Chapter 3
Context-free Languages

We now turn to another class of languages, the *context-free languages,* and the corresponding automata, the *pushdown automata (PDA),* actually non-deterministic PDA, which is not equivalent to PDA.

Context-free languages are very important to us, as almost (?) all the programming languages that we have been using, including *C*, *Java*, *Python*, belong to this class.

Question: Is a program `P` grammatically correct in terms of L? ☹

This is the very first step of what a compiler would do. We will see that this *acceptance problem is decidable* when L is context free. ☺

Still remember grammar?

The following is an example of a (context-free) grammar, G_1,

$$\begin{array}{rcl} A & \rightarrow & 0A1 \\ A & \rightarrow & B \\ B & \rightarrow & \sharp \end{array}$$

Generally speaking, a grammar consists of a set of *substitution rules,* each of which comprises a variable on the left and a string of variables and other symbols, called *terminals,* on the right.

Variables are often represented by capital letters, while terminals represented by lowercase letters, numbers and other symbols.

One of the variables is designated as the *start symbol,* where everything starts... .

From a grammar...

A grammar describes a language by generating each string of that language in the following way:

1. Write down the start variable.

2. Find a variable that is written down and a rule that starts with that variable. Replace the variable with the right-hand side of that rule.

3. Repeat step 2 until no variables are left.

The language generated by such a grammar is the collection of all the strings so generated.

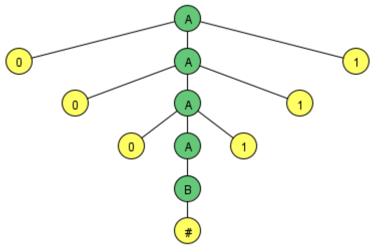
There must be a lot of such strings! ☹

An example

G_1 generates the string 000♯111 in the following *derivation:*

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111$$
$$\Rightarrow 000B111 \Rightarrow 000\sharp111.$$

The derivation of a string can also be represented by using a *parse tree.* Here is one generated with *JFLAP*.



Should we play with *JFLAP* again? ☺

Homework: Exercises 2.2 and 2.9.

...to its language

All strings generated via derivations constitute the *language of the grammar*, G, and denoted as $L(G)$.

For example, $L(G_1) = \{0^n \sharp 1^n : n \geq 0\}$.

Below is another context-free grammar, G_2.

$$\begin{array}{rcl}
\langle S \rangle & \Rightarrow & \langle NP \rangle \langle VP \rangle \\
\langle NP \rangle & \Rightarrow & \langle CN \rangle | \langle CN \rangle \langle PP \rangle \\
\langle VP \rangle & \Rightarrow & \langle CV \rangle | \langle CV \rangle \langle PP \rangle \\
\langle PP \rangle & \Rightarrow & \langle PREP \rangle \langle CN \rangle \\
\langle CN \rangle & \Rightarrow & \langle ARTICLE \rangle \langle NOUN \rangle | \langle NOUN \rangle \\
\langle CV \rangle & \Rightarrow & \langle VERB \rangle | \langle VERB \rangle \langle NP \rangle \\
\langle ARTICLE \rangle & \Rightarrow & a|the \\
\langle NOUN \rangle & \Rightarrow & boy|girl|flower|telescope \\
\langle VERB \rangle & \Rightarrow & likes|sees \\
\langle PERP \rangle & \Rightarrow & with
\end{array}$$

Question: What is $L(G_2)$?

Answer: It takes a while to find out, but...

...the following are some of the strings that can be generated by $L(G_2)$.

```
a boy sees
the boy sees a flower
a girl with a flower sees the boy
```

Below is a derivation that derives the first string.

$$\begin{aligned}
\langle S\rangle &\Rightarrow \langle NP\rangle\langle VP\rangle \\
&\Rightarrow \langle CN\rangle\langle VP\rangle \\
&\Rightarrow \langle ARTICLE\rangle\langle NOUN\rangle\langle VP\rangle \\
&\Rightarrow \text{a } \langle NOUN\rangle\langle VP\rangle \\
&\Rightarrow \text{a boy } \langle VP\rangle \\
&\Rightarrow \text{a boy } \langle CV\rangle \\
&\Rightarrow \text{a boy } \langle VERB\rangle \\
&\Rightarrow \text{a boy sees}
\end{aligned}$$

Question: Did we go through this stuff in *CS CS2010 Fundamentals of Computing*?

What is a *C* identifier?

As we learned in *CS 2470 System Programming in C*, the rule of composing a *C* identifier could be the following:

1. An identifier is a sequence of letters, digits, and underscore.

2. An identifier must start with a letter or an underscore.

3. Identifiers allow both upper- and lower-case letters.

These rules can be specified with a grammar.

$$\begin{array}{rcl} \langle id \rangle & \rightarrow & \langle letter \rangle \langle rest \rangle | \langle undrscr \rangle \langle rest \rangle \\ \langle rest \rangle & \rightarrow & \langle letter \rangle \langle rest \rangle | \langle digit \rangle \langle rest \rangle | \langle undrscr \rangle \langle rest \rangle | \epsilon \\ \langle letter \rangle & \rightarrow & a|b|\cdots|z|A|B|\cdots|Z \\ \langle digit \rangle & \rightarrow & \mathsf{0}|\mathsf{1}|\cdots|\mathsf{9} \\ \langle undrscr \rangle & \rightarrow & \_ \end{array}$$

In the above 'ϵ' stands for an empty string.

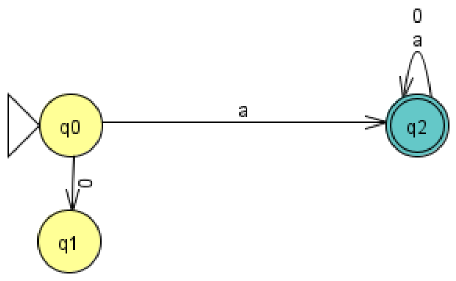
How does *it* know?

A *C* compiler accepts "`a0`" as a legitimate identifier since it can generate the following derivation:

$$\begin{array}{rcl} \langle id \rangle & \rightarrow & \langle letter \rangle \langle rest \rangle \\ & \rightarrow & \mathrm{a}\langle rest \rangle \\ & \rightarrow & \mathrm{a}\langle digit \rangle \langle rest \rangle \\ & \rightarrow & \mathrm{a0}\langle rest \rangle \\ & \rightarrow & \mathrm{a0} \end{array}$$

We can also use the following NFA, where 'a' stands for any letter, and '0' any digit.

$$q_0 \xrightarrow{a} q_2 \xrightarrow{0} q_2.$$



Assignment: Have a look at the grammar rules of *C*, *Java*, *Python* on the course page, where ":" means "$\rightarrow$".

Formal definition

Definition: A *context-free grammar* is a 4-tuple (V, Σ, R, S), where
1. V is a finite set of *variables,*
2. Σ is a finite set of *terminals,*
3. R is a finite set of *rules,* each of which contains a variable and a string of variables and terminals, $A \to w$, and
4. S $(\in V)$ is the *start variable.*

If u, v and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv yields uwv, written as $uAv \Rightarrow uwv$. We also write $u \overset{*}{\Rightarrow} v$ if either $u = v$ or there is a sequence $u_1, u_2, \ldots, u_k$, $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v.$$

The language of the grammar is $\{w : S \overset{*}{\Rightarrow} w\}$.

Any language that can be generated by a context-free grammar is called a *context-free language.*

Homework assignment

1. Revisit the *JFLAP* simulator software, and check out its *Grammar* component, especially the *Brute Force Parse* and *User Control Parse* methods, with the latter you can choose the rule that you want to apply. You might want to consult the tutorial.

2. Do Exercise 2.4 (b, c, e, f) and 2.6 (b) with *JFLAP.*

3. Use *JFLAP* to check out your results.

4. When you hand in this batch of homework, you have to attach screen shots and *JFLAP* program files that can be run in the *JFLAP* simulator.

Examples

Let $G_{1.5} = (\{S\}, \{a, b\}, R, S)$, where R is the follows:

$$S \rightarrow aSb|\epsilon.$$

Then, $L(G_{1.5}) = \{a^n b^n | n \geq 0\} = B$.

Let $G_3 = (\{S\}, \{a, b\}, R, S)$, where R is the follows:

$$S \rightarrow aSb|SS|\epsilon.$$

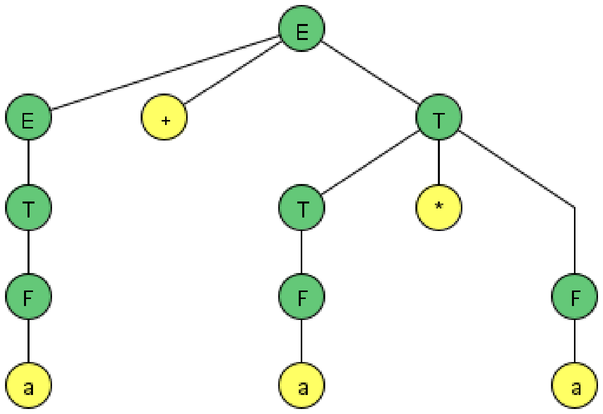
$L(G_3)$ contains strings such as `abab`, `aaabbb`, and `aababb`. If you interpret a as the left parenthesis and b the right one, then $L(G_3)$ is the language of all strings of properly nested parentheses, an important DFS property as mentioned in *CS 3221.*

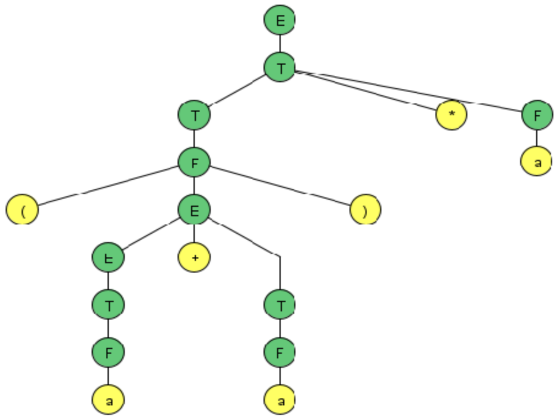
Consider $G_4 = (\{E, T, F\}, \{a, +, \times, (,)\}, R, \{E\})$, where R contains the following rules:

$$\begin{aligned}
\langle E \rangle &\rightarrow \langle E \rangle + \langle T \rangle | \langle T \rangle \\
\langle T \rangle &\rightarrow \langle T \rangle \times \langle F \rangle | \langle F \rangle \\
\langle F \rangle &\rightarrow (\langle E \rangle) | a
\end{aligned}$$

IS G_4 good enough?

Grammar G_4 describes a segment of a programming language concerned with arithmetic expressions. Here are the respective *parse trees* for $a + a \times a$ and $(a + a) \times a$.





Notice that in the tree for $a + a \times a$, the subexpression $a \times a$ is grouped together as one operand, while in the tree for $(a+a) \times a$, $a+a$ is grouped together, presumably, because of the parentheses. *Thus, G_4 does capture the required precedence and its override.* ☺

Homework: Exercise 2.1. Is $a+a+a$ correctly structured? Why? *JFLAP* is eager to help....☺

Ambiguity

Some grammar generates a string in more than one ways. Such a string thus has several different *parsing trees*, leading to different *meanings.* We could say that the *ambiguity coefficient* of such a word in this grammar is at least 2.

This phenomenon is undesirable for certain applications, particularly, in languages, either natural or artificial, *definitely not in programming languages.*

Question: What do you mean?

If a grammar generates the same string in several different ways, we say that the string is derived *ambiguously* in that grammar.

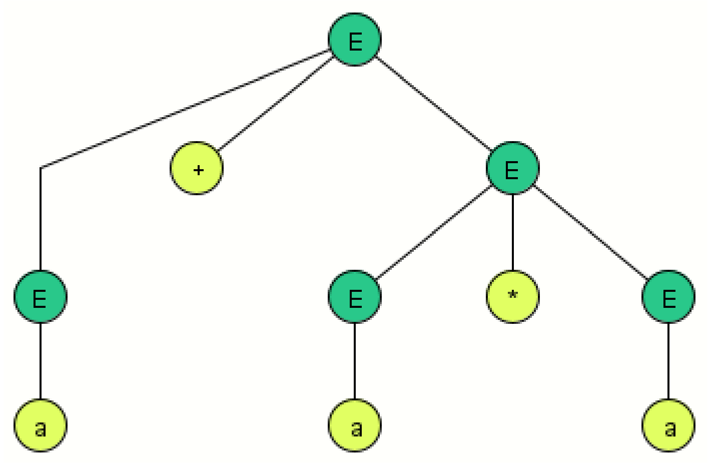
If a grammar generates any string ambiguously, we say the grammar itself is *ambiguous* ☹

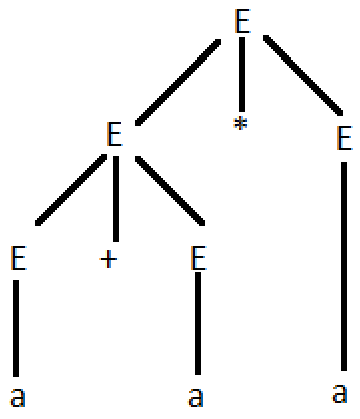
Examples

Assume that grammar G_5 contains the following rule:

$$\begin{aligned}\langle E\rangle &\rightarrow \langle E\rangle + \langle E\rangle | \langle E\rangle \times \langle E\rangle \\ \langle E\rangle &\rightarrow (\langle E\rangle) | a\end{aligned}$$

then, it generates $a + a \times a$ ambiguously, because there are two different derivation trees.





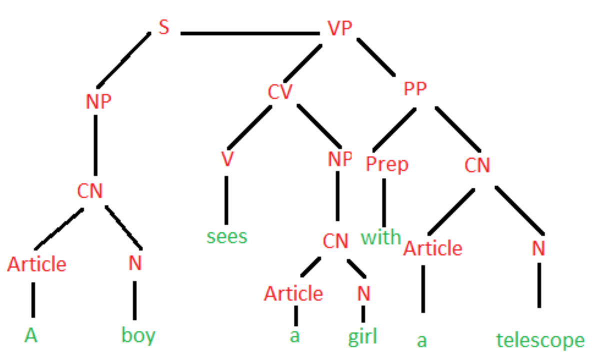
Question: Why did I not use *JFLAP* to draw the chart on the right? ☺

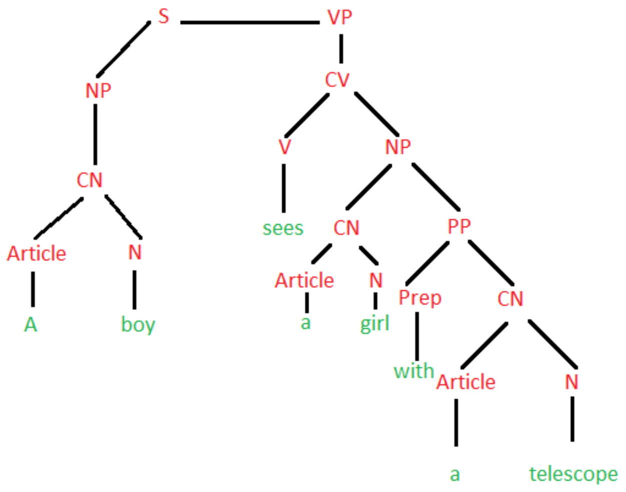
Answer: It did not let me. ☹

What do you mean by saying...

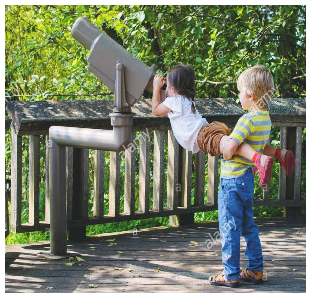
... “A boy sees a girl with a telescope”?

G_2 is ambiguous, just as any natural language, as it generates two parse trees.





These two trees lead to two different meanings.



An exception

Sometimes, a string can have two different derivations but the same parse tree. That is why *the meaning of a string is determined by the final parse tree, but not the construction process of the tree.* Then, its ambiguity coefficient is still one.

For example, we can apply G_4 to derive $a + a * a$ in two different ways, but both sharing the same parse tree. This is *not* considered as ambiguous. ☺

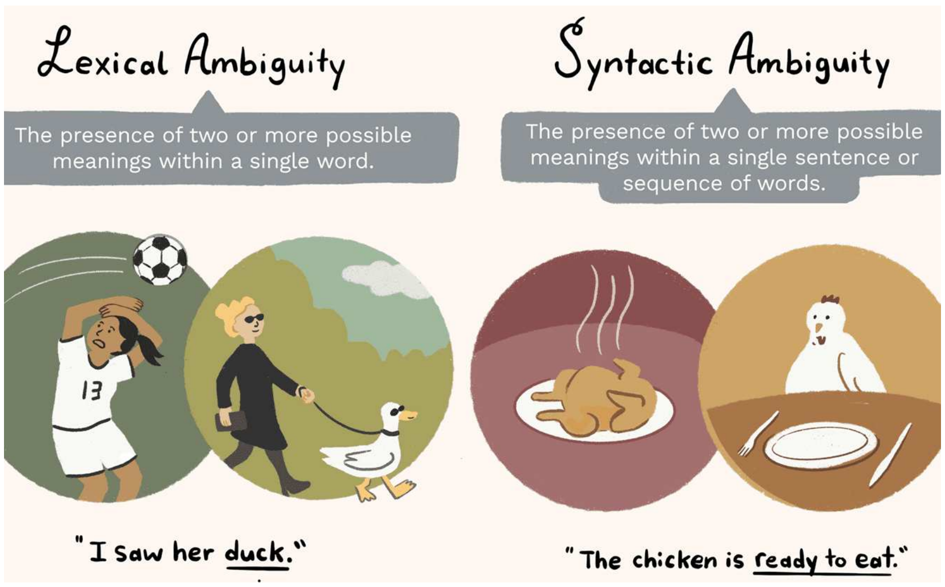
To concentrate on the structural issue, we define the derivation of a string in a grammar is a *leftmost derivation* if at every step the leftmost remaining variable is the one replaced.

The example on Page 6 gives a demo of such a leftmost derivation.

Really ambiguous

Definition: A string w is derived *ambiguously* in a context-free grammar G if it has at least two different leftmost derivations. A grammar is ambiguous if it generates some string ambiguously. G_2 is such a grammar.

If a language can only be generated by an ambiguous grammar, it is called *inherently ambiguous.* English is such a language. ☹


Lexical Ambiguity
The presence of two or more possible meanings within a single word.
Syntactic Ambiguity
The presence of two or more possible meanings within a single sentence or sequence of words.
13
"I saw her duck."
"The chicken is ready to eat."

Chomsky normal form

Chomsky normal form is one of the simplest and most useful forms of a context-free grammar.

Definition: a context-free grammar is in *Chomsky normal form* if every rule is of the form:

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \\ S &\rightarrow \epsilon, \end{aligned}$$

where a is any terminal and A, B and C are any variables, except that neither B nor C can be the start variable.

It is clear that any grammar in Chomsky normal form is context free. It turns out that the other side also holds true:

Theorem 2.6: Any context-free language is generated by a context-free grammar in Chomsky normal form.

Proof: *The proof is done via construction,* i.e., given any context-free grammar, we gradually replace all the "illegal" rules with those that are both legal and equivalent. At the end, we come up with a grammar in Chomsky normal form that is equivalent to the original one.

1. Add in a new start variable S_0 and the rule $S_0 \rightarrow S$, where S is the old start symbol. *Thus, S_0 won't occur on the right of any rule.*

This is similar to the GNFA construction (Cf. Page 90 of the FA notes)

2. Remove any rule in the form of $A \rightarrow \epsilon$, and then for each occurrence of A on the right-hand side of a rule, add a new rule with the occurrence removed, *reflecting all the effects of $A \rightarrow \epsilon$.*

For example, if we have a rule $R \rightarrow uAvAw$, before we remove $A \rightarrow \epsilon$, we would add in three: $R \rightarrow uvAw$, $R \rightarrow uAvw$, and $R \rightarrow uvw$.(?)

3. Remove any unit rule $A \rightarrow B$. Then whenever a rule $B \rightarrow u$ appears, we add the rule $A \rightarrow u$, unless this is a rule previously removed.

Now, every rule has at least two symbols on the right.

4. Replace each rule $A \rightarrow u_1u_2 \cdots u_k, k \geq 3$, with the following ones: $A \rightarrow u_1A_1, A_1 \rightarrow u_2A_2$, $\ldots, A_{k-2} \rightarrow u_{k-1}u_k$.

Every rule has two symbols on its right,...

Finally, if u_i is a terminal symbol, we replace it with a new variable U_i and add in a rule $U_i \rightarrow u_i$. $\square$

Question: I want to see... .

Question: Can *JFLAP 7.1* do such a conversion yet? ☹

An example will help...

Let a CFG G_6 be the following grammar:

$$\begin{aligned} S &\rightarrow ASA|aB \\ A &\rightarrow B|S \\ B &\rightarrow b|\epsilon \end{aligned}$$

1. Since S does occur on the right side of some rule, we add a new start symbol, S_0.

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA|aB \\ A &\rightarrow B|S \\ B &\rightarrow b|\epsilon \end{aligned}$$

2.1. In Step 2, we remove first $B \rightarrow \epsilon$, which leads to the following:

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA|aB|a \\ A &\rightarrow B|S|\epsilon \\ B &\rightarrow b \end{aligned}$$

2.2. We also remove $A \rightarrow \epsilon$ and get the following:

$$\begin{array}{rcl} S_0 & \rightarrow & S \\ S & \rightarrow & ASA|aB|a|SA|AS|S \\ A & \rightarrow & B|S \\ B & \rightarrow & b \end{array}$$

3.1. In Step 3, we remove a unit rule $S \rightarrow S$.

$$\begin{array}{rcl} S_0 & \rightarrow & S \\ S & \rightarrow & ASA|aB|a|SA|AS \\ A & \rightarrow & B|S \\ B & \rightarrow & b \end{array}$$

3.2. Removal of $S_0 \rightarrow S$ leads to the following:

$$\begin{array}{rcl} S_0 & \rightarrow & ASA|aB|a|SA|AS \\ S & \rightarrow & ASA|aB|a|SA|AS \\ A & \rightarrow & B|S \\ B & \rightarrow & b \end{array}$$

3.3. Removing $A \to B$ and $A \to S$ lead to the following, respectively:

$$\begin{array}{rcl} S_0 & \to & ASA|aB|a|SA|AS \\ S & \to & ASA|aB|a|SA|AS \\ A & \to & b|S \\ B & \to & b, \text{ and then,} \end{array}$$

$$\begin{array}{rcl} S_0 & \to & ASA|aB|a|SA|AS \\ S & \to & ASA|aB|a|SA|AS \\ A & \to & b|ASA|aB|a|SA|AS \\ B & \to & b \end{array}$$

4. After introducing additional rules and variables, we eventually obtain the following:

$$\begin{array}{rcl} S_0 & \to & AA_1|UB|a|SA|AS \\ S & \to & AA_1|UB|a|SA|AS \\ A & \to & b|AA_1|UB|a|SA|AS \\ A_1 & \to & SA \\ U & \to & a \\ B & \to & b \end{array}$$

Homework: Exercise 2.14 and Problem 2.26.

Big deal?

Problem 2.26. If G is a grammar in Chomsky normal form, then, for any string $w \in L(G)$, exactly $2|w|-1$ steps are required for any derivation of w.

Let $G(V, T, R, S)$ be the context-free grammar for a programming language, $G'(V', T, R', S_0)$ be its Chomsky equivalent, and let p be a program, then p is grammatically correct in terms of G iff

$$S \overset{*}{\Rightarrow} p \text{ iff } S_0 \overset{*}{\Rightarrow} p \text{ iff } S_0 \overset{2|p|-1}{\Rightarrow} p.$$

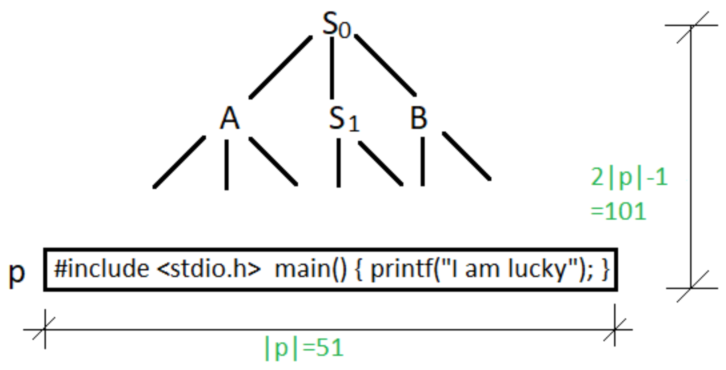
Given a grammar,G, and a string, w, it can be *effectively* decided whether or not w can be derived from G.

This is the basis of a compiler for any high-level programming language.

The CYK algorithm, as provided in *JFlap* is based on CNF. You will tell the difference with G_3 when trying `aab`.

Next step

The issue is that there are many possible rules to apply in each step: Which rule shlould we use to derive A, S_1 and B? If we have two options for each, we will have 2^3 possible derivations in layer 3,..., $2^{2|p|-1}$ possible ones in layer $2^{2|p|-1}$.



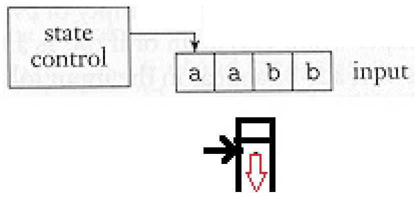
Thus, this process is not *efficient,* as an exponential number of derivations have to be tried ☹ to decide whether or not p can be derived.

2020 ACM A.M. Turing Award was awarded to Alfred Aho and Jeffrey Ullman, partly because of their work in coming up with various *efficient compilers.* ☺ (Cf. Course page)

Pushdown automata

Now, we present the automata corresponding to the context-free languages: the (nondeterministic) *pushdown automata (N)PDA.*

NPDA's are quite similar to the NFA's, except that they also have a *boundless stack* as an extra memory component, which allows the automata to accept some non-regular languages.



As we will see later, NPDAs are equivalent to the context-free grammars, which gives us two options for proving that a language is context free: we can either write a context-free grammar to *generate* such a language, or to produce a pushdown automata to *accept* it.

This is the practical side of a compiler for every programming language, including $C\flat$. ☺

How does an NPDA work?

While reading symbols, one at a time, from an input tape, an NPDA can also write symbols on the stack and read them back later.

Writing a symbol “pushes down” all the other symbols currently on the stack. Reading a symbol “pops” it. Just like the ordinary stack, at any moment, an element can only be pushed at the top and only the top element can be popped off, the famous FIFO property.

As we saw earlier, DFA is equivalent to NFA. But NPDA is *not* equivalent to PDA.

Later on, we will see, NTM (Turing Machine) is also equivalent to TM, while no body knows yet whether LBA (Linearly bounded automata) is equivalent to NLBA, the automata that accepts the class of *context sensitive languages.*

An example

Let’s see how an NPDA accepts $B = \{0^n1^n|n \geq 0\}$, which, as we have shown at the end of the last chapter, is not regular. ☹

Given an input string consisting of ‘0’ and ‘1’, the NPDA pushes all the‘0’s it reads into its stack, until its reads a ‘1’, when it starts to pop off a ‘0’ for this ‘1’. The process continues.

It accepts this input string if and only if all the input symbols are read exactly when the stack becomes empty for the first time. ☺

In other words, it will reject the input if 1) there is no ‘0’(s) left in the stack although there are still ‘1’ unread; 2) when it reads all the symbols, there are still ‘0’s left; or 2) ☹

Formal definition

The general situation in an NPDA is that, *in a certain state, this NPDA reads in an input symbol, observes the top of the stack, then gets into a new state and possibly replaces the original top element of the stack with a new one.*

Because we allow NPDA, there could be multiple legal next moves. Finally, the alphabet of the stack can be different from that of the inputs.

Definition: An NPDA is defined with $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite, and
1. Q is the set of *states,*
2. Σ is the *input alphabet,*
3. Γ is the *stack alphabet,*
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the *transition function,*
5. q_0 is the *start state,* and
6. $F \subseteq Q$ is the set of accept states.

How does an NPDA compute?

An NPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. It accepts an input $w = w_1 w_2 \cdots w_n$, for all $i, w_i \in \Sigma$, if there exists a sequence $(r_0, r_1, \ldots, r_n)$, for all $i, r_i \in Q$; and a sequence $(s_0, s_1, \ldots, s_n), s_i \in \Gamma^*, i \in [0, n]$, such that

1. $r_0 = q_0$ and $s_0 = \epsilon$.

2. For $i = 0, \ldots, n-1$, we have that

$$(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a),$$

where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$.

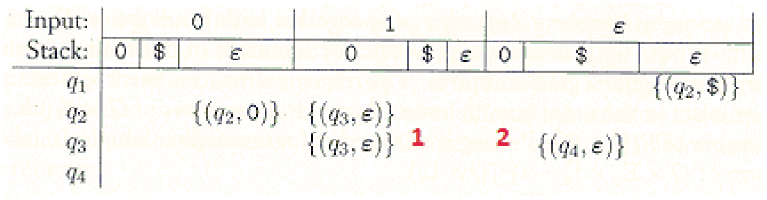
3. $r_n \in F$.

In the above, all the elements of Γ^*, such as t, represent the *contents of the stack,* where s_i represents a symbol in the stack, and $s_i t$ indicates that s_i is above t in the stack.

Examples of NPDA

Below shows an NPDA that accepts the language $\{0^n 1^n | n \geq 0\}$.

$PDA_1 = (\{q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, \$\}, \delta, q_1, \{q_1, q_4\})$, where δ is given by the following table.



Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$	1		2	$\{(q_4, \varepsilon)\}$	
q_4									

$$\delta(q_1, \epsilon, \epsilon) = (q_2, \$), \delta(q_2, 0, \epsilon) = (q_2, 0), \delta(q_2, 1, 0) = (q_3, \epsilon)$$
$$\delta(q_3, 1, 0) = (q_3, \epsilon), \delta(q_3, \epsilon, \$) = (q_4, \epsilon)$$

Note that a transition rule $(q, a, b) \mapsto (q', c)$, means that, if the PDA, in state q, reads an input a, while the top of the stack is b, then it moves to q', and replaces b with c.

Question: What is going on in Case 1 and 2 ☺ in the above chart?

Answer: Too many 1's in Case 1; and too many 0's in Case 2. (Cf. Page 28)

All the cases ...

With $(q, a, b) \mapsto (q', c)$, one of the following happens:

1. If $a = \epsilon$, then PDA makes this move without reading in any input; Otherwise, it reads a; e.g., $\delta(q_2, 0, \epsilon) = (q_2, 0)$, but $\delta(q_3, \epsilon, \$) = (q_4, \epsilon)$

2. If $b \neq \epsilon$ and $c \neq \epsilon$, then we simply pop off b and push in c. Stay tuned... ☺

3. If $b = \epsilon$, but $c \neq \epsilon$, then PDA makes this move without popping any element, but pushes in c; , e.g., $\delta(q_1, \epsilon, \epsilon) = (q_2, \$)$.

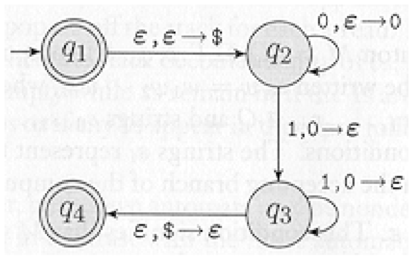
4. if $b \neq \epsilon$ but $c = \epsilon$, then PDA pops off b, but doesn't need to push in any new symbol into the stack; , e.g., $\delta(q_2, 1, 0) = (q_3, \epsilon)$.

5. if $a = b = c = \epsilon$, i.e., $(q, \epsilon, \epsilon) \mapsto (q', \epsilon)$, it is a "free ride" from q to q'. Stay tuned... ☺

State Diagram

Similar to the FA case, it is a lot easier to use a graphical notation to represent an NPDA, mainly its transition. We again adopt the notation of the *state diagram.*

Below is the state diagram of PDA_1, where, for any transition rule, $(q, a, b) \mapsto (q', c)$, we label an arrow from q to q' with $a, b \mapsto c$.



Question: Is it easier to look at when compared with the tabular function as given on Page 30? ☺

Let's play *it* out... with *JFLAP*.

A little useful trick

In particular, with the transition $(q_1, \epsilon, \epsilon) \mapsto (q_2, \$)$, we move from state q_1 to q_2 by putting in '\$' into the stack.

Notice that the formal definition of NPDA does not specify any mechanism to test if the stack is empty.

We provide PDA_1 with an additional stack symbol, \$, and put it in as an initial symbol. Thus, when this symbol is read again, M_1 knows the stack is empty.

This technique is actually pretty general, and can be used in other cases as well.

Notice that *JFLAP* always *implicitly* puts in 'Z' first. Thus PDA_1 will have a 'Z' below the '\$' inside the stack to begin with. ☺

Another NPDA example

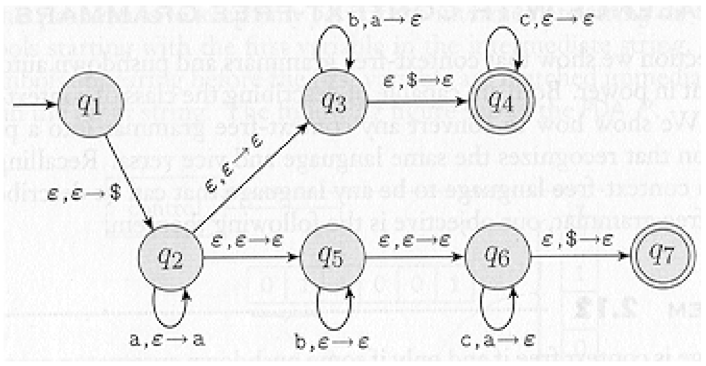
How does an NPDA accepts the following language?

$$\{a^ib^jc^k|i,j,k \geq 0 \text{ and } i=j \text{ or } i=k\}.$$

Informally, the PDA begins by reading and pushing all the a's into the stack, then match them with *either* the b's or the c's. The question is which one?

This is when NPDA comes into the scene, using the "free ride" rule of $\epsilon, \epsilon \rightarrow \epsilon$. (Cf. Item 5 on Page 31)

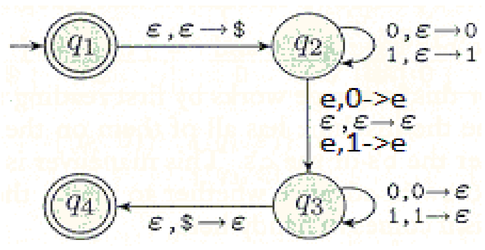
An NPDA can *guess* whether to match with b's or c's, as shown by the following PDA_2:



Yet another example

The following shows a NPDA, PDA_3, that accepts all the palindromes consisting of 0 and 1, i.e., the language $\{ww^R | w \in \{0,1\}^*\}$.

It begins to put the \$ into the stack, then keeps reading and pushing. *At each point,* it guesses that the mid point of the string has been read, thus begins to pop the stack to match the rest of the input. If after all the pops, the stack is empty, this attempt accepts the input, otherwise, it would reject *it.*



Question: Why do we have to add those two rules, $\epsilon, 0 \to \epsilon; \epsilon, 1 \to \epsilon$? What do $0, \epsilon \to \epsilon; 1, \epsilon \to \epsilon$ mean, will they work?

Homework assignment

1. Revisit the *JFLAP* simulator software, and check our its *Pushdown Automata* component.

2. Enter the above three examples on Pages 33, 35, and 36, and play with them.

3. Send in the machines as constructed,and your playing with them.

5. Complete Exercises 2.4(b, e), and 2.10. Use *JFLAP* to verify your results.

Notice that, for Exercise 2.9 and 2.10, we come up with both a grammar that generates, and an NPDA that accepts,

$$\{a^ib^jc^k|i,j,k \geq 0 \text{ and}(i = j \vee j = k)\}.$$

Question: Is CFG the same as NPDA?

CFG $\equiv$ NPDA

Theorem: If a language is context-free, then some NPDA accepts it.

If a language is context-free, there exists a CFG, G, that generates its language. We will construct a NPDA that *accepts* an input iff G *generates* it.

Receiving an input string, the NPDA begins by writing a start variable on its stack. This NPDA will then make an *educated guess* as which substitution rule in G to use for the next derivation step. We only need to find one correct way... .

In general, we just replace a variable at the top of the stack with the RHS of some rule and continue....

The bottom line is that, if the grammar does generate such a string, the machine will accept it.

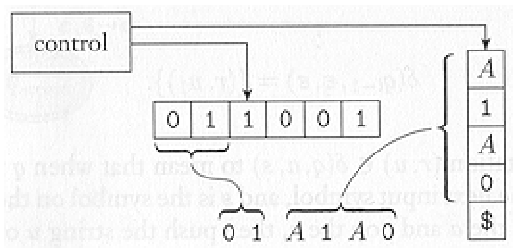
A trick

One obvious place to store all the intermediate strings in the NPDA is its stack. However, as the PDA has to simulate further derivations, it has to look for variables, which are not always at the top of the stack.

Houston, we have a problem. ☹

The solution is to keep on popping off the terminal symbols at the top of the stack with the corresponding symbols in the input string. If they don't match, we simply drop this branch.

Thus, the top element is always a variable which can be further replaced. ☺



Guess this one won't work out. ☹

An informal presentation

1. Place the marker symbol ‘\$’ and the start variable on the stack.

2. Repeat the following steps:
a. If the top of the stack is a variable symbol A, *nondeterministically* select one of the rules for A and substitute A with the right-hand side of the rule.

b. If the top of the stack is a terminal symbol a, read the next symbol from the input and compare it with a. If they do match, repeat. Otherwise, reject this branch.

c. If all branches are rejected, reject the input. ☹

c. If the top of the stack is ‘\$’ when all the symbols in an input are read, enter the accept state to accept this input. ☺

Proof: We construct an NPDA,

$$P = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, \{q_{\text{accept}}\})$$

for a given CFG.

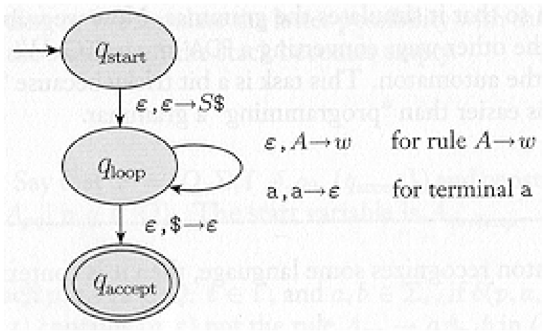
Besides q_{start} and q_{accept}, we also need q_{loop} to simulate this process, and *other states needed to elaborate our simplification* (Page 42)

We begin to define δ, by implementing step 1 using $\delta(q_{\text{start}}, \epsilon, \epsilon) = \{(q_{\text{loop}}, S\$)\}$, where S is the start variable in the CFG. Below are the rules for the main loop:

1. For every rule $A \to w$, add in
$$\delta(q_{\text{loop}}, \epsilon, A) = \{(q_{\text{loop}}, w)\}.$$
2. For every terminal symbol, a, add in
$$\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \epsilon)\}.$$
3. Finally, add in
$$\delta(q_{\text{loop}}, \epsilon, \$) = \{(q_{\text{accept}}, \epsilon)\}.$$

The state diagram

Below is the state diagram of the general NPDA we just developed.



q_{start}
$\varepsilon,\varepsilon\to S\$$
q_{loop}
$\varepsilon, A\to w$ for rule $A\to w$
$a, a\to\varepsilon$ for terminal a
$\varepsilon,\$\to\varepsilon$
q_{accept}

Question: Will this NPDA accept something which is NOT in the language?

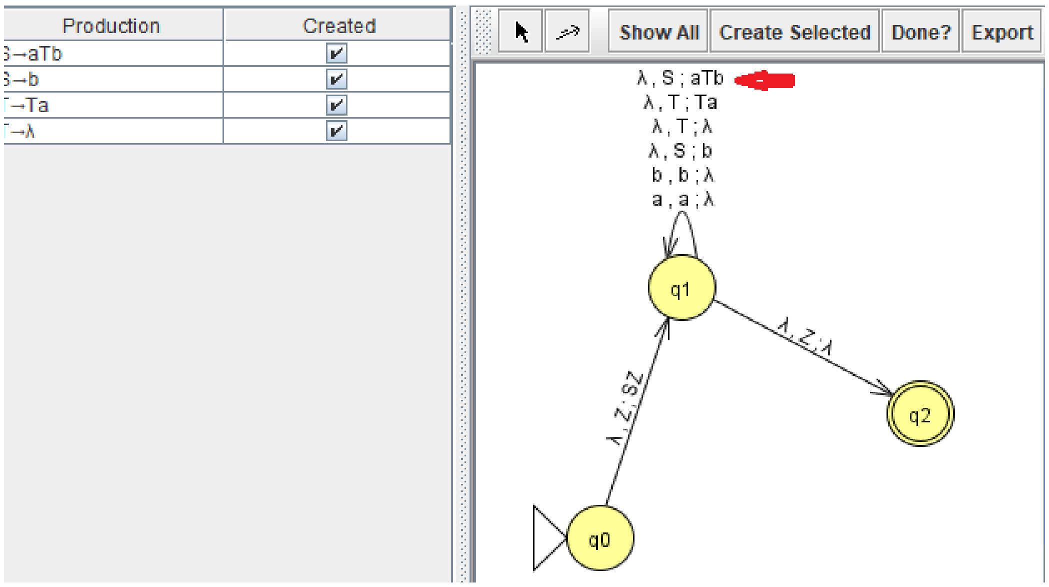
Answer: No. As usual, this NPDA goes to the last rule to accept a string only when it sees ϵ in the input tape, i.e., all the symbols have been swallowed; and it sees '\$' in the stack, i.e., there is nothing left there, when we move into q_{accept}. ☺

Example (I)

Given the following CFG G_6:

$$\begin{aligned} S &\rightarrow aTb|b \\ T &\rightarrow Ta|\epsilon. \end{aligned}$$

Below is the diagram of an equivalent NPDA, that *JFlap* provided. ☺



Production	Created
S→aTb	✓
S→b	✓
T→Ta	✓
T→λ	✓

Question: What does the following rule do?

$$\delta(q_1, \epsilon, S) = (q_1, aTb)$$

Answer: It would pop S, and push in aTb...☹

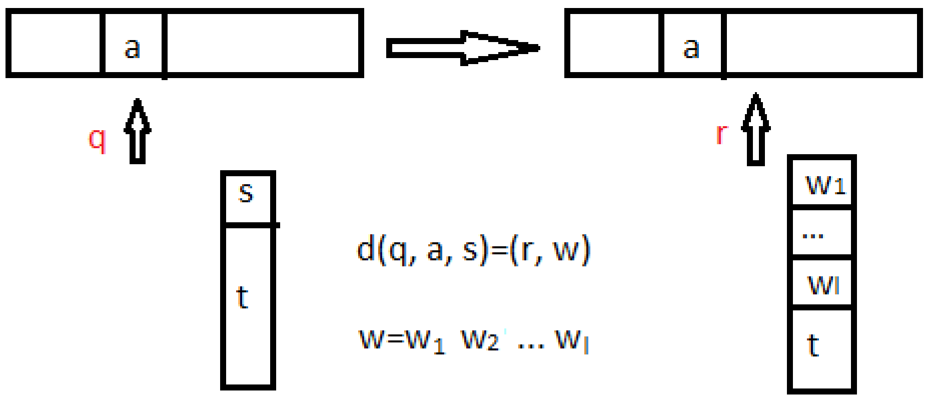
Did Dr. Shen lie again?

The rules collection in the first pass does not fit NPDA, if $|w| > 1$, as we can only push one thing in at a time, by definition as given on Page 29. ☹ We need to fill in some details.

Let q, r be states of an NPDA, let $a \in \Sigma_\epsilon$, $s \in \Gamma_\epsilon$, and let $w = w_1 \cdots w_l \in \Gamma^*$, we use

$$\delta(q, a, s) = (r, w), \text{ e.g., } \delta(q_1, \epsilon, S) = (q_1, aTb)$$

to mean that, in state q, after reading a from the input and seeing s at the top of the stack, the machine will go to state r, after popping off s and *writing w into the stack.*

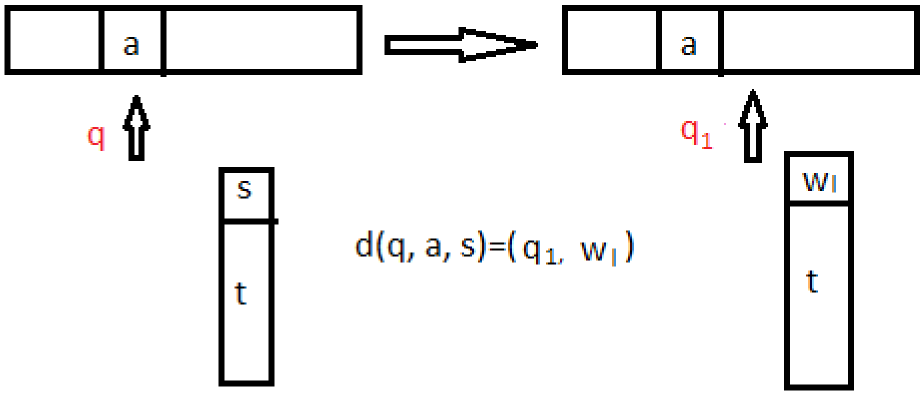


How to elaborate?

Now, we cannot push the whole string $w = w_1 w_2 \cdots w_l$ into the stack in a single shot. ☹

Technically, we will do it one at a time, by introducing new states, $q_1, \ldots q_{l-1}$ and adding the following transition rules.

$$\delta(q, a, s) = \delta(q, a, s) \cup \{(q_1, w_l)\}$$



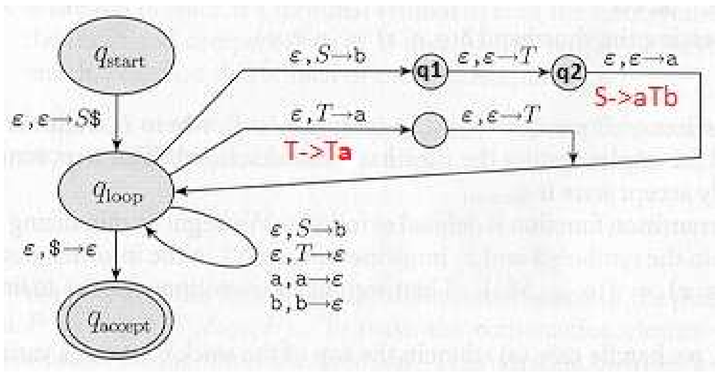
$$\begin{aligned}
\delta(q_1, \epsilon, \epsilon) &= \{(q_2, w_{l-1})\}, \\
\delta(q_2, \epsilon, \epsilon) &= \{(q_3, w_{l-2})\}, \\
&\cdots \\
\delta(q_{l-1}, \epsilon, \epsilon) &= \{q_l, w_1)\} = \{(r, w_1)\}
\end{aligned}$$

Example (II)

Given the following CFG G_6:

$$\begin{array}{rcl} S & \rightarrow & aTb|b \\ T & \rightarrow & Ta|\epsilon. \end{array}$$

Below is the diagram of an equivalent NPDA.



q_{start}
$\varepsilon,\varepsilon\to S\$$
q_{loop}
$\varepsilon,\$\to\varepsilon$
q_{accept}
$\varepsilon, S\to$b
q1
$\varepsilon,\varepsilon\to T$
q2
$\varepsilon,\varepsilon\to$a
S->aTb
$\varepsilon, T\to$a
$\varepsilon,\varepsilon\to T$
T->Ta
$\varepsilon, S\to$b
$\varepsilon, T\to\varepsilon$
a,a$\to\varepsilon$
b,b$\to\varepsilon$

For example, the rule $S \rightarrow aTb$ is replaced with

$$\delta(q_{\text{loop}}, \epsilon, S) = (q_{\text{loop}}, aTb),$$

which is further elaborated as the following:

$$\begin{array}{l} \delta(q_{\text{loop}}, \epsilon, S) = (q_1, b); \\ \delta(q_1, \epsilon, \epsilon) = (q_2, T); \\ \delta(q_2, \epsilon, \epsilon) = (q_{\text{loop}}, a). \end{array}$$

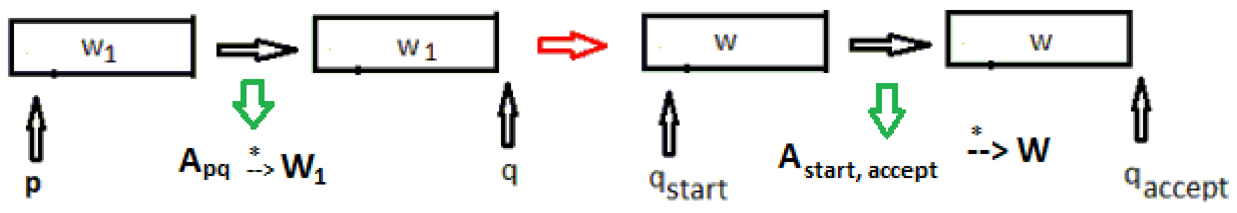
JFlap is not *here* yet, with the current version 7.1..., but the converted NPDA does work. ☺

The other side...

Theorem: If a NPDA accepts a language, it must be a CFL.

Given a NPDA, P, we need to construct a CFG that generates all the strings that P accepts, i.e., *such a string brings P from its start state to its accept state.*

For each pair of states p and q, we use a variable A_{pq} to generate all the strings that bring P from p with an empty stack to q with an empty stack. It thus doesn't touch anything in the stack: $A_{pq} \xrightarrow{*} w_1$.



w_1
w_1
p
$A_{pq} \xrightarrow{*} W_1$
q
w
w
q_{start}
$A_{start, accept} \xrightarrow{*} W$
q_{accept}

In particular, $A_{q_{start},q_{accept}}$, the *start variable,* will generate all the strings, w, that P accepts.

Let's simplify *it* a little...

Now that we have all the variables, including the start variable, the terminals stay the same, the only thing left are the transition rules. ☺

To streamline the process, we modify the given NPDA P such that

1) it has a single accept state,

2) it empties its stack before accepting, and

3) each transition rule either pushes a symbol into a stack, or pops one off, the stack, but does not do both, or none.

Question: Is this too demanding? ☹

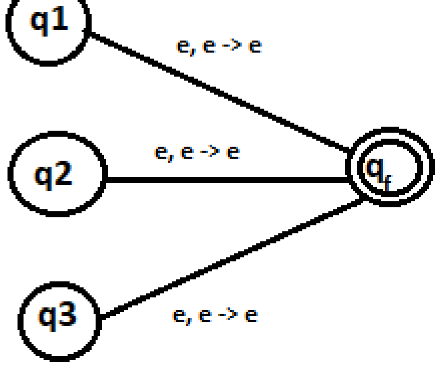
Answer: Of course not. ☺

Question: How come?

Let's patch it up...

1. How to have only one accept state?

Let P be a NPDA with $|F| > 1$. We convert P to another NPDA P' such that $L(P') = L(P)$ and P' has only one accept state as follows: We add in a new state q_f as the only accept state of P', and, for every original accept state, q in P, we insert a transition from q to q_f with a free *Uber* ride $\epsilon, \epsilon \rightarrow \epsilon$.



This revised, but equivalent, machine, P', has just one accept state, q_f, and $L(P) = L(P')$.

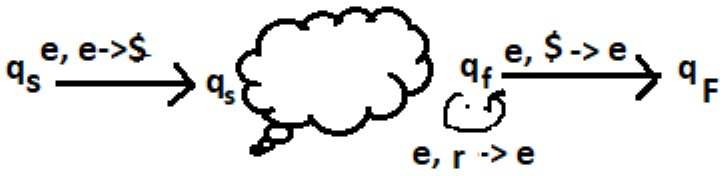
2. How to empty the stack?

Let P' be a NPDA with q_s as its start state, and q_f as its only accept state, and let \$ be the special "end-of-stack" symbol. We come up with a new machine P'' by adding in a new start state, q_S, and a new accept state, q_F, and the following transition rules:

a) We add in $\epsilon, \epsilon, \rightarrow \$$ in between q_S to q_s;

b) For every $\gamma \in \Gamma$, we add in $\epsilon, \gamma \rightarrow \epsilon$ from q_f back to q_f to pop them off, and finally,

c) $\epsilon, \$ \rightarrow \epsilon$ in between q_f to q_F.



Now the stack is clean when we are ready to accept... and $L(P'') = L(P') = L(P)$.

3. How to let all the rules either push in a symbol, or pop off a symbol, but not both?

a) Assume there are rules that do both. Let $a, b \to c$ be a transition rule between p and q. We add in a new state, r, in between p and q and replace the original rule with the following two rules: $a, b \to \epsilon$ between p and r; and $\epsilon, \epsilon \to c$ between r and q.

$$\begin{tikzcd}
p \arrow[r, "\text{a, b -> c}"] & q \arrow[r, Rightarrow] & p \arrow[r, "\text{a, b -> e}"] & r \arrow[r, "\text{e, e -> c}"] & q
\end{tikzcd}$$

b) Assume there are rules that do nothing. Let $a, \epsilon \to \epsilon$ be a transition rule between p and q. Let $b \in \Gamma$, we add in a new state, r, in between p and q, and then replace the original one with the following two rules: $a, \epsilon \to b$ between p and r; and $\epsilon, b \to \epsilon$ between r and q.

$$\begin{tikzcd}
p \arrow[r, "\text{a, e ->e}"] & q \arrow[r, Rightarrow] & p \arrow[r, "\text{a, e -> b}"] & r \arrow[r, "\text{e, b -> e}"] & q
\end{tikzcd}$$

Now this original NPDA P, after the above adjustments, satisfies all the desired properties, with the language unchanged. ☺

A key observation

With this final fitting machine, for any string x, P's first move on x must be a push, as there is nothing to pop at the very first moment; similarly the last move on x must be a pop, as the stack must be emptied after the last moment.

Two things can happen during the computation on x : a.) The symbol popped at the end, e.g., c, is the same symbol P pushed on the stack at the very beginning; b.) These two symbols are different, e.g., it pushes c when entering p, but pops d when exiting q.

We notice that, in Case a, the stack is empty only at the beginning and at the end.

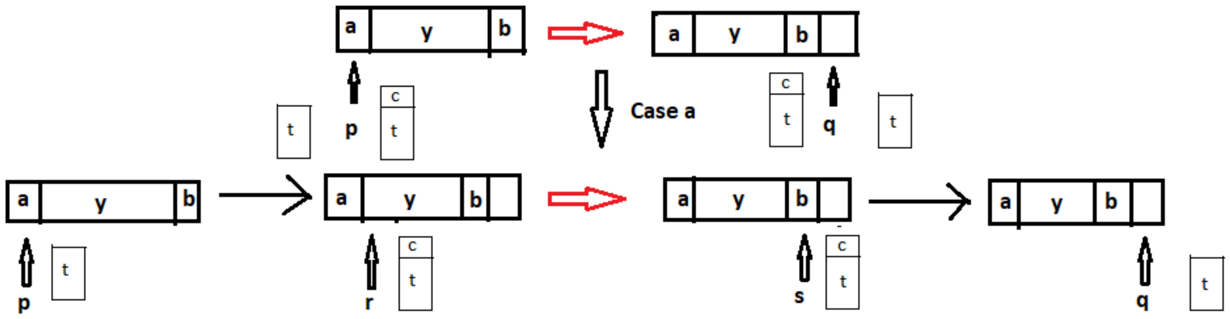
In Case b, the very first symbol in the stack must be popped off at some point, where the stack becomes empty for the first time.

What should we do?

We simulate Case a with the rule

$$A_{pq} \quad \to \quad aA_{rs}b \ (\xrightarrow{*} ayb = x),$$

where a is the first symbol that P reads after it reaches state p, and b is the last symbol it reads before entering q; and r is the state following p and s is the state preceding q.



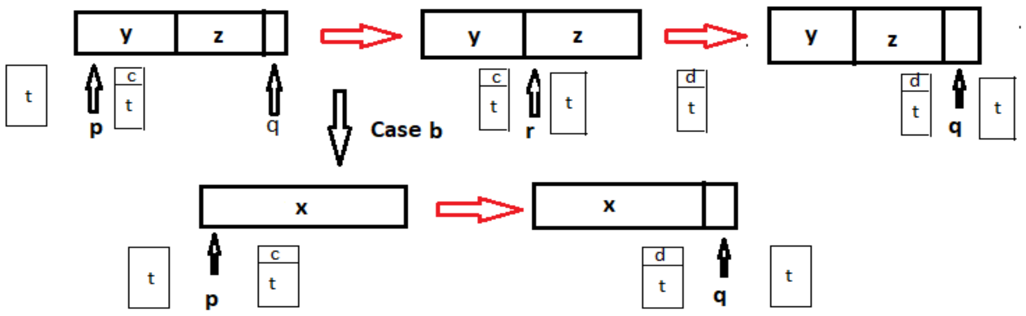
Since y brings P from r to s, w/o touching the stack, *equivalently, with an empty stack,* thus generated by A_{rs}; x $(= ayb)$ would bring it from p to q, w/o touching the stack, *equivalently, with an empty stack*, thus generated by A_{pq}.

How about the other case?

We also simulate Case b with the rule

$$A_{pq} \to A_{pr}A_{rq} \ (\xrightarrow{*} yA_{rq} \xrightarrow{*} yz = x),$$

where r is the state where the stack became empty for the first time. Notice that $r \neq q$ for this case. Otherwise, we are going back to the first case. The symbol d is added in the middle somewhere... .



Let y, z be strings that brings P from p to r, and from r to q, respectively, thus generated by A_{pr} and A_{rq}, respectively; x $(= y \circ z)$ would bring P from p to q, thus generated by A_{pq}.

Again, `t`, the content of the stack, plays no part here. ☺

The grammar

Let $P = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, \{q_{\text{accept}}\})$, and construct $G = (V, \Sigma, A_{q_{\text{start}}, q_{\text{accept}}}, R)$, where $V = \{A_{pq} : p, q \in Q, \}$.

We only need to define R, the collection of transition rules for G: ☺

1. For each $p, q, r, s \in Q$, $c \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if $(r, c) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, c)$, then add in the rule

$$A_{pq} \rightarrow aA_{rs}b.$$

Here c is the stack symbol that we push in at the beginning and pop off at the end in Case a.

2. For each $p, q, r \in Q$, add the rule for Case b.

$$A_{pq} \rightarrow A_{pr}A_{rq}.$$

3. Finally, for every $p \in Q$, add in

$$A_{pp} \rightarrow \epsilon. \Box$$

The last one just says that it would take nothing to bring P from p back to itself.

They are equivalent

We will show that, given any NPDA, P, the just constructed grammar, G, generates exactly the same language as P accepts. ☺

We will actually prove something a bit more general (Page 46): For any two states $p, q \in Q$, A_{pq} generates $x \in \Sigma^*$, iff x brings P from p with empty stack to q with empty stack, or equivalently, *w/o touching the bottom of the stack.*

Once this proved, then in particular we have that $A_{q_{\text{start}}, q_{\text{accept}}}$ generates x iff x brings P from q_{start} to q_{accept}, *w/o touching the bottom of the stack,* i.e., $L(G) = L(P)$.

By the *Background* chapter (Cf. Page 23), we need to do it in two cases: 1. $L(G) \subseteq L(P)$, and 2. $L(P) \subseteq L(G)$.

First part

Claim 1: If A_{pq} generates x, then x can bring P from p with an empty stack to q with an empty stack, equivalently, w/o touching the bottom of the stack.

Proof: We make an induction on the number of steps in the derivation of x. If it takes only one step, then the rule that Γ uses can't have a variable on its right-hand side. The only such rule is in the form of $A_{pp} \to \epsilon$. Clearly, ϵ does take P from p to p.

Assume it holds when the derivation contains only k steps, and let $A_{pq} \xrightarrow{*} x$ in $k+1$ steps.

By construction, the first step of the derivation will be either $A_{pq} \to aA_{rs}b$, or $A_{pq} \to A_{pr}A_{rs}$:

$$A_{pq} \to aA_{rs}b \overset{k}{\Rightarrow} ayb = x \quad \text{or}$$

$$A_{pq} \to A_{pr}A_{rq} \overset{k_1}{\Rightarrow} yA_{rq} \overset{k_2}{\Rightarrow} yz = x,$$

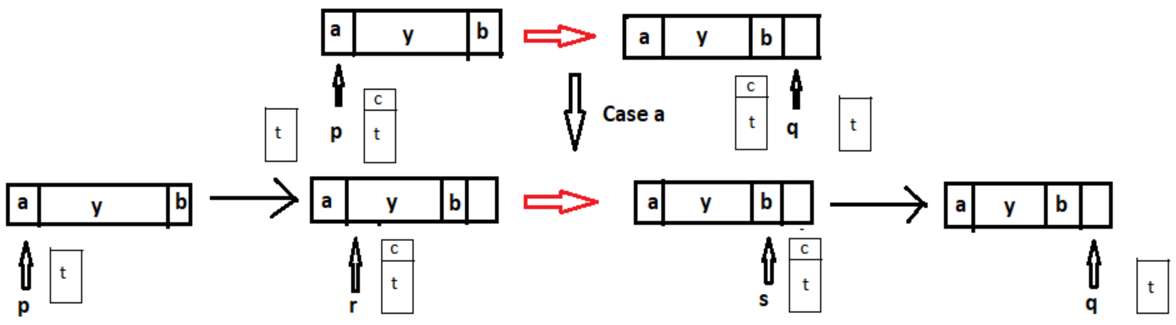
where $k, k_1, k_2 \leq k$.

First case

Consider the portion y, of x $(= ayb)$, that A_{rs} generates, i.e., $A_{rs} \overset{k}{\rightarrow} y$.

By the inductive assumption, y brings P from r with an empty stack to s with an empty stack.

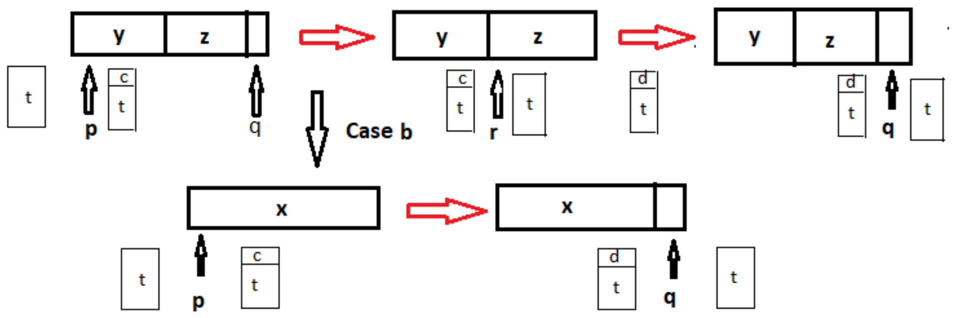
$A_{pq} \rightarrow aA_{rs}b \in R$ since, for some $c \in \Gamma$, $(r, c) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, c)$. Thus, at p, P reads in a, goes to r and pushes c into the stack, and then the string y brings it to s, without changing c. Finally, it reads b, popping off c and goes into q.



Thus, x $(= ayb)$ brings P from p to q without touching the stack during the entire process, or effectively, with empty stacks at both ends.

Second case

In this case, consider the portions y and z that A_{pr} and A_{rq} generates, respectively.



Because P derive them in no more than k steps, by the inductive assumption, y brings P from p to r and z brings P from r to q, without touching the stack during the entire process, or effectively, with empty stacks at both ends.

Hence, x $(= yz)$ brings P from p to q, as well.

This concludes the proof of the first part of the proof, i.e., *for any two states $p, q \in Q$, if A_{pq} generates $x \in \Sigma^*$,then x brings P from p with empty stack to q with empty stack.*

Second part

Claim 2: If x can bring P from p with an empty stack to q with an empty stack, A_{pq} generates x.

Proof: We again prove this result by making an induction on the number of steps in the computation of P that goes from p to q with empty stacks on input x.

If it just takes 0 steps, then P must start and end in the same state, p. Also, the only thing it can read in 0 steps is ϵ. By construction, for any $p \in Q$, $A_{pp} \to \epsilon$ is a rule of G.

Suppose P has a computation where x brings P from p to q with empty stacks in $k+1$ steps, then either the stack is empty only at the beginning and end of this computation, or it becomes empty elsewhere, too.

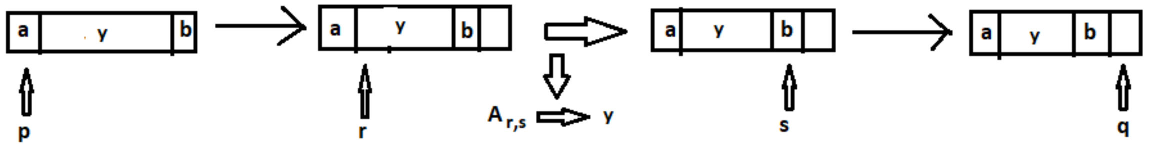
In the first case, the symbol, c, that is pushed into the stack is the same as that is popped at the last move.

Let a be the input symbol that P reads in the first move, and b be the symbol read in the last move, r be the state after the first move, and s be the state before the last move.

Then $(r, c) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, c)$. By construction of the grammar, $A_{pq} \rightarrow aA_{rs}b \in R$.

Let $x = ayb$. As y brings P from r to s without touching the content of the stack, in $k - 1$ steps, by inductive assumption,

$$A_{rs} \xrightarrow{*} y$$

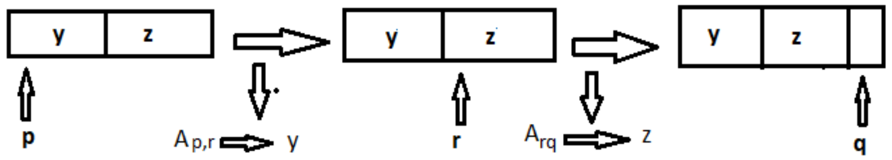


Hence, $A_{pq} \rightarrow aA_{rs}b \xrightarrow{*} ayb = x$.

Thus, A_{pq} generates x in G.

In the second case, let r be the state where the stack became empty for the first time, then the portions of computation from p to r and r to q each contains at most k steps.

Let y and z be the strings that bring P from p to r, and from r to q, respectively.



By the inductive assumption,

$$A_{pr} \overset{*}{\Rightarrow} y \text{ and, } A_{rq} \overset{*}{\Rightarrow} z.$$

Therefore, $A_{pq} \rightarrow A_{pr}A_{rq} \overset{*}{\Rightarrow} yA_{rq} \Rightarrow yz = x.$

Hence, A_{pq} also generates x.

This completes the proof of the claim, i.e., *for any two states $p, q \in Q$, if a string x ($\in \Sigma^*$) brings P from p with empty stack to q with empty stack, then A_{pq} generates x in G.*

Hence, CFG $\equiv$ NPDA.

Programming languages

As we saw on Pages 7 and 8, RL can be used to recognize a particular pattern such as an identifier; and CFL, as we just saw, can be used to specify a language and guide us through the process of parsing a program in that language. Most of the *C*-like programs can be specified by a Context-free grammar.

Those aspects of programming languages that can be modeled by Context-free grammars are called *syntax*.

Efficient algorithms, in $\Theta(n)$, have been worked out to parse programs written in certain restricted context-free grammar, such as LL and LR grammar. (*JFlap* does implement both.)

They play a significant role in programming as we know *it*. (Cf. Course page)

Syntax vs. semantics

A syntactically correct program does not need to be an acceptable program. ☹ For example, the following segment does not make sense.

```
char a;
a=3.2;
```

Context-free grammar cannot be used to enforce the requirement that such “type crash” not be permitted.

Such a rule enforcement is part of the *semantics* of a programming language, and it has something to do with the *meaning* of a particular construct.

Various efforts have been made, and some are rather successful, but none is as universally accepted as context-free grammar has been for syntax.

What does a compiler do?

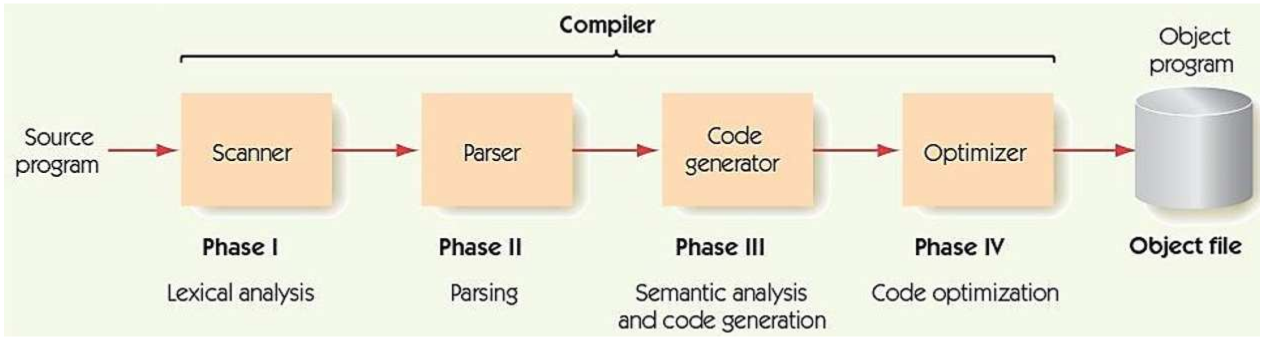
It essentially goes through four steps.

1. *Lexical analysis:* The compiler groups individual characters in a program and groups them into *tokens.*

2. *Parsing:* It goes through a sequence of tokens to see if it forms a syntactically correct program in a specific program language.

3. *Semantic analysis and code generation:* It analyzes the meaning of a program and generates the proper code.

4. *Code optimization:* It tries to make the just generated code more efficient.



An example

Given the following assignment statement,

```
x = x + y + z;
```

using the regular expression as shown on Page 84 of the Regular language notes, the lexical analysis phase can produce the following by categorizing all the tokens:

Token Type	*Classification*
x	1
=	3
x	1
+	4
y	1
+	4
z	1
;	6

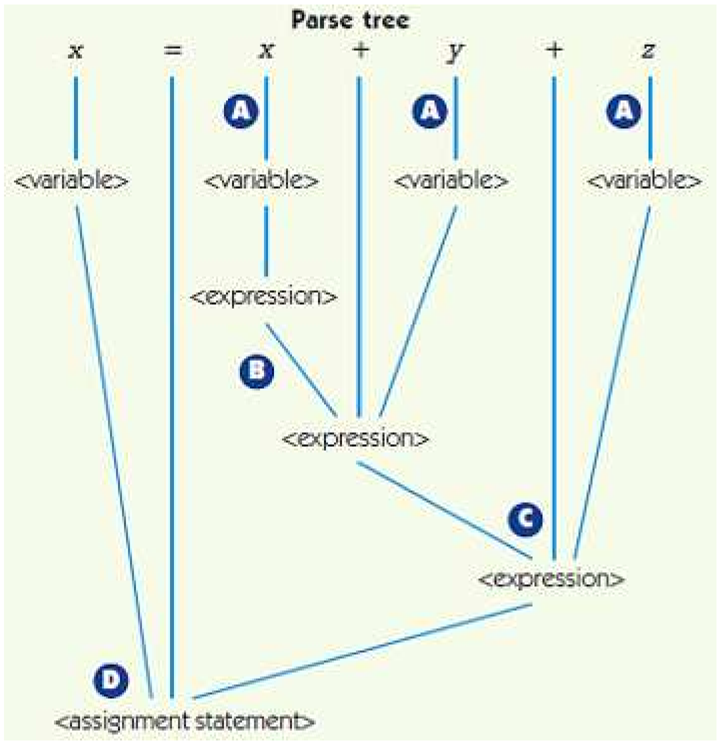
Here category 1 stands for a variable. In general, an *identifier* could contain multiple letters such as “delta” (Cf. Pages 7 and 8).

Question: All the variables fit. Then what? ☺

Parsing

The compiler then generates a parse tree, using the following grammar, which is context-free, but not regular:

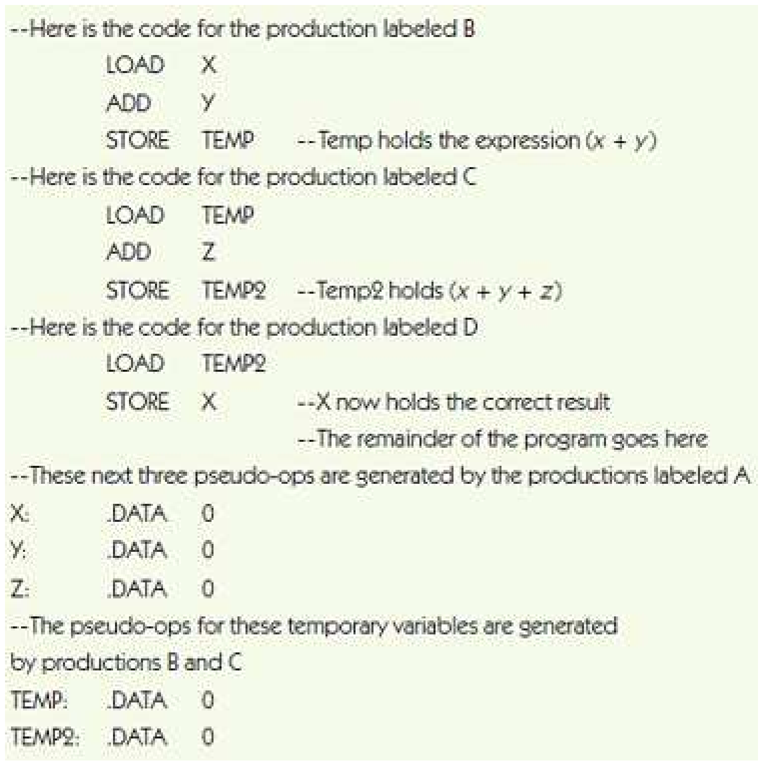
$$\begin{aligned}\text{assignmentStatement} &\Rightarrow \text{variable} = \text{exp}\\ \text{exp} &\Rightarrow \text{variable}|\text{exp} + \text{variable}\\ \text{variable} &\Rightarrow \text{x}|\text{y}|\text{z}\end{aligned}$$



Question: It is grammatically correct. Notice it is left associative, as G_4 does (Cf. Page 11)

Now the code...

With the above tree, the compiler can then generate the following code:



```
--Here is the code for the production labeled B
            LOAD    X
            ADD     Y
            STORE   TEMP    --Temp holds the expression (x + y)
--Here is the code for the production labeled C
            LOAD    TEMP
            ADD     Z
            STORE   TEMP2   --Temp2 holds (x + y + z)
--Here is the code for the production labeled D
            LOAD    TEMP2
            STORE   X       --X now holds the correct result
                            --The remainder of the program goes here
--These next three pseudo-ops are generated by the productions labeled A
X:          .DATA   0
Y:          .DATA   0
Z:          .DATA   0
--The pseudo-ops for these temporary variables are generated
by productions B and C
TEMP:       .DATA   0
TEMP2:      .DATA   0
```

Question: Is it the best that we could get?

Answer: Probably not, with so many loads ☹

The optimized code

It turns out that we can further improve the above code as follows, with a lot less loads:

```
.begin
   LOAD   X
   ADD    Y
   ADD    Z
   STORE  X
   HALT
X:  .DATA   0
Y:  .DATA   0
Z:  .DATA   0
.end
```

Without the help of such languages, it is quite likely that we won't be able to do what we have been doing these days. ☺

A course on *Compiler* covers much much more details in this area.

Question: Are all languages context free?

Every regular language is CFL

By definition, a regular language is accepted by an FA, which is a NPDA, without a stack, or its stack is never used. Moreover, we just proved that every language that is accepted by a NPDA is a context-free language.

Thus, any regular language is trivially a context-free language, which explains why $RG \subset CFG$ on Page 89. ☺

On Page 88 of the *FA* notes, it is shown that a language is regular if it is described by a regular expression based on regular operators (Pages 24 of *FA* note). Thus, yet another way of showing a regular language is CFL is to show all the regular operators can be specified through a CF grammar.

Homework: Exercises 2.15, 2.16, and 2.17.

Non-context-free languages

Question: Why did we write $CFG \subset CSG$ but not $CFG \subseteq CSG$?

We now present another pumping lemma which can be used to show that certain languages are not context-free. Thus, $CFG \subseteq CSG$ but $CSG \not\subseteq CFG$, i.e., $CFG \subset CSG$ (Cf. Page 89).

Again the idea is that if A is context free, it must satisfy certain properties. Thus, if a language fails to satisfy any of these "pumping" properties, it cannot be context-free. ☺

Pumping Lemma for CFL: If A is a context-free language, then there is a number l, *the critical length,* where, if s is any string in A of length at least l, then s may be divided into five pieces $s = uvxyz$ satisfying the following three properties:

1. For each $j \geq 0, uv^ixy^iz \in A$,
2. $|vy| > 0$
3. $|vxy| \leq l.$

The basic idea

If A is context free, then there is a CFG, G $(= (V, T, \Gamma, S))$, that generates it. We want to show that any "sufficiently long" string $s, |s| \geq l$, in A can be pumped, either up or down, and remains in A.

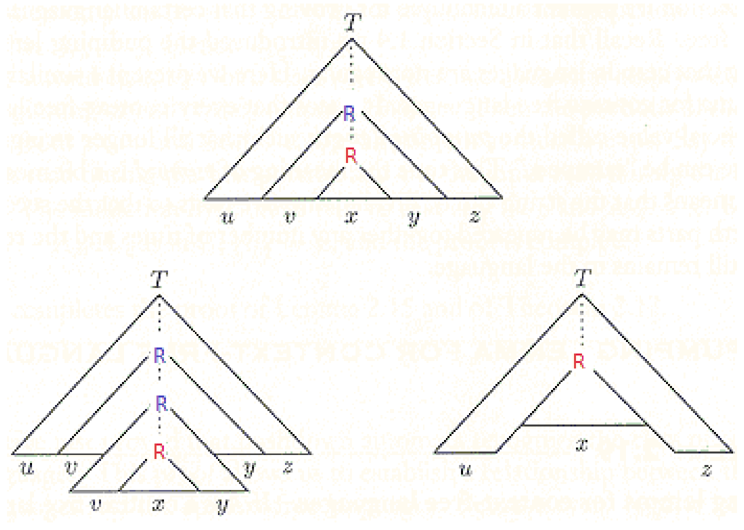
Let s be such a "sufficiently long" string string in A. Because $s \in A$, there is a parse tree for it, based on G, with a long path.

The critical observation is that, when the path is long enough, namely, when the number of symbols along that path is strictly more than $|V|$, the number of variables, some variable, R, must repeat itself, according to the *pigeonhole* principle.

We will get to the details of this critical length later on... .

How to cut the word?

When a variable R repeats itself, the following figure shows that we can replace the parse tree under the first occurrence, *counted from bottom up,* of R with that under the second one, R, and still derive a string in A. This process can be repeated to get pumped-up tree.



We can also do the opposite, and replace the second R with the first R to get the bottom right derivation tree, the pumped-down one.

Proof of the lemma

Let G $(= (V, T, \Gamma, S))$ be a CFG for A, and let b be the maximum number of symbols in the right-hand side of a rule, in Γ, such that $b > 1$.

Thus, in T, any parse tree of G, no node can have more than b children. If we call the start variable at level 0, then the length of the string at level 1 is at most b, strictly less than b^2; the length of a word at level 2 is at most b^2, strictly less than b^3; ....

In particular, if $h(T)$, the height of T, is less than h (A), then the length of the string generated by this parse tree is less than b^h (B).

Since $A \rightarrow B$ iff $\neg B \rightarrow \neg A$, if the length of a string is at least b^h, then the height of its derivation tree must be at least h.

$$h(T) < h \rightarrow |s| < b^h \Leftrightarrow |s| \geq b^h \rightarrow h(T) \geq h.$$

The final kick

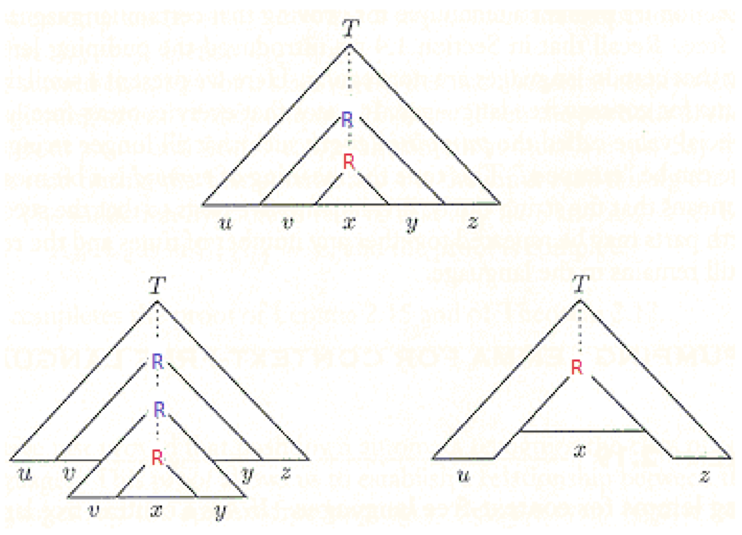
Let V be the number of variables in G, let $|s| \geq l = b^{|V|+2}$, and let τ be *the tree that generates* s *with the smallest number of nodes.* Then, $h(\tau) \geq |V| + 2$.

As τ has a height at least $|V| + 2$, some path in τ has a length at least $|V| + 2$. As the last symbol of this path must be a terminal symbol, this path contains at least $|V| + 1$ variables.

By the pigeonhole principle, some variable, R, must repeat in at least two different locations of τ.

We select such an R that *it repeats among the first* $|V|+1$ *variables,* counted from bottom up.

We now divide s into $uvxyz$ as follows:



Each occurrence of R generates a part of s.

The upper occurrence of R generates vxy, which is strictly longer than x, the part generated by the lower occurrence of R. If this is not true, i.e., both v and y are empty, then we can have another tree, with the sub-tree rooted at R being replaced with that rooted at R that still generates s, with at least one less variable. This contradicts our assumption that τ contains minimum number of nodes.

Thus, $|vy| > 0$.

The other two pieces

Now, we may replace, repeatedly, the parse tree generated by R with that generated by R and get parse trees to generate $uv^ixy^iz, i \geq 1$, and other way around, i.e., replacing the tree root at the upper R with the one rooted at R to get a tree that generates uxz.

Therefore, all the strings in the form of uv^ixy^iz belong to $A, i \geq 0$.

Finally, as both occurrences of R are located within $|V| + 1$ variables in the path, thus, the height of the subtree rooted at the upper R is at most $|V| + 2$, which means that the length of the string it generates, i.e., vxy is at most $l(= b^{|V|+2})$. $\square$

Some examples

Example 1: $B = \{a^n b^n c^n | n \geq 0\}$ is not a CFL.

Assume that B is context free. By definition, there is a context free grammar $G = (V, T, \Gamma, S)$ such that $L(G) = B$.

Let l be the critical length as given in the Lemma, i.e., $l = b^{|V|+2}$, where b is the length of the longest RHS of any rule in R. We choose $s = a^l b^l c^l$. Since $s \in B$ and $|s| = 3l \geq l$, by the pumping lemma, $s = uvxyz$, such that

1. For each $j \geq 0, uv^i xy^i z \in B$,
2. $|vy| > 0$
3. $|vxy| \leq l$.

If, e.g., v contains a, b and c, v must contain all the b's, and at least one a and b, thus $|vxy| \geq |v| \geq l + 2 > l$, contradicting the fact that $|vxy| \leq l$.

Hence both v and y contain at most two different kinds of symbols.

If, e.g., v contains two kinds of symbols. Without loss of generality, v contains both a and $b, v = a^p b^q$ Then, uv^2xy^2z contains a mixed segment of $a^p b^q a^p b^q$, thus can't be in B. This contradicts the fact that $uv^2xy^2z \in B$.

Thus, both v and y contain only one kind of symbols. However, in this latter case, $uxz \notin B$, either, since $|vy| > 0$, v and y can't be both empty. Thus, in uxz, the number of the respective symbols won't match, as the one that is not contained by v and y will be more than those contained. This contracts the fact that $uxz \in B$.

Therefore, by the Pumping lemma for CFL, B cannot be a context-free language. $\square$

We will be able to *see* it when playing with *JFLAP*... . ☺

Homework: Exercise 2.13 (a,b).

Example 2: $D = \{ww | w \in \{0, 1\}^*\}$ is not a CFL.

The critical point of applying pumping lemma is to find an appropriate s, which is not always easy. We might use $s = 0^l10^l1$. Clearly s satisfies the conditions, thus, s can be split into $uvxyz$. But, it does not lead to the needed contradiction. For example, if x happens to be 1 and $|v| = |y|$, then $uv^ixy^iz \in D, i \geq 0$. ☹

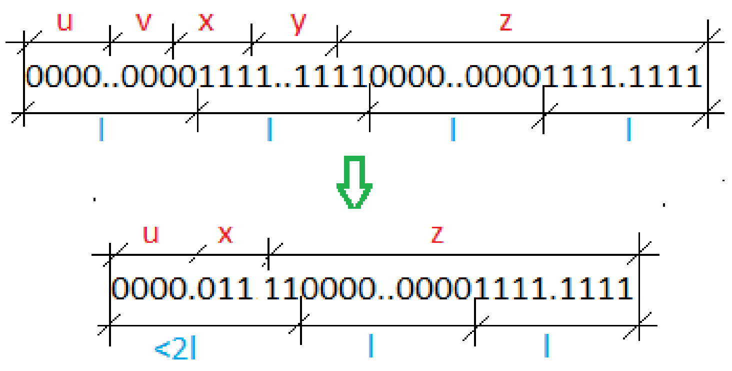
The key observation is that any CFL has to be accepted by an NPDA, which can only use a stack. Thus, if you can find a string in the language which seems quite difficult for a stack to accept, this string might be a good candidate. ☺

For example, the string we used before $s = 0^l10^l1$ is not that difficult to be accepted with a stack (?), but $s_1 = 0^l1^l0^l1^l$ is a different story.

Question: Why is that?

Let $s_1 = uvxyz$, where $|vy| > 0$, i.e., it can't be the case that both v and y are empty.

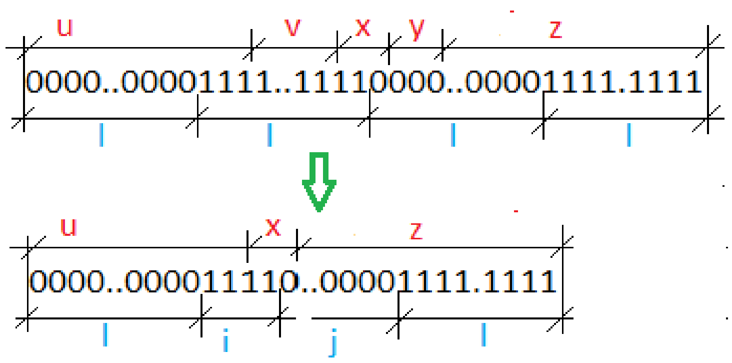
If vxy occurs in the first half of s_1, then in uxz, i.e., when we remove both v and y from s_1, at least one symbol will be removed from the first 0^l1^l, while the second 0^l1^l will stay the same. Thus, uxy can't be a sequence of two identical copies, i.e., $uxy \notin D$.



Similarly, vxy can't occur in the second half of s_1.

Question: Where could it be?

Thus, the vxy part can only occur in the middle of s_1. Then, the string uxz, must be in the form of $0^l1^i0^j1^l$. As v and y cannot be both empty, i and j cannot be both l. Hence, $uxz \notin D$. ☹ □



Again, we will be able to *see* it clearly, when playing with *JFLAP*... . ☺

Since every regular language must be context free, D $(= \{ww|w \in \{0,1\}^*\})$ can't be regular.

Notice that $\{www|w \in \{0,1\}^*\}$ is not regular (Cf Homework 1.29(b)), but $\Sigma^* = \{w|w \in \{0,1\}^*\}$ is (Cf. Page 84 in the RL/FA notes).

Closure properties of CFL

It is easy to see that CFL is closed under union.

Question: How about intersection and complementation?

Let G_7 be defined as

$$S_1 \to S_{ab}C_s,\ \ S_{ab} \to aS_{ab}b|\epsilon,\ C_s \to C_sc|\epsilon$$

and G_8 be defined as

$$S_2 \to A_sS_{bc},\ \ A_s \to A_sa|\epsilon,\ \ S_{bc} \to bS_{bc}c|\epsilon$$

Then

$$L_7 = L(G_7) = \{a^nb^nc^m : m, n \geq 0\}$$
$$L_8 = L(G_8) = \{a^mb^nc^n : m, n \geq 0\}.$$

Since $B = L_7 \cap L_8$, CFL is not closed under intersection.

It is also not closed under complementation because of the De Morgan's law.

$$A \cap B = \overline{\overline{A} \cup \overline{B}}.$$

Regular grammar

Definition: A *regular grammar* is a 4-tuple (V, Σ, Γ, S), where
1. V is a finite set called the *variables,*
2. Σ is a finite set, called the *terminals,*
3. Γ is a finite set of *rules,* each of which contains a variable on the left and a string of a variable and terminals on the right. *If a rule does contains a variable, the variable must occur as the last one.*
4. S is the *start variable.*

The identifier grammar on Page 7 is regular.

For any language, L, if there exists a regular grammar, G, $L(G) = L$, we say L is regular. Obvious, any regular language is also context free.

We can show that the two definitions of a language being regular agree with each other.

An example

The language $L = \{w | w \in (abab \cup baba)^*\}$ is certainly regular, according to the "old" definition (?). On the other hand, we also have the following grammar G such that $L(G_9) = L$.

$G_9 = (V, \Sigma, \Gamma, S)$, where $V = \{S, A, B\}$, $\Sigma = \{a, b\}$, and Γ contains the following rules:

$$S \rightarrow bA, S \rightarrow aB,$$
$$A \rightarrow abaS, B \rightarrow babS,$$
$$S \rightarrow \epsilon.$$

For example,

$$S \rightarrow bA \rightarrow babaS \rightarrow babaaB$$

$$\rightarrow babaababS \rightarrow babaabab.$$

Homework: Generate $babaabababab$ using the above grammar *by hand* ☹ and verify with *JFlap*..

Context-sensitive grammar

Definition: A *context-sensitive grammar* is a 4-tuple (V, Σ, Γ, S), where
1. V is a finite set called the *variables,*
2. Σ is a finite set, called the *terminals,*
3. Γ is a finite set of *rules,* each of which contains a variable and a string of variables and terminals, and, for every rule in R,

$$g \rightarrow h, |g| \leq |h|.$$

4. S is the *start variable.*

For any language, L, such that there exists a context-sensitive grammar, G, $L(G) = L$, we say L is context sensitive.

Obvious, any CFL is a context sensitive, since it has only one variable on the left-hand side.

An example

The language $\{a^nb^nc^n|n \geq 0\}$, although not context-free (Pages 77 and 78), is context sensitive, with the following grammar, G_{10}, we can show that $L(G_{10}) = \{a^nb^nc^n|n \geq 0\}$.

G contains the following variables: S, B, and C, the terminal symbols a, b and c, with S as its start variable, and the following rules:

$$\begin{aligned} S \rightarrow aSBC, S \rightarrow aBC, \\ CB \rightarrow BC, bB \rightarrow bb, \\ aB \rightarrow ab, bC \rightarrow bc, \\ cC \rightarrow cc. \end{aligned}$$

For example,

$$S \rightarrow aSBC \rightarrow aaBCBC \rightarrow aabCBC$$

$$\rightarrow aabBCC \rightarrow aabbCC \rightarrow aabbcC \rightarrow aabbcc.$$

Homework: Generate $aaabbbccc$ using the above grammar *by hand* ☹.

What automata?

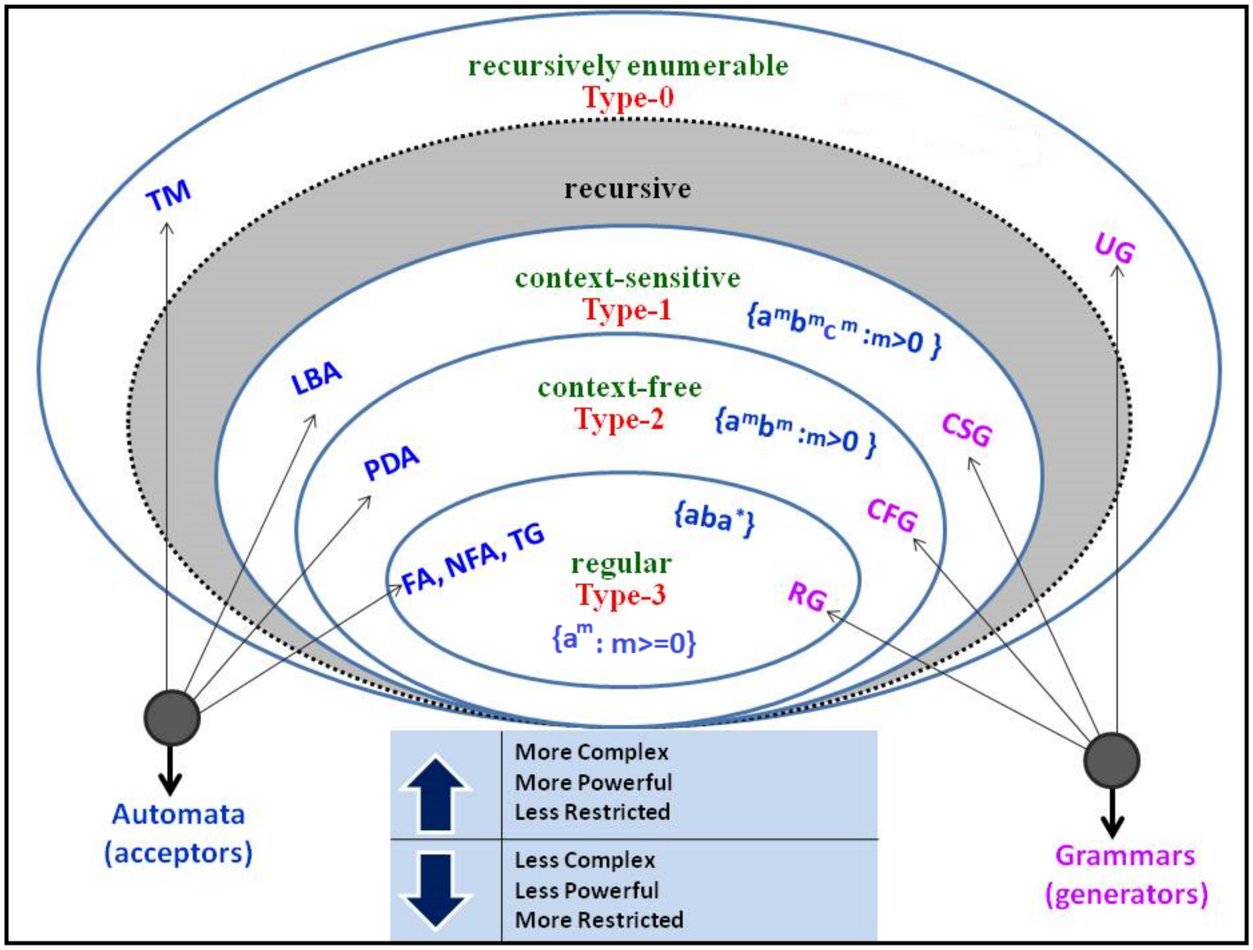
We have so far seen that regular languages correspond to (D)FA; context-free languages correspond to NPDA; a natural question is then what type of automata correspond to context sensitive languages, i.e., those generated by a context sensitive grammar?

A short answer is that context-sensitive languages correspond to LBA (*Linear Bounded Automaton*), a special kind of *Turing Machine* in the sense that, while it contains with an infinite tape, only a finite contiguous portion whose length is *a linear function* of the length of the initial input can be accessed by its read/write head.

Finally, TM corresponds to the set of *recursively enumerable* languages.

A summary...

...with the following:



A *recursive* language is also called *decidable.*

We will have a look at TM in the next chapter.