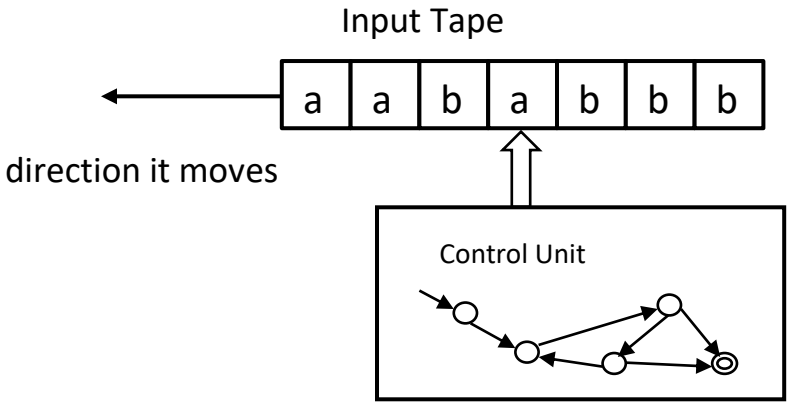
Push-Down Automata (PDA)

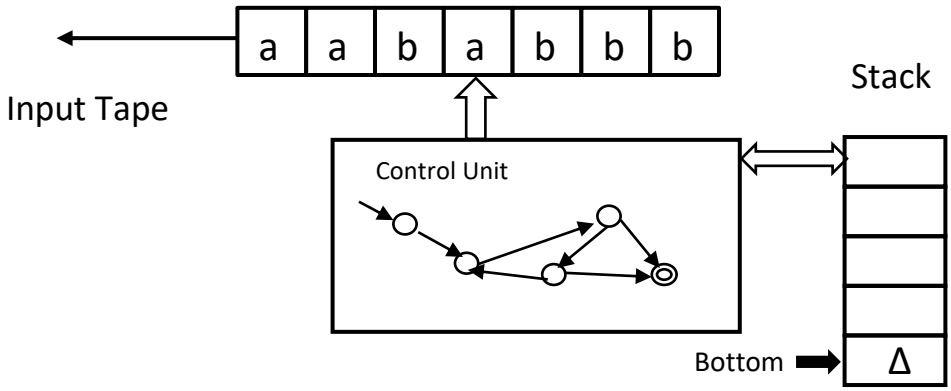
We have studied CFG which, as we have seen, can generate non-regular languages. Hence, some of the languages generated by such grammars cannot be recognized by FSA's (do you remember why?...). FSA are weak because the only mean they have to "remember" anything is via their states which are, by definition, in finite number. To address this problem we may consider to add to an FSA some infinite "memory." The simplest structure for such infinite memory is the stack.

In addition to the stack we will add an "input tape" that the machine will read. To relate this to what we have already studied this is what an FSA showing its input tape would look like:



Initially the machine is positioned on the leftmost character of the tape. The tape moves in one direction (towards the left).

This is now what the PDA looks like:



Δ denotes the symbol used to mark the bottom of the stack. **This symbol never appears elsewhere on the stack**. When the stack is completely empty (i.e. the Δ is gone) the machine halts.
Remember that, for FA's, deterministic and non-deterministic machines accepted the same languages. However, as we will see, this is not true with PDA's. The basic PDA is a non-deterministic machine (NPDA), the deterministic PDA's (DPDA) form a proper subset of the NPDA's. That means that there will be languages accepted by PDA's which are not accepted by any DPDA. From a practical standpoint (i.e., parsing) a deterministic machine is the most desirable.

Definition

A **push-down acceptor (PDA = NPDA)** is defined by seven elements:

- Q - a finite set Q of internal states of the control unit (generally there won't be many of these),
- Σ - an input alphabet (of course finite),
- Γ - a finite set of stack symbols called the stack alphabet (it can overlap with the input alphabet but does not have to and it includes a "bottom-of-stack" symbol),
- δ: - a transition function (described below),
- $q_0 \in Q$ - - an initial state,
- $z \in \Gamma$ - a stack start symbol, (otherwise known as "bottom of stack)
- $F \subseteq Q$ - a set of final states.

δ is a function from Q x (Σ ∪ {λ}) to finite subsets of Q x Γ*. This means that:

$$\delta(q, a, c) = \{ (q', w) \}$$

describes a transition where the machine will be in state q and will:

1. read the next symbol (a), pop the symbol at the top of the stack (c),
2. move to the next input symbol, move to state q', and push a new string onto the stack (w), this string may include c and may be empty.

In other words, each machine is defined by a septuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

Note: as indicated earlier, this **standard** definition of a PDA is that of a **non-deterministic** machine. All context-free languages can be recognized by a NPDA but not all languages can be recognized by a deterministic PDA. Since there must be a stack symbol for any transition to take place, once z is gone the machine cannot move and it stops (or halts).

Now, what do we do?

1. Given a language to recognize, construct a PDA which accepts that language. The "skeleton" method comes in handy again. Think of the machine required to accept the legitimate strings and worry about undefined transitions. Often these will end up being "crashes" which will finish the work quite easily.

   Example: design a PDA to recognize $L = \{a^n b^n \mid n > 0\}$

   $\delta(q_0, a, 0) = \{(q_0, 10)\}$, [0 designates the bottom of the stack]
   $\delta(q_0, a, 1) = \{(q_0, 11)\}$, [1 is the stack symbol for an "a"]
   $\delta(q_0, b, 1) = \{(q_1, \lambda)\}$, [each b pops a "1" from the stack]
   $\delta(q_1, b, 1) = \{(q_1, \lambda)\}$, [q_1 records we moved to b side]
   $\delta(q_1, \lambda, 0) = \{(q_2, \lambda)\}$, [$q_2$is final state when stack is empty]

   Hence the machine is:

   $(Q = \{q_0, q_1, q_2\}, \quad \Sigma = \{a, b\}, \quad \Gamma = \{0, 1\}, \quad z = 0, \quad F = \{q_2\})$

Note that, as we did with NFA, since this machine is non-deterministic we don't require that it be complete either. Some transitions are left undefined and in such cases the machine crashes for the corresponding input (e.g. $\delta(q_1, a, x)$ is undefined and the machine crashes if it is in state q_1 and receives an a.)
When it gets to q_2 the stack is empty and the machine stops. Since q_2 is final the machine accepts.
The method: read the a's and push them on the stack. When you get to the b's pop a matching a for each b you read. At the end of the input the stack should be empty.

Note: the *default* meaning of PDA is non-deterministic. In addition to no transition on some inputs, there may be multiple transitions on others. As indicated above, if a transition is missing the machine "crashes" which is the same as a reject.

In a way similar to the way we defined acceptance for NFA's, we define the language accepted by a PDA as "all the strings for which there is *some* path that leads to an "accept," even though there might be many other paths which do not. For strings not in the language the PDA may end in a "reject" state, crash, or loop endlessly.

If each input string defines a single path through the PDA we will call it a deterministic PDA (DPDA). As we will see later a DPDA need not be complete either. In addition, the presence of λ as the input symbol does not necessarily mean that the machine is non-deterministic (this is due to the fact that a stack symbol plays a role in any transition). As a matter of fact, the machine above is deterministic.

2. Given a PDA, what language does it accept?

   Example:

   $Q = \{q_0, q_1, q_2, q_3\}$,
   $\Sigma = \{a, b\}$,
   $\Gamma = \{0, 1\}$,
   $z = 0$,
   $F = \{q_3\}$

   $\delta(q_0, a, 0) = \{(q_1, 10), (q_2, 0)\}$,
   $\delta(q_0, \lambda, 0) = \{(q_3, \lambda)\}$,
   $\delta(q_1, a, 1) = \{(q_1, 11)\}$,
   $\delta(q_1, b, 1) = \{(q_2, \lambda)\}$,
   $\delta(q_2, b, 1) = \{(q_2, \lambda)\}$,
   $\delta(q_2, \lambda, 0) = \{(q_3, \lambda)\}$,

Definitions

Instantaneous Description (ID)

In order to be able to describe the future behavior of a PDA we need to provide three pieces of information:

the current state of the machine
the portion of the string which has not been read yet
the content of the stack.

Note that the current state and the stack account for the beginning of the input and this is the reason why we don't need to include it.

We define an instantaneous description (ID) as a triplet (q, w, u) where

q is the current state of the control unit
w is the unread part of the input string
u is the stack content.

Move

A move from one ID to another is denoted by $\vdash$ and reflects how the pushdown automaton reads a symbol, changes state, and pushes a string onto the stack.

Assuming that $\delta(q_1, a, b)$ includes (q_2, y) (remember a PDA is a non-deterministic machine), the move will be:

$$(q_1, aw, bx) \vdash (q_2, w, yx)$$

A move involving an arbitrary number of steps (including 0) will be denoted as usual by putting a * above the $\vdash$.

Language Accepted by a PDA

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ be a PDA (of course non-deterministic). The language accepted by M is the set of strings

$$L(M) = \{ w \in \Sigma^* \mid (q_0, w, z) \ {}^*\!\vdash (p, \lambda, u), p \in F, u \in \Gamma^* \}$$

In other words, the PDA must be in a final state at the end of the string. Note that the content of the stack is irrelevant. We will see later that accepting on an empty stack, or accepting on an empty stack while in a final state are two variations which do not give any more, or any less power to the PDAs.

Your turn . . .

Construct PDAs which accept each of the following languages:

1. Palindrome with centermark
   $L = \{ wcw^R \mid w \in \{a, b\}^* \}$

2. Odd palindromes

$$L = \{ w(a + b)w^R \mid w \in \{a, b\}^* \}$$

3. Even palindromes

$$L = \{ ww^R \mid w \in \{a, b\}^+ \}$$

CFG = PDA

Properties

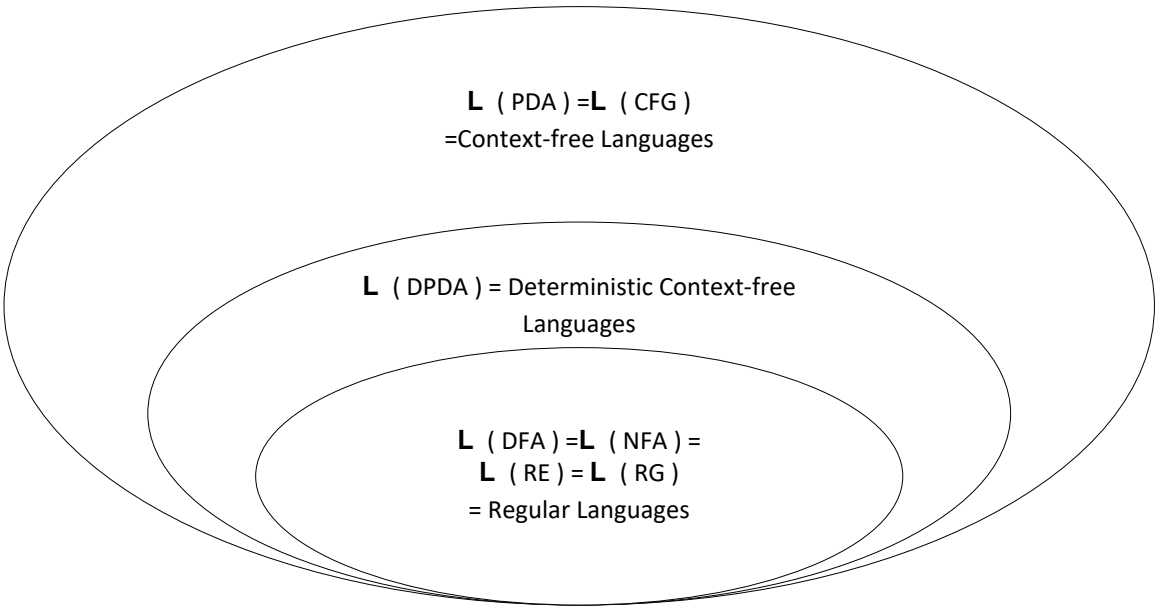
- For every language L generated by a CFG there is an PDA (non-deterministic) which accepts L. We will refer to the languages accepted by PDAs as L (PDA)

- There exists languages accepted by PDA which are not accepted by any DPDA. In other words:

$$L(DPDA) \subset L(PDA)$$

(Contrast this with the FA's where L (NFA) = L (DFA))

Hence, the picture is as follows:



Relation to Parsing

PDA's represent the theoretical foundation behind the front-end of compilers.

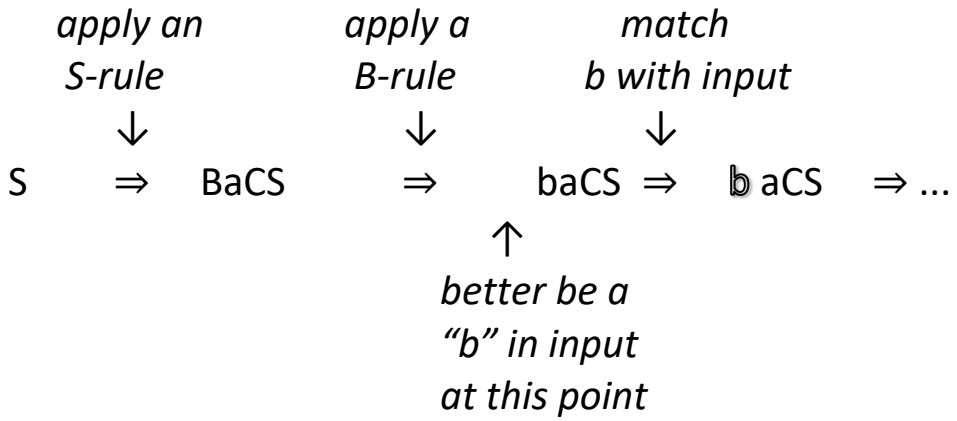
Note that, from what we have just said not all CFG will lead to a deterministic parser. Those which do are called "**deterministic context-free grammars**" and the language they generate are called "**deterministic context-free languages**."

The proof that CFL = L (NPDA) is similar to that of the Kleene theorem for FAs and uses conversion algorithms. It involves two sides:

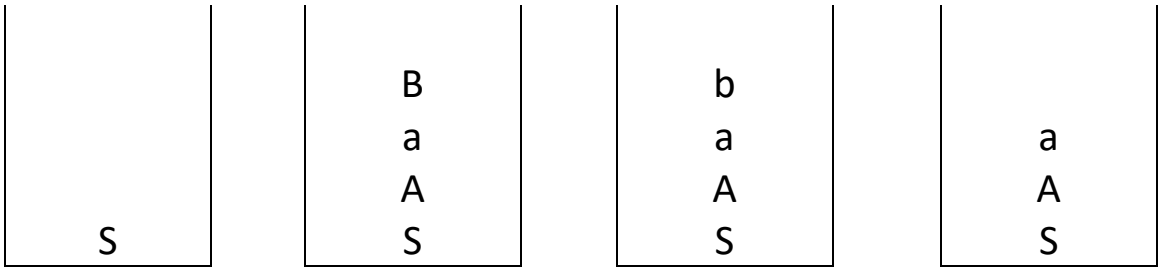
1. NPDA ⇒ CFG long, boring, not that interesting practically. So we won't do it.
2. CFG ⇒ NPDA much more interesting as it involves the creation of a parser from the language specifications.

A "Generic Parser"

Use a "skeleton" for valid strings: it is based on a leftmost derivation. We use the stack to hold the current step of the derivation. For example, let us consider the following derivation and look how parsing would parallel it:



As a first step we would put S on the stack, pop it and replace it by BaAS (S would go first on the stack). Then we would pop B and replace it by b. We would then pop b and, since it is a terminal it should match the current input symbol, if it does the next step would pop a, etc. Failure to match a popped terminal to the current input symbol results in an error (or requires backtracking). Here is a picture of the stack:



The algorithm ...

push S
loop
pop x
if x is an NT, push right side of a rule for that NT
(*note: there could be choices*)
if x is a T, match with next input character, then crash or continue
until No more s on the stack, and whole input matched

which simulates a left-most derivation

A good question: are all languages context-free

The answer is no. There are languages generated by phrase structure grammars which are not accepted by PDA's (or generated by CFG). There is a pumping lemma for CFL but we won't discuss it ☺ . Example of languages which are not context-free:

$L = \{ a^n b^n c^n \mid n \geq 0\}$ (cannot count both b's and c's)

$L = \{ ww \mid w \in \{a, b\}^* \}$ (ww^R is but a PDA cannot handle ww)

$L = \{ a^p \mid p = n! \ \& \ n \geq 0\}$ (requires a complex computation a PDA cannot perform)

$L = \{ a^n b^j \mid n = j^2 \}$ (same thing)

PDA can only perform linear computations on the relationship between number of a's and b's.

Extra Material not required

This material will not be covered in class, some of it we did talk about previously and this may clarify things, the rest is for those interested.

We will see later that this corresponds to what is called top down parsing.

Example:

$S \rightarrow AbB \mid BS$
$A \rightarrow a \mid BB$
$B \rightarrow b \mid \lambda$

Trace acceptance of bb, ab and rejection of ba (you might want to write a leftmost derivation for these strings first.).

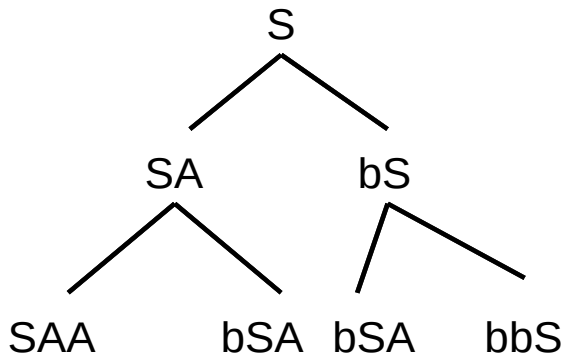
Approaches to Parsing

Let us examine non-deterministic parsing (the same procedures apply to deterministic parsing but we don't have choices). We will assume we scan the input from left to right (since we write left to right we tend to think that way too but it would not have to be so).

Top-down

Start with S, systematically try all the possible rules for the leftmost non-terminal until the string w is found, or we are certain that w does not belong to L(G). This amounts to building, or exploring, the "total language tree."

Example: $S \rightarrow SA \mid bS$
$A \rightarrow abS \mid ba$



This will build the parse tree from the root to the leaves, hence its name. It is relatively easy to recover from making the wrong choice by using backtracking. Because we don't check the string until we get to a leaf this is not very a efficient technique. Moreover, backtracking is very inefficient. Hence practical methods avoid backtracking and use "predictive parsing" techniques where, at each step one can uniquely choose an alternative or declare an error. One of the easy to use technique is **"recursive descent."** This is a technique easily implemented by hand. Other techniques are table-driven and require a program to create the table (think of it as a transition table) and a driver which processes the table. We will look at this later.

Bottom-up

Start with string w, apply rules backward trying to "reduce" to S.

Example: $S \rightarrow Ab$
$A \rightarrow bA \mid a$

Take the string bab, its rightmost derivation (actually it is also a leftmost derivation, but bottom up reconstructs a rightmost derivation) is

$$S \Rightarrow Ab \Rightarrow bAb \Rightarrow bab$$

backward we would reduce a to A, then bA to A, finally Ab to S.

$$b\mathbf{a}b \Leftarrow \mathbf{b}\boldsymbol{A}b \Leftarrow \boldsymbol{A}\mathbf{b} \Leftarrow S$$

we quit when we are at S or we have exhausted all possibilities. You can see that this will build the parse tree from the leaves to the root, hence its name. Recovering from the wrong choice is much harder, specially if you build the parse tree as you go. Because we start from the string this is a more efficient approach.

In this type of parsing we alternate between "shifting" an input symbol onto the stack and "reducing" a string on the top of the stack to a variable which is then pushed on the top of the stack. For this reasons, the parsers are also called **"shift-reduce"** parsers.

Deterministic Parsing

It would be nice if we could always make the "right" choice (especially with bottom-up parsing). This would give us a "deterministic" parser. First, only some languages can be accepted by **deterministic push-down automata**. Such languages are called **deterministic context-free languages**. We can also accomplish deterministic parsing with the concept of "look ahead" that is by looking at a finite number of symbols following the scanned symbol. There are two major categories of such grammars:

For top-down parsing:

LL (k) the first L denotes left to right scanning of the input and the second L refers to leftmost derivation which is what the parsing constructs. The k refers to the length of the "look ahead" needed (k=1 is the only value realistic in practice). Such grammars are suited to recursive descent or to LL(1) parsing.

For bottom-up parsing:

LR (k) the first L denotes left to right scanning of the input and the R refers to rightmost derivation which is what the parsing constructs. The k refers to the length of the "look ahead." The practical techniques used are, in order of increasing power, SLR(1), LALR(1) and LR(1).

Closure Properties of Context-Free Languages

Context-free languages are closed under:

- union
  If L_1 and L_2 are CFL so is $L_1 + L_2$

- product
  If L_1 and L_2 are CFL so is $L_1 \cdot L_2$

- Kleene star
  If L is a CFL so is L*

However, the intersection of two CFL's may or may not be a CFL. We already know that regular languages are also context-free and we know that regular languages are closed under intersection, hence the may. But this is not always true of more complex languages. For example:

$L_1 = \{ a^n b^n c^m \mid n, m \geq 0 \}$ context-free
$L_2 = \{ a^m b^n c^n \mid m, n \geq 0 \}$ context-free

but

$L_3 = L_1 \cap L_2 = \{ a^n b^n c^n \mid n \geq 0 \}$ is not context-free.

Similarly, the complement of a CFL may or may not be a CFL.

One can show that if the complement of two CFL was always a CFL, then the intersection would also always be a CFL (we have just seen an example that this is not so).