Context-free grammars and languages

Basics of string generation methods

What's so great about regular expressions?

A regular expression is a

string representation

of a regular language

This allows "**storing a whole**" **language** as a (single) **string**

Example

Let $L = \{w : w \in \{0,1\}^* \wedge w \text{ ends with } 1\}$

The regular expresssion

$$(0+1)^* \circ 1$$

describes completely this language.

Therefore L, which is an infinite language,
can be "stored" by storing R (a string of length 8).

A language that is not regular

Theorem 8.1: The language over the alphabet {0, 1}

$$L = \{w : (\exists n \in N) \wedge w = 0^n 1^n\}$$ is not regular.

Proof:

By contradiction.

Assume that L is regular. Since L is infinite, its regular expression R must contain the star closure of a non-empty, non-null regular expression S. Since the alphabet is $\{0,1\}$, S must be either:

$0^*, 1^*, (0+1)^*, (0 \circ 1)^*,$ or $(1 \circ 0)^*$.

Now, since $L((0+1)^*) = \{0,1\}^*$, $(0+1)^*$ produces a language that contains L, $S \neq (0+1)^*$.

Proof (cont.)

Since $L((0 \circ 1)^*) = \{\lambda, 01, 0101, 010101, \ldots\}$, $(0 \circ 1)^*$ produces strings that are not in L. Therefore, $S \neq (0 \circ 1)^*$. A similar arguments proves that $S \neq (1 \circ 0)^*$. Assume that $S = 0^*$.

Then, $M = \{w : (\exists x, y \in \{0,1\}^*) \wedge (\forall n \in N) w = x0^n y\}$

must be a sub-language of L. But this is not the case, as well.

Similarly, if $S = 1^*$, the language:

$Q = \{w : (\exists x, y \in \{0,1\}^*) \wedge (\forall n \in N) w = x1^n y\}$

must be a sub-language of L. This is also false.

Therefore, there is no regular expression for L. This contradicts the assumption that L is regular.

Motivating grammars

The language

$$L = \{w : w \in \{0,1\}^* \wedge (\exists n \in N) \wedge w = 0^n 1^n\}$$

shows the existence of non-regular languages.

Can they be described in a single string?

The answer comes with the introduction of grammars.

Grammars are in principle designed for:

describing, **analyzing** and **generating** languages

Definition of context-free grammar

Definition 8.1: A context-free grammar (CFG) is a four-tuple $G = (V, A, R, S)$ where

1. V is a finite set of symbols called variables;
2. A is an alphabet
3. $V \cap A = \phi$
4. $R \subseteq V \times (V \cup A)^*$ is the set of rules

   $(v, w) \in R$ is usually denoted $v \rightarrow w$ and called a production
5. $S \in V$ is the starting or initial variable.

Example

Let $G = (\{S\}, \{0,1\}, R, S)$

where

$$R = \{S \rightarrow 0S1 \mid 01\}$$

Here | means “or”. Thus,

$$S \rightarrow 0S1 \mid 01 \Leftrightarrow S \rightarrow 0S1 \vee S \rightarrow 01$$

And thus,

$$R = \{S \rightarrow 0S1, S \rightarrow 01\}$$

The string derivation process

The generation of a string with CFGs is called **derivation**

Strings are **derived** as follows:

1. Select a production whose left hand side is the **start variable**. Ex: $S \rightarrow 0S1$
2. **Select a variable** in the rightmost side of this production. Ex: in the previous CFG we have no choice but *S*, as *S* is the only variable.

String derivation process (cont.)

3. **Select a production initiated with** the chosen variable. Ex: Assume that we choose:

$$S \rightarrow 01$$

4. **Replace the selected variable with the rightmost side of the selected production**. Ex: Choice 3. yields

$$S \rightarrow 0S1 \rightarrow 0011$$

5. Continue until **no variables are left**. In the example, no variables are left. The derivation ends with 0011

Denoting derivations

Let $G = (V, A, R, S)$ be a context-free grammar. Assume that $w \in (V \cup A)^*$ is a string derived from G through a sequence of n productions.

Then, we write

$$S \xrightarrow[G,n]{} w$$

or, if the number of productions is not important,

$$S \xrightarrow[G]{} w$$

Example

With the CFG of the previous example:

$$S \rightarrow 0S1$$

$$\rightarrow 00S11 \text{ (after replacing } S \text{ with } S \rightarrow 0S1)$$

$$\rightarrow 000111 \text{(after replacing } S \text{ with } S \rightarrow 01)$$

These derivations are denoted:

$$S \xrightarrow[G,1]{} 0S1,$$

$$S \xrightarrow[G,2]{} 00S11, \text{ and}$$

$$S \xrightarrow[G,3]{} 000111, \text{ respectively.}$$

The language of a context-free grammar

Definition 8.1:The **language** of a context-free grammar *G =(V,A,R,S),* denoted *L(G),* is the set of all strings in *A** that are **generated** (this is, derived with) by *G.* Thus,

$$L(G) = \{w : w \in A^* \wedge (\exists n \in N) \wedge S \xrightarrow[G,n]{} w\}$$

Definition 8.2: If a language over an alphabet *A* can be generated by a context-free grammar, then the language is said to be a **context-free language**

Representing context-free languages

Since a CFG is a **four-tuple of finite mathematical objects,** it has string representations. Thus, **context-free languages can be stored** (without loss of information) **as a string representing its context-free grammar**

Unlike regular languages whose regular expressions are easy to find, either **finding a context-free grammar for a language** or **identifying the language of a context-free grammar is, in general, much involved**

Thus, the two main theoretical problems are

1. Given a context – free language, find a grammar for generating it
2. Given a context – free grammar, find its language

Recursive productions

Definition: A production is said to be recursive if a variable on its left hand side occurs on its right hand side. A production is indirectly recursive if a left hand side variable occurs on a right hand side, two or more derivations ahead

Example: Consider the grammar:

G = ({S}, {0, 1}, {S→1|0S}, S)

Then, the productions:

S→0S→01, S→0S→00S→001, etc…are all recursive

Recursive grammars

Definition: A grammar is said to be recursive if it contains either a recursive or an indirectly recursive production

A first theoretical principle:

A grammar for an infinite language must be recursive

Attacking the first theoretical problem

This principle contributes to the search for an answer to problem 1, this is, finding the grammar of a language

First, if the language is finite, there is a simple (although not too elegant) solution:

1. Put variables and strings in one – to – one correspondence
2. Write a first rule assigning the star variable to each one of the variables created in 1.
3. Write a rule assigning each of the variables created in 1. its corresponding string

Attacking the first theoretical problem

Now, if the language is infinite, it is the union of a finite (possible empty) and an infinite language. Then, we can treat the finite part independently, with the previous method. As for the infinite part,

1. Identify all recursive patterns in the (infinite part of the) language
2. Derive from these patterns, the shortest possible productions generating strings
3. Use 2. to derive the corresponding rules

Illustration

Consider the language

L = {λ, 01, 0101, 010101, 01010101… }

1. Recursive pattern: 01 → 0101 → …
2. How do we produce it? Let V be a variable. Then, the pattern is produced by: 01 → 01V or 01 → V01
3. A recursive rule for creating this pattern is: S → 01S, where S is the start variable

The second problem

Definition: A set is said to be inductively defined if it can be fully characterized with the next three rules:

a) There is a starting element

b) There is a rule for constructing a new element from an existing element

c) There are no more elements in the set, except those constructed in a) and b).

Second theoretical principle: Any language defined by a grammar is an inductively defined set

Attacking the second problem

Now we are given a grammar and the question is: What is its language? The second principle suggests describing the language through an inductive definition. The method is as follows:

1. Identify all derivations of the grammar that contain no recursive productions
2. Put all strings produced by derivations in 1. as the base case of the inductive definition
3. Transform every recursive production into a rule for constructing a new element from existing ones

Illustration

Consider the context – free grammar:

G = ({S, V}, {0, 1}, R, S)

where R = {S → λ | 0V, V → 1 | 1V}

1. Derivations that don't contain recursive productions: S→λ, S→0V→01
2. **Base case: λ, 01 in L(G)**
3. The only recursive production of G is V → 1V. This production is used in derivations of the form of: S→0V→01V→011V→… The corresponding inductive rule is thus,

   Induction: If 0y is in L(G) then 01y is in L(G)

Regular languages revisited

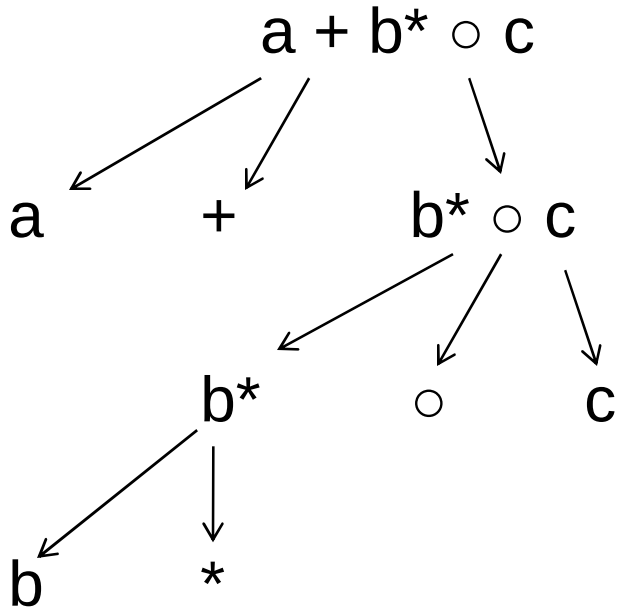
Next we'll prove that the class of all regular languages is a sub – class of the class of all context – free languages

Our strategy is to find a method for constructing a grammar out of a regular expression

Parsing regular expressions

Is the same as analyzing a regular language: a **recursive process** for identifying the expression's **basic regular components**. The recursion **decomposes first the lowest hierarchy operation**

Example: The parse tree of a + b* ○ c is:



Method 8.1:

Context-free grammar generating the language of a regular expression

1. Write the parsing **tree** of the regular expression
2. Replace the **root** and each **internal node that is not an operation**, with a **variable**
3. Keep the terminals as the leaves, except for terminals under **a sub-tree rooted by a variable** *V* **having the operation** * **as child.** In these cases, replace the terminal *a,* with the string *aU*, where *U* is a **variable**
4. Write the rules as follows:

   4.1 The **root** is the **starting variable**, and each **parent-children edge correspond to a rule**. For building these rules:

   4.2 Assign to each triple of **sibling nodes** *V,* ○, *U* the string *VU*

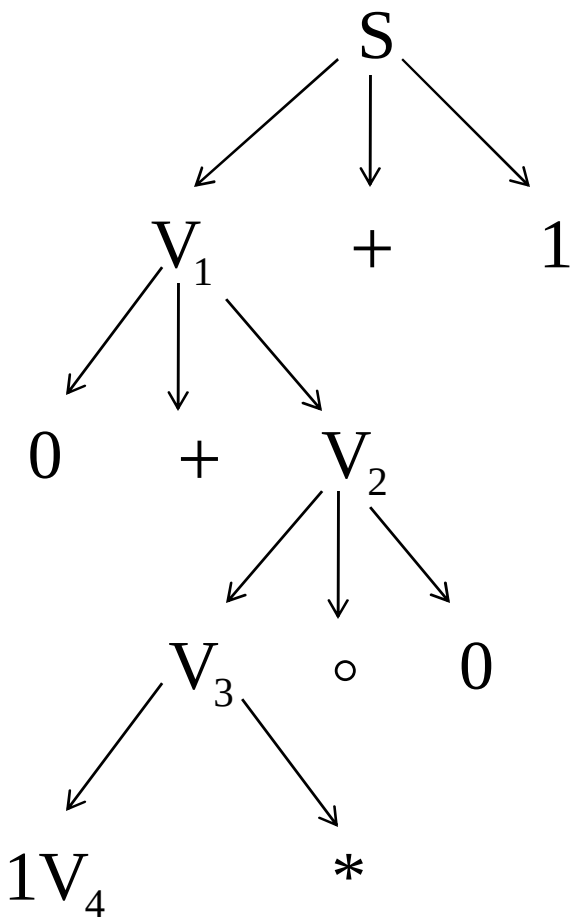
   4.3 Assign to each triple of **sibling nodes** *V,* +, *U* the string *V* | *U*

   4.4 For each node *V* having *U* and * as children, create $V \rightarrow \lambda | U$

   4.5 For each **leaf** *aU* in a **sub-tree rooted by a variable** *V* **having * as child**, create $U \rightarrow V$ and the rule $U \rightarrow \lambda$

Example 1: Consider R = 0+1*○0+1

Parse tree after replacements



Corresponding set of rules:

$$R = \{ \quad S \rightarrow V_1 \mid 1,$$
$$V_1 \rightarrow 0 \mid V_2,$$
$$V_2 \rightarrow V_3 0,$$
$$V_3 \rightarrow 1V_4 \mid \lambda,$$
$$V_4 \rightarrow V_3 \mid \lambda \quad \}$$

Just checking: Two productions with the previous grammar

The resulting grammar is:

$$G=(\{S,V_1,...,V_7\},\{0,1\},R,S)$$

Where *R* is the set of rules defined in slide 24

- The strings *w*=0 and *w* = 1, which are in *L(0+1*○0+1)*, are produced by *G* as follows:

$$S \to 1$$

$$S \to V_1 \to 0$$

- The string *w*= 1110, which is also in *L(0+1*○0+1)*, is produced by *G* as follows:

$$S \to V_1 \to V_2 \to V_3 0 \to 1V_4 0 \to 1V_3 0$$
$$\to 11V_4 0 \to 11V_3 0 \to 111V_4 0 \to 1110$$

Regular languages and context – free languages

Theorem: Every regular language is a context – free language

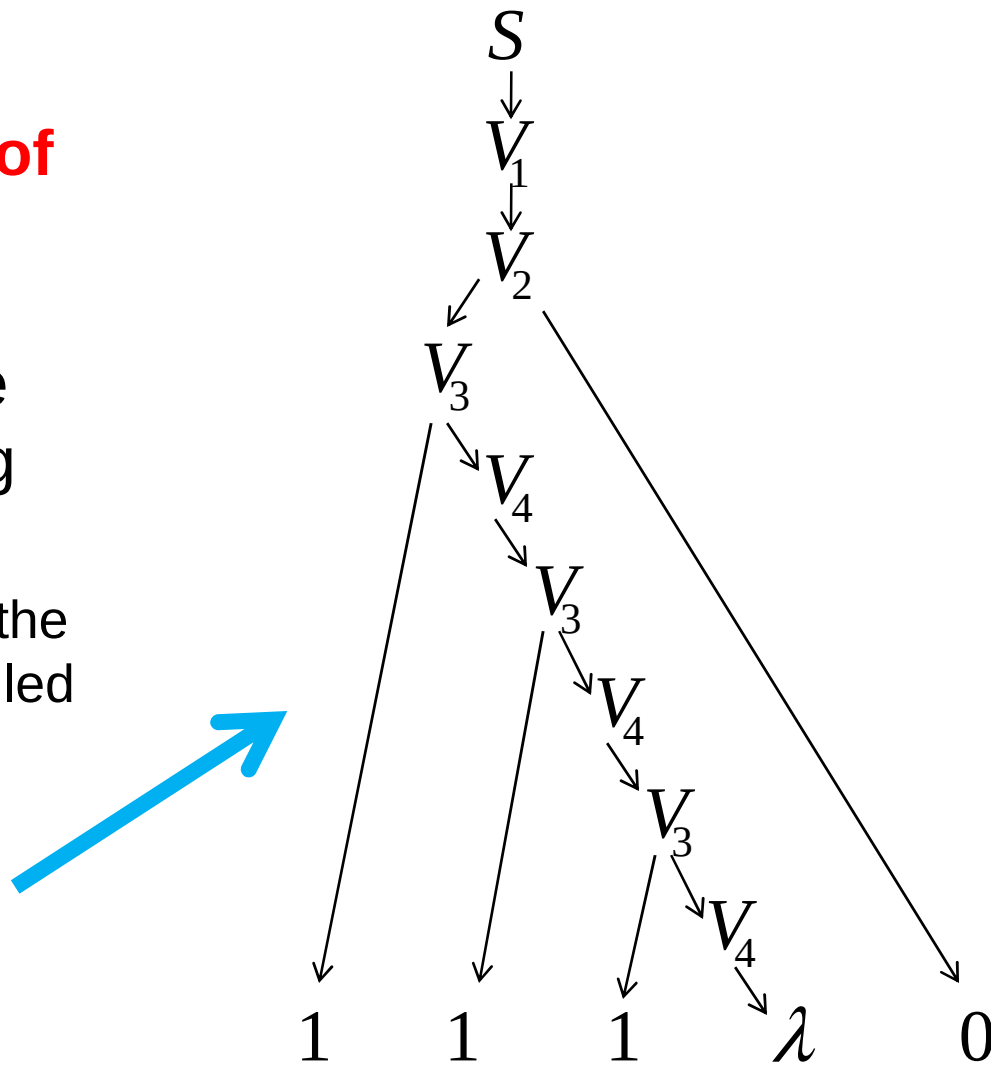
Proof: Let L be a regular language. Then, there is at least one regular expression R, such that L = L(R). But then, L = L(G) where G is the context – free grammar obtained by applying Method 8.1 to R has a regular expression. Thus, L is context - free

String parsing

Definition: The **parsing of a given string** is the generation of a **tree representation** of the derivation of the string by a grammar.

- The tree representing the string generation is called **parse tree**.

Example: Parse tree for w=1110 under the previous grammar



Grammars vs. regular expressions

Unlike a regular expression whose role is mostly descriptive, the grammar of a regular language (as any other grammar) **constitutes** (the basis of) **a computational method** (not necessarily the most efficient) for **generating the strings** of the language

INDEED, underlying grammars is the idea of storing a language not just as a (static) mathematical description but **as a computational method for producing its strings**

BE FULLY AWARE OF THIS CONCEPTUAL FACT!!!!