What we'll talk about today **CS164 Sections 1 & 4**

- LL1 parsing
- LL1 on a relly nice grammar
- LL1 on a nice grammar
  - First and Follow Sets
- LL1 on an almost nice grammar
  - Left Factoring
- LL1 failing miserably with a not quite nice grammar

LL1 parsing, the idea:

You will start with a single non-terminal, the Start production.
You want to expand your non-terminals by using the production rules. i.e. build the tree top down.
You would like to be able to be able to decide which production to use based on the next token.
Given a non-terminal with several possible productions, how do we tell which one to use next? That's the question at the heart of LL1 parsing.

Really nice grammar:

E -> + T T | (T)
T-> int | * T T | (E)

	+	(	*	int	)
E					
T					

Here are some sample strings you can produce from this grammar.

+ int int
+ * int * int (+ int int) int
(* ((int)) * int int)

In this case, for a given non-terminal, it's easy to tell which rule to use if we know what the next token of the input is. Simply pick the production whose first token matches the next token in the input. We'll generalize this idea for cases where the grammar is not so nice.

Nice grammar:

E->T X
X->+ E | ε
T->(E) | int Y
Y->* T | ε

	+	(	*	int	)
E					
X					
T					
Y					

In this case, we can use the same rule as before. But how do we know what the first token will be for a production like E-> TX.

First and Follow Sets.

In order to solve this problem, you want to find the set of all terminals that could possibly start a string corresponding to this production.

For example, you would like to find all possible terminals that could start a string that can be produced by T X in our above example. Such a set is called the First set of T X, and is denoted as First(T X).

You can define it recursively:

Suppose you are given a sentence of terminals and non-terminals A1 A2 A3…An

Then there are four cases we need to consider:

- If n=1 and A1 is a Terminal, First(A1) = { A1 }

- If n != 1 and First(A1) doesn't contain ε,
  First(A1 A2 A3…An) = First(A1)

- If n=1 and A1 is a non-Terminal, Find all productions of the form
  A1 -> $B_{1,1}$ $B_{2,1}$, … $B_{n1, 1}$
  A1 -> $B_{1,2}$ $B_{2,2}$, … $B_{n2, 2}$
  …
  A1 -> $B_{1,k}$ $B_{2,k}$, … $B_{nk, k}$
  Then, First(A1) = Union(First($B_{1,i}$, …, $B_{ni,i}$))

- If n != 1 and First(A1) contains ε,
  First(A1 A2 A3…An) = Union(First(A1) - ε, First(A2 A3…An))

First sets allow you to know where to use each production. From our above example,

First(TX) = { (, int }
First(+E) = {+}
First((E)) = { (}
First(int Y) = { int }
First(* T) = { * }

So from this we can fill the following entries in the table:

	+	(	*	int	)
E		TX		TX	
X	+E				
T		(E)		int Y	
Y			* T		

But how do we know when to use the ε rules?

Then you need Follow Sets, which are the set of tokens that could potentially come after a non Terminal.

To compute Follow(X),

Start by setting FollowX to empty

Find all rules of the form Y-> … X A1..An

Then, for each of this rule, add First(A1…An) to Follow(X)

And if ε is in First(A1…An), you must also add Follow(Y) to Follow(X).

Note that Follow(Y) may require that you know Follow(X), which leads to a chicken-egg problem. How do you fix it? You have to do it iteratively.

Initialize the Follow sets to {}. Then, update them according to the following equations (which you can derive from the above rules), and repeat until they stop changing.

Follow(E) = {), $} U Follow(X)
Follow(X) = Follow(E)
Follow(T) = First(X) U Follow(E) U Follow(Y)
Follow(Y) = Follow(T)

From this, you will get

Follow(E) = {), $ }
Follow(X) = {) , $ }
Follow(T) = { + , ε,), $}
Follow(Y) = { + , ε,), $}

Now, we can handle the cases we were missing.

	+	(	*	int	)
E	ERROR	TX	ERROR	TX	ERROR
X	+E	ERROR	ERROR	ERROR	ε
T	ERROR	(E)	ERROR	int Y	ERROR
Y	ε	ERROR	* T	ERROR	ε

Almost Nice grammar

In some cases, we may be given grammars for which it seems you can not use this technique.

E->T+E | T
T->int | int * T | (E)

Many times, you can rescue them by using **Left Factoring**

Factor out common prefixes: T, int

E->T X
X->+ E | ε
T->(E) | int Y
Y->* T | ε

L1 failing miserably with a not quite nice grammar

Mostly for your own entertainment, here is an example where left factoring is not enough, and LL1 will fail miserably:

E->T+E | T
T->int | int * T | (E) | int +

After left factoring, we get:

E->T X
X->+ E | ε
T->(E) | int Y | T
Y->* T | ε | +

This is a perfectly legal context free grammar, and it is in fact unambiguous. The problem is that the Follow set of Y contains +, which means that on seeing a + you should use the epsilon rule for Y, but + is also in the First set for +, so this creates a conflict; you can't tell which rule to use by just looking ahead by one.