs into uncomputability for Halting in other models. For example:

Theorem 11.1 — Turing Machine Halting. Let $TMHALT : \{0,1\}^* \to \{0,1\}$ be the function that on input strings $M \in \{0,1\}^*$ and $x \in \{0,1\}^*$ outputs 1 if the Turing machine described by M halts on the input x and outputs 0 otherwise. Then $TMHALT$ is uncomputable.

P Once again, this is a good point for you to stop and try to prove the result yourself before reading the proof below.

Proof. We have seen in Theorem 9.2 that for every NAND++ program P there is an equivalent Turing machine M_P such that for every x, that computes the same function. The machine M_P exactly simulated P, in the sense that M_P halts on x if and only P halts on x (and moreover if they both halt, they produce the same output). Going back to the proof of Theorem 9.2, we can see that the transformation

of the program P to the Turing machine $M(P)$ was described in a *constructive* way.

Specifically, we gave explicit instructions how to build the Turing machine $M(P)$ given the description of the program P. Thus, we can view the proof of Theorem 9.2 as a high level description of an *algorithm* to obtain M_P from the program P, and using our "have your cake and eat it too" paradigm, this means that there exists also a NAND++ program R such that computes the map $P \mapsto M_P$. We see that

$$HALT(P,x) = TMHALT(M_P,x) = TMHALT(R(P),x) \tag{11.1}$$

and hence if we assume (towards the sake of a contradiction) that $TMHALT$ is computable then Eq. (11.1) implies that $HALT$ is computable, hence contradicting Theorem 10.2. ■

The same proof carries over to other computational models such as the λ *calculus*, *two dimensional* (or even one-dimensional) *automata* etc. Hence for example, there is no algorithm to decide if a λ expression evaluates the identity function, and no algorithm to decide whether an initial configuration of the game of life will result in eventually coloring the cell $(0,0)$ black or not.

The uncomputability of halting and other semantic specification problems motivates coming up with **restricted computational models** that are **(a)** powerful enough to capture a set of functions useful for certain applications but **(b)** weak enough that we can still solve semantic specification problems on them. In this lecture we will discuss several such examples.

11.1 *Turing completeness as a bug*

We have seen that seemingly simple computational models or systems can turn out to be Turing complete. The following webpage lists several examples of formalisms that "accidentally" turned out to Turing complete, including supposedly limited languages such as the C preprocessor, CCS, SQL, sendmail configuration, as well as games such as Minecraft, Super Mario, and the card game "Magic: The gathering". This is not always a good thing, as it means that such formalisms can give rise to arbitrarily complex behavior. For example, the postscript format (a precursor of PDF) is a Turing-complete programming language meant to describe documents for printing. The expressive power of postscript can allow for short description of very complex images. But it also gives rise to some nasty surprises, such

as the attacks described in this page ranging from using infinite loops as a denial of service attack, to accessing the printer's file system.

An interesting recent example of the pitfalls of Turing-completeness arose in the context of the cryptocurrency Ethereum. The distinguishing feature of this currency is the ability to design "smart contracts" using an expressive (and in particular Turing-complete) language.

In our current "human operated" economy, Alice and Bob might sign a contract to agree that if condition X happens then they will jointly invest in Charlie's company. Ethereum allows Alice and Bob to create a joint venture where Alice and Bob pool their funds together into an account that will be governed by some program P that decides under what conditions it disburses funds from it. For example, one could imagine a piece of code that interacts between Alice, Bob, and some program running on Bob's car that allows Alice to rent out Bob's car without any human intervention or overhead.

Specifically Ethereum uses the Turing-complete programming solidity which has a syntax similar to Javascript. The flagship of Ethereum was an experiment known as The "Decentralized Autonomous Organization" or The DAO. The idea was to create a smart contract that would create an autonomously run decentralized venture capital fund, without human managers, were shareholders could decide on investment opportunities. The DAO was the biggest crowdfunding success in history and at its height was worth 150 million dollars, which was more than ten percent of the total Ethereum market. Investing in the DAO (or entering any other "smart contract") amounts to providing your funds to be run by a computer program. i.e., "code is law", or to use the words the DAO described itself: "The DAO is borne from immutable, unstoppable, and irrefutable computer code". Unfortunately, it turns out that (as we'll see in the next lecture) understanding the behavior of Turing-complete computer programs is quite a hard thing to do. A hacker (or perhaps, some would say, a savvy investor) was able to fashion an input that would cause the DAO code to essentially enter into an infinite recursive loop in which it continuously transferred funds into their account, thereby cleaning out about 60 million dollars out of the DAO. While this transaction was "legal" in the sense that it complied with the code of the smart contract, it was obviously not what the humans who wrote this code had in mind. There was a lot of debate in the Ethereum community how to handle this, including some partially successful "Robin Hood" attempts to use the same loophole to drain the DAO funds into a secure account. Eventually it turned out that the code is

mutable, stoppable, and refutable after all, and the Ethereum community decided to do a "hard fork" (also known as a "bailout") to revert history to before this transaction. Some elements of the community strongly opposed this decision, and so an alternative currency called Ethereum Classic was created that preserved the original history.

11.2 Regular expressions

One of the most common tasks in computing is to *search* for a piece of text. At its heart, the *search problem* is quite simple. The user gives out a function $F : \{0,1\}^* \to \{0,1\}$, and the system applies this function to a set of candidates $\{x_0, \ldots, x_k\}$, returning all the x_i's such that $F(x_i) = 1$. However, we typically do not want the system to get into an infinite loop just trying to evaluate this function! For this reason, such systems often do not allow the user to specify an *arbitrary* function using some Turing-complete formalism, but rather a function that is described by a restricted computational model, and in particular one in which all functions halt. One of the most popular models for this application is the model of regular expressions. You have probably come across regular expresions if you ever used an advanced text editor, a command line shell, or have done any kind of manipulations of text files.[1]

A *regular expression* over some alphabet Σ is obtained by combining elements of Σ with the operation of concatenation, as well as $|$ (corresponding to *or*) and $*$ (corresponding to repetition zero or more times).[2] For example, the following regular expression over the alphabet $\{0,1\}$ corresponds to the set of all even length strings $x \in \{0,1\}^*$ where the digit at location $2i$ is the same as the one at location $2i+1$ for every i:

$$(00|11)* \tag{11.2}$$

The following regular expression over the alphabet $\{a,b,c,d,0,1,2,3,4,5,6,7,8,9\}$ corresponds to the set of all strings that consist of a sequence of one or more of the letters *a*-*b* followed by a sequence of one or more digits (without a leading zero):

$$(a|b|c|d)(a|b|c|d)^*(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^* \tag{11.3}$$

[1] Sections 1.3 and 1.4 in Sipser's book are excellent resources for regular expressions. Sipser's book also discusses the equivalence of regular expressions with *finite automata*.

[2] Common implementations of regular expressions in programming languages and shells typically include some extra operations on top of $|$ and $*$, but these can all be implemented as "syntactic sugar" using the operators $|$ and $*$.

Formally, regular expressions are defined by the following recursive definition:[3]

Definition 11.2 — Regular expression. A *regular expression* exp over an alphabet Σ is a string over $\Sigma \cup \{"(","),"|","*"\}$ that has one of the following forms: 1. $exp = \sigma$ where $\sigma \in \Sigma$

2. $exp = (exp'|exp'')$ where exp', exp'' are regular expressions.

3. $exp = (exp')(exp'')$ where exp', exp'' are regular expressions. (We often drop the parenthesis when there is no danger of confusion and so write this as $exp\ exp'$.)

4. $exp = (exp')*$ where exp' is a regular expression.

Finally we also allow the following "edge cases": $exp = \emptyset$ and $exp = ""$. [4]

[3] Just like recursive functions, we can define a concept recursively. A definition of some class $\mathcal{C}$ of objects can be thought of as defining a function that maps an object o to either $VALID$ or $INVALID$ depending on whether $o \in \mathcal{C}$. Thus we can think of Definition 11.2 as defining a recursive function that maps a string exp over $\Sigma \cup \{"(","),"|","*"\}$ to $VALID$ or $INVALID$ depending on whether exp describes a valid regular expression.

[4] These are the regular expressions corresponding to accepting no strings, and accepting only the empty string respectively.

Every regular expression exp computes a function $\Phi_{exp} : \Sigma^* \to \{0,1\}$, where $\Phi_{exp}(x) = 1$ if x *matches* the regular expression. So, for example if $exp = (00|11)^*$ then $\Phi_{exp}(x) = 1$ if and only if x is of even length and $x_{2i} = x_{2i+1}$ for every $i < |x|/2$. Formally, the function is defined as follows:

1. If $exp = \sigma$ then $\Phi_{exp}(x) = 1$ iff $x = \sigma$.
2. If $exp = (exp'|exp'')$ then $\Phi_{exp}(x) = \Phi_{exp'}(x) \vee \Phi_{exp''}(x)$ where $\vee$ is the OR operator.
3. If $exp = (exp')(exp'')$ then $\Phi_{exp}(x) = 1$ iff there is some $x', x'' \in \Sigma^*$ such that x is the concatenation of x' and x'' and $\Phi_{exp'}(x') = \Phi_{exp''}(x'') = 1$.
4. If $exp = (exp')*$ then $\Phi_{exp}(x) = 1$ iff there are is $k \in \mathbb{N}$ and some $x_0, \ldots, x_{k-1} \in \Sigma^*$ such that x is the concatenation $x_0 \cdots x_{k-1}$ and $\Phi_{exp'}(x_i) = 1$ for every $i \in [k]$.
5. Finally, for the edge cases $\Phi_{\emptyset}$ is the constant zero function, and $\Phi_{""}$ is the function that only outputs 1 on the constant string.

We say that a function $F : \Sigma^* \to \{0,1\}$ is *regular* if $F = \Phi_{exp}$ for some regular expression exp. We say that a set $L \subseteq \Sigma^*$ (also known as a *language*) is *regular* if the function F s.t. $F(x) = 1$ iff $x \in L$ is regular. For example let $\Sigma = \{a, b, c, d, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $F : \Sigma^* \to \{0,1\}$ be the function such that $F(x)$ outputs 1 iff x consists of one or more of the letters *a*-*b* followed by a sequence of one or more digits (without a leading zero). As shown by Eq. (11.3), the function $F : \Sigma^* \to \{0,1\}$ is computable by a regular expression. For example the string *abc*12078 matches the expression of Eq. (11.3)

since $\Phi_{a|b|c|d}(a) = 1$, ,$\Phi_{(a|b|c|d)^*}(bcd) = 1$, $\Phi_{(1|2|\cdots|9)}(1) = 1$, and $\Phi_{(0|\cdots|9)^*}(2078) = 1$.

P The definitions above might not be easy to grasp in a first read, so you should probably pause here and go over it again until you understand why it corresponds to our intuitive notion of regular expressions. This is important not just for understanding regular expressions themselves (which are used time and again in a great many applications) but also for getting better at understanding recursive definitions in general.

We can think of regular expressions as a type of "programming language". That is, we can think of a regular expression exp over the alphabet Σ as a program that computes some function $\Phi : \Sigma^* \to \{0,1\}$.[5] But it turns out that the "halting problem" for these programs is easy: they always halt.

Theorem 11.3 — Regular expression always halt. For every set Σ and $exp \in (\Sigma \cup \{"(","),"|","*"\})*$, if exp is a valid regular expression over Σ then Φ_{exp} is a total function from Σ^* to $\{0,1\}$. Moreover, there is an always halting NAND++ program P_{exp} that computes Φ_{exp}.

Proof. Definition 11.2 gives a way of recursively computing Φ_{exp}. The key observation is that in our recursive definition of regular expressions, whenever exp is made up of one or two expressions exp', exp'' then these two regular expressions are *smaller* than exp, and eventually (when they have size 1) then they must correspond to the non-recursive case of a single alphabet symbol.

Therefore, we can prove the theorem by induction over the length m of exp (i.e., the number of symbols in the string exp, also denoted as $|exp|$). For $m = 1$, exp is a single alhpabet symbol and the function Φ_{exp} is trivial. In the general case, for $m = |exp|$ we assume by the induction hypothesis that we have proven the theorem for $|exp| = 1, \ldots, m-1$. Then by the definition of regular expressions, exp is made up of one or two sub-expressions exp', exp'' of length smaller than m, and hence by the induction hypothesis we assume that $\Phi_{exp'}$ and $\Phi_{exp''}$ are total computable functions. But then we can follow the definition for the cases of concatenation, union, or the star operator to compute Φ_{exp} using $\Phi_{exp'}$ and $\Phi_{exp''}$. ■

The proof of Theorem 11.3 gives a recursive algorithm to evaluate

[5] Regular expressions (and context free grammars, which we'll see below) are often thought of as *generative models* rather than computational ones, since their definition does not immediately give rise to a way to *decide* matches but rather to a way to generate matching strings by repeatedly choosing which rules to apply.

whether a given string matches or not a regular expression. However, it turns out that there is a much more efficient algorithm to match regular expressions. One way to obtain such an algorithm is to replace this recursive algorithm with dynamic programming, using the technique of memoization.[6] It turns out that the resulting dynamic program only requires maintaining a finite (independent of the input length) amount of state, and hence it can be viewed as a finite state machine or finite automata. The relation of regular expressions with finite automata is a beautiful topic, and one we may return to later in this course.

[6] If you haven't taken yet an algorithms course such as Harvard CS 124, you might not know these techniques. This is OK; while the more efficient algorithm is crucial for the many practical applications of regular expressions, it is not of great importance to this course.

11.2.1 *Limitations of regular expressions*

The fact that functions computed by regular expressions always halt is of course one of the reasons why they are so useful. When you make a regular expression search, you are guaranteed that you will get a result. This is why operating systems, for example, restrict you for searching a file via regular expressions and don't allow searching by specifying an arbitrary function via a general-purpose programming language. But this always-halting property comes at a cost. Regular expressions cannot compute every function that is computable by NAND++ programs. In fact there are some very simple (and useful!) functions that they cannot compute, such as the following:

Theorem 11.4 — Matching parenthesis. Let $\Sigma = \{\langle, \rangle\}$ and $MATCHPAREN : \Sigma^* \to \{0,1\}$ be the function that given a string of parenthesis, outputs 1 if and only if every opening parenthesis is matched by a corresponding closed one. Then there is no regular expression over Σ that computes $MATCHPAREN$.

Theorem 11.4 is a consequence of the following result known as the *pumping lemma*:

Theorem 11.5 — Pumping Lemma. Let exp be a regular expression. Then there is some number n_0 such that for every $w \in \{0,1\}^*$ with $|w| > n_0$ and $\Phi_{exp}(w) = 1$, it holds that we can write $w = xyz$ where $|y| \geq 1$, $|xy| \leq n_0$ and such that $\Phi_{exp}(xy^k z) = 1$ for every $k \in \mathbb{N}$.

Proof Idea: The idea behind the proof is very simple (see Fig. 11.1). If we let n_0 be, say, twice the number of symbols that are used in the expression exp, then the only way that there is some w with

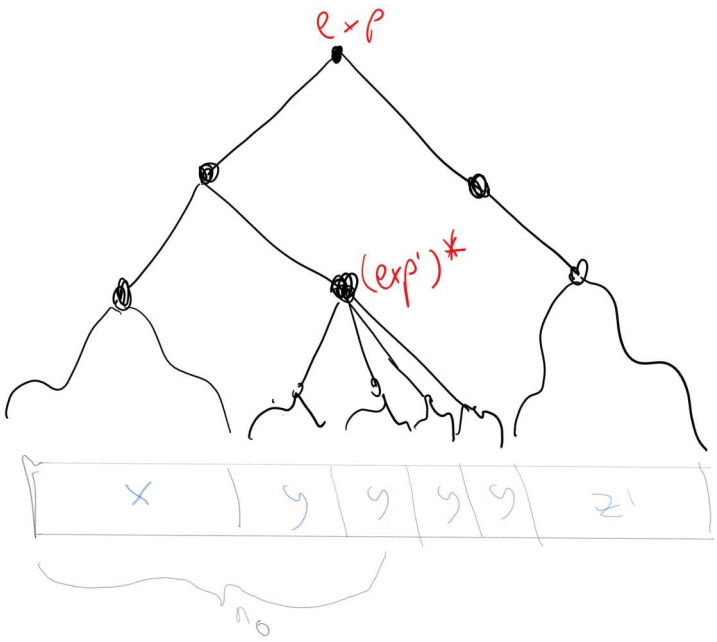

Figure 11.1: To prove the "pumping lemma" we look at a word w that is much larger than the regular expression exp that matches it. In such a case, part of w must be matched by some sub-expression of the form $(exp')^*$, since this is the only operator that allows matching words longer than the expression. If we look at the "leftmost" such sub-expression and define y^k to be the string that is matched by it, we obtain the partition needed for the pumping lemma.

$|w| > n_0$ and $\Phi_{exp}(w) = 1$ is that exp contains the $*$ (i.e. star) operator and that there is a nonempty substring y of w that was matched by $(exp')^*$ for some sub-expression exp' of exp. We can now repeat y any number of times and still get a matching string.

Proof of Theorem 11.5. To prove the lemma formally, we use induction on the length of the expression. Like all induction proofs, this is going to be somewhat lengthy, but at the end of the day it directly follows the intuition above that *somewhere* we must have used the star operation. Reading this proof, and in particular understanding how the formal proof below corresponds to the intuitive idea above, is a very good way to get more comfort with inductive proofs of this form.

Our inductive hypothesis is that for an n length expression, $n_0 = 2n$ satisfies the conditions of the lemma. The base case is when the expression is a single symbol or that it is $\emptyset$ or "" in which case the condition is satisfied just because there is no matching string of length more than one. Otherwise, exp is of the form **(a)** $exp'|exp''$, **(b)**, $(exp')(exp'')$, **(c)** or $(exp'')^*$ where in all these cases the subexpressions have fewer symbols than exp and hence satisfy the induction hypothesis.

In case **(a)**, every string w matching exp must match either exp' or exp''. In the former case, since exp' satisfies the induction hypothesis, if $|w| > n_0$ then we can write $w = xyz$ such that xy^kz matches exp' for every k, and hence this is matched by exp as well.

In case **(b)**, if w matches $(exp')(exp'')$. then we can write $w = w'w''$ where w' matches exp' and w'' matches exp''. Again we split to subcases. If $|w'| > 2|exp'|$, then by the induction hypothesis we can write $w' = xyz$ of the form above such that xy^kz matches exp' for every k and then xy^kzw'' matches $(exp')(exp'')$. This completes the proof since $|xy| \leq 2|exp'|$ and so in particular $|xy| \leq 2(|exp'| + |exp''|) \leq 2|exp$, and hence zw'' can be play the role of z in the proof. Otherwise, if $|w'| \leq 2|exp'|$ then since $|w|$ is larger than $2|exp|$ and $w = w'w''$ and $exp = exp'exp''$, we get that $|w'| + |w''| > 2(|exp'| + |exp''|)$. Thus, if $|w'| \leq 2|exp'|$ it must be that $|w''| > 2|exp''|$ and hence by the induction hypothesis we can write $w'' = xyz$ such that xy^kz matches exp'' for every k and $|xy| \leq 2|exp''|$. Therefore we get that $w'xy^kz$ matches $(exp')(exp'')$ for every k and since $|w'| \leq 2|exp'|$, $|w'xy| \leq 2(|exp'| + |exp'|)$ and this completes the proof since $w'x$ can play the role of x in the statement.

Now in the case **(c)**, if w matches $(exp')^*$ then $w = w_0 \cdots w_t$ where w_i is a nonempty string that matches exp' for every i. If $|w_0| >$

$2|exp'|$ then we can use the same approach as in the concatenation case above. Otherwise, we simply note that if x is the empty string, $y = w_0$, and $z = w_1 \cdots w_t$ then $xy^k z$ will match $(exp')^*$ for every k. ■

> **R** **Recursive definitions and inductive proofs** When an object is *recursively defined* (as in the case of regular expressions) then it is natural to prove properties of such objects by *induction*. That is, if we want to prove that all objects of this type have property P, then it is natural to use an inductive steps that says that if o', o'', o''' etc have property P then so is an object o that is obtained by composing them.

Given the pumping lemma, we can easily prove Theorem 11.4:

Proof of Theorem 11.4. Suppose, towards the sake of contradiction, that there is an expression exp such that $\Phi_{exp} = MATCHPAREN$. Let n_0 be the number from Theorem 11.4 and let $w = \langle^{n_0}\rangle^{n_0}$ (i.e., n_0 left parenthesis followed by n_0 right parenthesis). Then we see that if we write $w = xyz$ as in Theorem 11.4, the condition $|xy| \leq n_0$ implies that y consists solely of left parenthesis. Hence the string xy^2z will contain more left parenthesis than right parenthesis. Hence $MATCHPAREN(xy^2z) = 0$ but by the pumping lemma $\Phi_{exp}(xy^2z) = 1$, contradicting our assumption that $\Phi_{exp} = MATCHPAREN$. ■

The pumping lemma is a very useful tool to show that certain functions are *not* computable by a regular language. However, it is *not* an "if and only if" condition for regularity. There are non regular functions which still satisfy the conditions of the pumping lemma. To understand the pumping lemma, it is important to follow the order of quantifiers in Theorem 11.5. In particular, the number n_0 in the statement of Theorem 11.5 depends on the regular expression (in particular we can choose n_0 to be twice the number of symbols in the expression). So, if we want to use the pumping lemma to rule out the existence of a regular expression exp computing some function F, we need to be able to choose an appropriate w that can be arbitrarily large and satisfies $F(w) = 1$. This makes sense if you think about the intuition behind the pumping lemma: we need w to be large enough as to force the use of the star operator.

> **R** **Regular expressions beyond searching** Regular expressions are widely used beyond just searching. First, they are typically used to define *tokens* in

various formalisms such as programming data description languages. But they are also used beyond it. One nice example is the recent work on the NetKAT network programming language. In recent years, the world of networking moved from fixed topologies to "software defined networks", that are run by programmable switches that can implement policies such as "if packet is SSL then forward it to A, otherwise forward it to B". By its nature, one would want to use a formalism for such policies that is guaranteed to always halt (and quickly!) and that where it is possible to answer semantic questions such as "does C see the packets moved from A to B" etc. The NetKAT language uses a variant of regular expressions to achieve that.

11.3 Context free grammars

If you have ever written a program, you've experienced a *syntax error*. You might have had also the experience of your program entering into an *infinite loop*. What is less likely is that the compiler or interpreter entered an infinite loop when trying to figure out if your program has a syntax error. When a person designs a programming language, they need to come up with a function $VALID : \{0,1\}^* \to \{0,1\}$ that determines the strings that correspond to valid programs in this language. The compiler or interpreter computes $VALID$ on the string corresponding to your source code to determine if there is a syntax error. To ensure that the compiler will always halt in this computation, language designers typically *don't* use a general Turing-complete mechanism to express the function $VALID$, but rather a restricted computational model. One of the most popular choices for such a model is *context free grammar*.

To explain context free grammars, let's begin with a canonical example. Let us try to define a function $ARITH : \Sigma^* \to \{0,1\}$ that takes as input a string x over the alphabet $\Sigma = \{(,),+,-,\times,\div,0,1,2,3,4,5,6,7,8,9\}$ and returns 1 if and only if the string x represents a valid arithmetic expression. Intuitively, we build expressions by applying an operation to smaller expressions, or enclosing them in parenthesis, where the "base case" corresponds to expressions that are simply numbers. A bit more precisely, we can make the following definitions:

- A *number* is either a sequence of digits.[7]
- An *operation* is one of $+,-,\times,\div$

[7] For simplicity we drop the condition that the sequence does not have a leading zero, though it is not hard to encode it in a context-free grammar as well.

- An *expression* has either the form "*number*" or the form "*subexpression1 operation subexpression2*" or "(*subexpression*)".

A context free grammar (CFG) is a formal way of specifying such conditions.[8] We can think of a CFG as a set of rules to *generate* valid expressions. In the example above, the rule $expression \Rightarrow expression \times expression$ tells us that if we have built two valid expressions $exp1$ and $exp2$, then the expression $exp1 \times exp2$ is valid above.

[8] Sections 2.1 and 2.3 in Sipser's book are excellent resources for context free grammars.

We can divide our rules to "base rules" and "recursive rules". The "base rules" are rules such as $number \Rightarrow 0$, $number \Rightarrow 1$, $number \Rightarrow 2$ and so on, that tell us that a single digit is a number. The "recursive rules" are rules such as $number\ Rightarrow1\ number$ that tell us that if we add a digit to a valid number then we still have a valid number.

Definition 11.6 — Context Free Grammar. Let Σ be some finite set. A *context free grammar (CFG) over* Σ is a triple (V, R, s) where V is a set disjoint from Σ of *variables*, R is a set of *rules*, which are pairs (v, z) (which we will write as $v \Rightarrow z$) where $v \in V$ and $z \in (\Sigma \cup V)^*$, and $s \in V$ is the starting rule.

The example above of well-formed arithmetic expressions can be captured formally by the following context free grammar:

- The alphabet Σ is $\{(,), +, -, \times, \div, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- The variables are $V = \{expression\ ,\ number\ ,\ digit\ ,\ operation\}$.
- The rules correspond the set R containing the following pairs:
- $operation \Rightarrow +$, $operation \Rightarrow -$, $operation \Rightarrow \times$, $operation \Rightarrow \div$
- $digit \Rightarrow 0$,..., $digit \Rightarrow 9$
- $number \Rightarrow digit$
- $number \Rightarrow digit\ number$
- $expression \Rightarrow number$
- $expression \Rightarrow expression\ operation\ expression$
- $expression \Rightarrow (expression)$
- The starting variable is $expression$

There are various notations to write context free grammars in the literature, with one of the most common being Backus–Naur form where we write a rule of the form $v \Rightarrow a$ (where v is a variable and a

is a string) in the form `<v> := a`. If we have several rules of the form $v \mapsto a$, $v \mapsto b$, and $v \mapsto c$ then we can combine them as `<v> := a|b|c` (and this similarly extends for the case of more rules). For example, the Backus-Naur description for the context free grammar above is (using ASCII equivalents for operations):

```
operation := +|-|*|/
digit := 0|1|2|3|4|5|6|7|8|9
number := digit|digit number
expression := number|expression operation expression|(
    expression)
```

Another example of a context free grammar is the "matching parenthesis" grammar, which can be represented in Backus-Naur as follows:

```
match := ""|match match|(match)
```

You can verify that a string over the alphabet { `(,)` } can be generated from this grammar (where `match` is the starting expression and `""` corresponds to the empty string) if and only if it consists of a matching set of parenthesis.

11.3.1 *Context-free grammars as a computational model*

We can think of a CFG over the alphabet Σ as defining a function that maps every string x in Σ^* to 1 or 0 depending on whether x can be generated by the rules of the grammars. We now make this definition formally.

Definition 11.7 — Deriving a string from a grammar. If $G = (V, R, s)$ is a context-free grammar over Σ, then for two strings $\alpha, \beta \in (\Sigma \cup V)^*$ we say that β *can be derived in one step* from α, denoted by $\alpha \Rightarrow_G \beta$, if we can obtain β from α by applying one of the rules of G. That is, we obtain β by replacing in α one occurence of the variable v with the string z, where $v \Rightarrow z$ is a rule of G.

We say that β *can be derived* from α, denoted by $\alpha \Rightarrow^*_G \beta$, if it can be derived by some finite number k of steps. That is, if there are $\alpha_1, \ldots, \alpha_{k-1} \in (\Sigma \cup V)^*$, so that $\alpha \Rightarrow_G \alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \cdots \Rightarrow_G \alpha_{k-1} \Rightarrow_G \beta$.

We define the *function computed by* (V, R, s) to be the map $\Phi_{V,R,s} : \Sigma^* \to \{0,1\}$ such that $\Phi_{V,R,s}(x) = 1$ iff $s \Rightarrow^*_G x$.

We say that $F : \Sigma^* \to \{0,1\}$ is *context free* if $F = \Phi_{V,R,s}$ for some

CFG (V, R, s) we say that a set $L \subseteq \Sigma^*$ (also knoan as a *language*) is *context free* if the function F such that $F(x) = 1$ iff $x \in L$ is context free.

A priori it might not be clear that the map $\Phi_{V,R,s}$ is computable, but it turns out that we can in fact compute it. That is, the "halting problem" for context free grammars is trivial, or in other words, we have the following theorem:

Theorem 11.8 — Context-free grammars always halt. For every CFG (V, R, s) over Σ, the function $\Phi_{V,R,s} : \Sigma^* \to \{0,1\}$ is computable. [9]

[9] While formally we only defined computability of functions over $\{0,1\}^*$, we can extend the definition to functions over any finite Σ^* by using any one-to-one encoding of Σ into $\{0,1\}^k$ for some finite k. It is a (good!) exercise to verify that if a function is computable with respect to one such encoding, then it is computable with respect to them all.

Proof. We only sketch the proof. It turns out that we can convert every CFG to an equivalent version that has the so called *Chomsky normal form*, where all rules either have the form $u \to vw$ for variables u, v, w or the form $u \to \sigma$ for a variable u and symbol $\sigma \in \Sigma$, plus potentially the rule $s \to$ "" where s is the starting variable. (The idea behind such a transformation is to simply add new variables as needed, and so for example we can translate a rule such as $v \to u\sigma w$ into the three rules $v \to ur$, $r \to tw$ and $t \to \sigma$.)

Using this form we get a natural recursive algorithm for computing whether $s \Rightarrow_G^* x$ for a given grammar G and string x. We simply try all possible guesses for the first rule $s \to uv$ that is used in such a derivation, and then all possible ways to partition x as a concatenation $x = x'x''$. If we guessed the rule and the partition correctly, then this reduces our task to checking whether $u \Rightarrow_G^* x'$ and $v \Rightarrow_G^* x''$, which (as it involves shorter strings) can be done recursively. The base cases are when x is empty or a single symbol, and can be easily handled. ■

11.3.2 *The power of context free grammars*

While we can (and people do) talk about context free grammars over any alphabet Σ, in the following we will restrict ourselves to $\Sigma = \{0,1\}$. This is of course not a big restriction, as any finite alphabet Σ can be encoded as strings of some finite size. It turns out that context free grammars can capture every regular expression:

Theorem 11.9 — Context free grammars and regular expressions. Let exp be a regular expression over $\{0,1\}$, then there is a CFG (V, R, s) over $\{0,1\}$ such that $\Phi_{V,R,s} = \Phi_{exp}$.

Proof. We will prove this by induction on the length of exp. If exp is an expression of one bit length, then $exp = 0$ or $exp = 1$, in which case we leave it to the reader to verify that there is a (trivial) CFG that computes it. Otherwise, we fall into one of the following case: **case 1:** $exp = exp'exp''$, **case 2:** $exp = exp'|exp''$ or **case 3:** $exp = (exp')^*$ where in all cases exp', exp'' are shorter regular expressions. By the induction hypothesis have grammars (V', R', s') and (V'', R'', s'') that compute $\Phi_{exp'}$ and $\Phi_{exp''}$ respectively. By renaming of variables, we can also assume without loss of generality that V' and V'' are disjoint.

In case 1, we can define the new grammar as follows: we add a new starting variable $s \notin V \cup V'$ and the rule $s \mapsto s's''$. In case 2, we can define the new grammar as follows: we add a new starting variable $s \notin V \cup V'$ and the rules $s \mapsto s'$ and $s \mapsto s''$. Case 3 will be the only one that uses *recursion*. As before we add a new starting variable $s \notin V \cup V'$, but now add the rules $s \mapsto$ "" (i.e., the empty string) and also add for every rule of the form $(s', \alpha) \in R'$ the rule $s \mapsto s\alpha$ to R.

We leave it to the reader as (again a very good!) exercise to verify that in all three cases the grammars we produce capture the same function as the original expression. ■

It turns out that CFG's are strictly more powerful than regular expressions. In particular, as we've seen, the "matching parenthesis" function $MATCHPAREN$ can be computed by a context free grammar, whereas, as shown in Theorem 11.4, it cannot be computed by regular expressoins. However, there are some simple languages that are *not* captured by context free grammars, as can be shown via the following version of Theorem 11.5

Theorem 11.10 — Context-free pumping lemma. Let (V, R, s) be a CFG over Σ, then there is some $n_0 \in \mathbb{N}$ such that for every $x \in \Sigma^*$ with $|\sigma| > n_0$, if $\Phi_{V,R,s}(x) = 1$ then $x = abcde$ such that $|b| + |c| + |d| \leq n_1$, $|b| + |d| \geq 1$, and $\Phi_{V,R,s}(ab^kcd^ke) = 1$ for every $k \in \mathbb{N}$.

Proof. We only sketch the proof. The idea is that if the total number of symbols in the rules R is k_0, then the only way to get $|x| > k_0$ with $\Phi_{V,R,s}(x) = 1$ is to use *recursion*. That is there must be some $v \in V$ such that by a sequence of rules we are able to derive from v the value bvd for some strings $b, d \in \Sigma^*$ and then further on derive from v the string $c \in \Sigma^*$ such that bcd is a substring of x. If try to take the minimal such v then we can ensure that $|bcd|$ is at most some constant depending on k_0 and we can set n_0 to be that constant ($n_0 =$

$10|R|k_0$ will do, since we will not need more than $|R|$ applications of rules, and each such application can grow the string by at most k_0 symbols). Thus by the definition of the grammar, we can repeat the derivation to replace the substring bcd in x with $b^k c d^k$ for every $k \in \mathbb{N}$ while retaining the property that the output of $\Phi_{V,R,s}$ is still one. ■

Using Theorem 11.10 one can show that even the simple function $F(x) = 1$ iff $x = ww$ for some $w \in \{0,1\}^*$ is not context free. (In contrast, the function $F(x) = 1$ iff $x = ww^R$ for $w \in \{0,1\}^*$ where for $w \in \{0,1\}^n$, $w^R = w_{n-1}w_{n-2}\cdots w_0$ is context free, can you see why?.)

R **Parse trees** While we present CFGs as merely *deciding* whether the syntax is correct or not, the algorithm to compute $\Phi_{V,R,s}$ actually gives more information than that. That is, on input a string x, if $\Phi_{V,R,s}(x) = 1$ then the algorithm yields the sequence of rules that one can apply from the starting vertex s to obtain the final string x. We can think of these rules as determining a connected directed acylic graph (i.e., a *tree*) with s being a source (or *root*) vertex and the sinks (or *leaves*) corresponding to the substrings of x that are obtained by the rules that do not have a variable in their second element. This tree is known as the *parse tree* of x, and often yields very useful information about the structure of x. Often the first step in a compiler or interpreter for a programming language is a *parser* that transforms the source into the parse tree (often known in this context as the abstract syntax tree). There are also tools that can automatically convert a description of a context-free grammars into a *parser* algorithm that computes the parse tree of a given string. (Indeed, the above recursive algorithm can be used to achieve this, but there are much more efficient versions, especially for grammars that have particular forms, and programming language designers often try to ensure their languages have these more efficient grammars.)

11.4 *Unrestricted grammars*

The reason we call context free grammars "context free" is because if we have a rule of the form $v \mapsto a$ it means that we can always replace v with the string a, no matter the *context* in which v appears. More generally, we want to consider cases where our replacement rules depend on the context.[10] This gives rise to the notion of *general*

[10] TODO: add example

grammars that allow rules of the form (a, b) where both a and b are strings over $(V \cup \Sigma)^*$. The idea is that if, for example, we wanted to enforce the condition that we only apply some rule such as $v \mapsto 0w1$ when v is surrounded by three zeroes on both sides, then we could do so by adding a rule of the form $000v000 \mapsto 0000w1000$ (and of course we can add much more general conditions).

TO BE CONTINUED, UNDECIDABILITY OF GENERAL GRAMMARS.

11.5 Lecture summary

- The uncomputability of the Halting problem for general models motivates the definition of restricted computational models.
- In restricted models we might be able to answer questions such as: does a given program terminate, do two programs compute the same function?

11.6 Exercises

11.7 Bibliographical notes

[11]

[11] TODO: Add letter of Christopher Strachey to the editor of The Computer Journal. Explain right order of historical achievements. Talk about intuitionistic, logicist, and formalist approaches for the foudnations of mathematics. Perhaps analogy to veganism. State the full Rice's Theorem and say that it follows from the same proof as in the exercise.

11.8 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

11.9 Acknowledgements