Compiler Construction

Parsing I

Ran Shaham and Ohad Shacham
School of Computer Science
Tel-Aviv University

Administration

- Forum

  https://forums.cs.tau.ac.il/viewforum.php?f=64

- Project Teams
  - Send me an email if you can't find a team
  - Send me your team if you found one and didn't send an email
  - Check excel file on website

- First PA is at:
  - http://www.cs.tau.ac.il/research/ohad.shacham/wcc08/pa/pa1/pa1.pdf

Programming Assignment 1

- Implement a scanner for IC
- **`class Token`**
  - At least – line, id, value
  - Should extend java_cup.runtime.Symbol
  - Numeric token ids in **`sym.java`**
    - Will be later generated by JavaCup
- **`class Compiler`**
  - Testbed - calls scanner to print list of tokens
- **`class LexicalError`**
  - Caught by Compiler
- **Don't forget to generate scanner and recompile Java sources when you change the spec**
- You need to download and install **both** JFlex and JavaCup

- Assume
  - class identifiers starts with a capital letter
  - Other identifiers starts with a non capital letter

sym.java

```
public class sym {
    public static final int EOF = 0;
    public static final int ID = 1;
     ...
}
```

- Defines symbol constant ids
- Communicate between parser and scanner
- Actual values don’t matter
  - Unique value for each tokes
- Will be generated by cup in PA2

Token class

```
import java_cup.runtime.Symbol;

public class Token extends Symbol {
    public int getId() {...}
    public Object getValue() {...}
    public int getLine() {...}
    ...
}
```

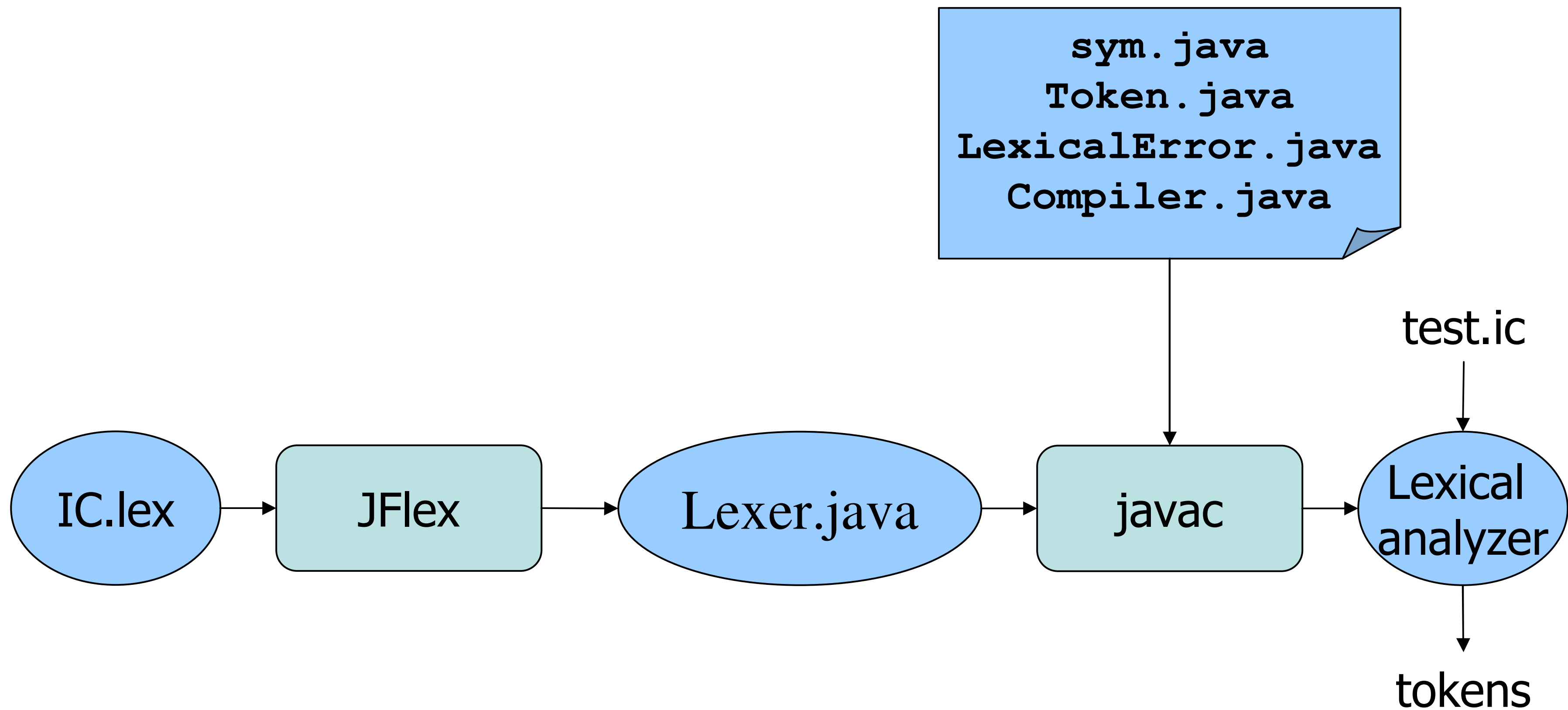
JFlex directives to use

`%cup`	(integrate with cup)
`%line`	(count lines)
`%type Token`	(pass type Token)
`%class Lexer`	(gen. scanner class)

%cup

- %implements java_cup.runtime.Scanner
  - Lex class implements java_cup.runtime.Scanner
- %function next_token
  - Returns the next token
- %type java_cup.runtime.Symbol
  - Return token Class

Structure



Directions

- Download Java
- Download JFlex
- Download JavaCup
- Put JFlex and JavaCup in classpath
- Eclipse
  - Use ant build.xml
  - Import jflex and javacup
- Apache Ant

Directions

- Use skeleton from the website
- Read Assignment
- Use Forum

Tools

- Ant
  - Make environment
  - A build.xml included in the skeleton
  - Download from:
    - http://ant.apache.org
  - Use:
    - ant – to compile
    - ant scanner – to run JFlex

Tools

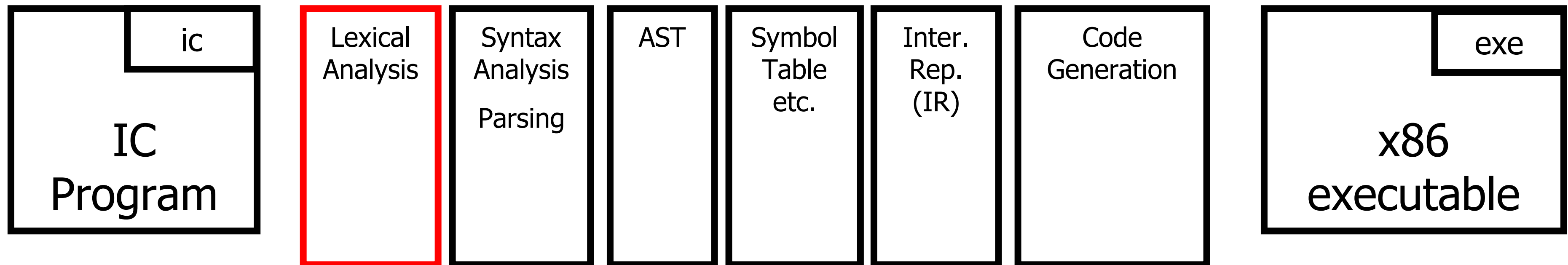
- JFlex
  - Lexical analyzer generator
  - Download from:
    - http://jflex.de/

  - Manual: http://jflex.de/manual.pdf
  - Add $MyJFlex/lib/JFlex.jar to your classpath
  - Use:
    - java JFlex.Main IC.lex
    - ant scanner – for ant users

Tools

- Cup
  - Parser generator
  - Download from:
    http://www2.cs.tum.edu/projects/cup/
  - Manual:
    http://www2.cs.tum.edu/projects/cup/manual.html
  - Put java-cup-11a.jar and java-cup-11a-runtime.jar in your classpath
  - Use:
    - java -jar java-cup-11a.jar <your file.cup>
    - ant libparser – for ant users

IC compiler



Parsing

Input:

- Sequence of Tokens

Output:

- Abstract Syntax Tree

- Decide whether program satisfies syntactic structure

Parsing errors

- Error detection
  - Report the most relevant error message
  - Correct line number
  - Current v.s. expected token
- Error recovery
  - Recover and continue to the next error
  - Heuristics for good recovery to avoid many spurious errors
    - Search for a semi-column and ignore the statement
    - Ignore the next n errors

Parsing

- Context Free Grammars (CFG)

- Captures program structure (hierarchy)
- Employ formal theory results
- Automatically create “efficient” parsers

Grammar:

$S \rightarrow$ if E then S else S

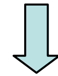
$S \rightarrow$ print E

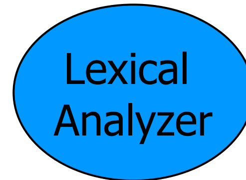
$E \rightarrow$ num

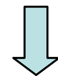
From text to abstract syntax

program text

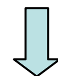
```
5 + (7 * x)
```







token stream



Grammar:

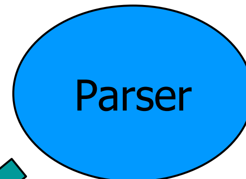
E → **id**

E → **num**

E → E + E

E → E * E

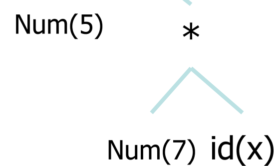
E → (E)



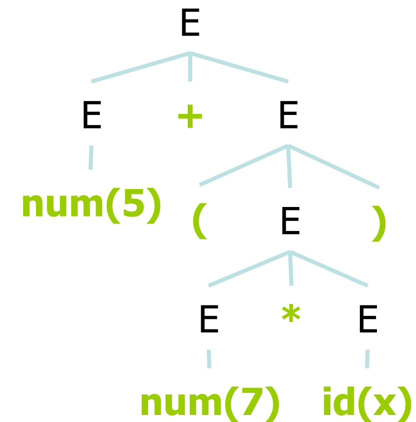
syntax error

valid

Abstract syntax tree



parse tree



From text to abstract syntax

Note: a parse tree describes a run of the parser,
an abstract syntax tree is the result of a successful run

token stream

Grammar:

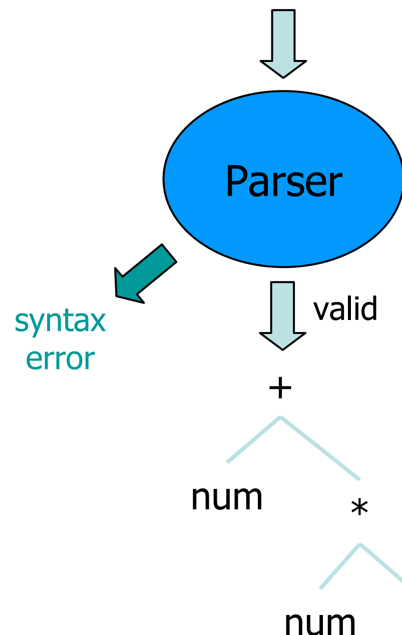
E → **id**

E → **num**

E → E **+** E

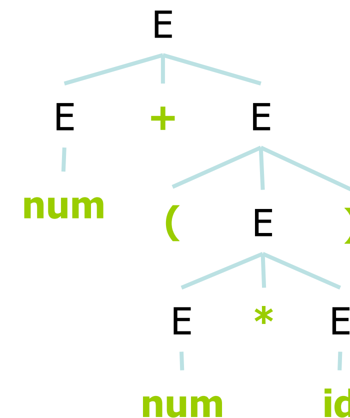
E → E ***** E

E → **(** E **)**



Abstract syntax tree

parse tree



Parsing terminology

Grammar rules (חוקי דקדוק):

E → **id**
E → **num**
E → E **+** E
E → E ***** E
E → **(** E **)**

Symbols (סימנים):

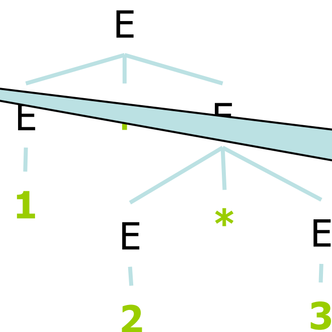
terminals (tokens) **+ * ( ) id num**
non-terminals E

Convention: the non-terminal appearing in the first derivation rule is defined to be the **initial non-terminal**

Derivation (גזירה):

E
E **+** E
1 + E
1 + E ***** E
1 + 2 * E
1 + 2 * 3

Parse tree (עץ גזירה):



Each step in a derivation is called a *production*

Ambiguity

Grammar rules:

E → **id**
E → **num**
E → E **+** E
E → E ***** E
E → **(** E **)**

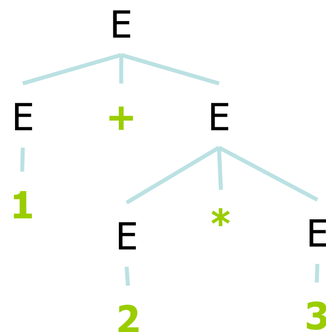
Definition: a grammar is *ambiguous* *(רב-משמעי)* if there exists an input string that has two different derivations

Leftmost derivation

Derivation:

E
E + E
1 + E
1 + E * E
1 + 2 * E
1 + 2 * 3

Parse tree:

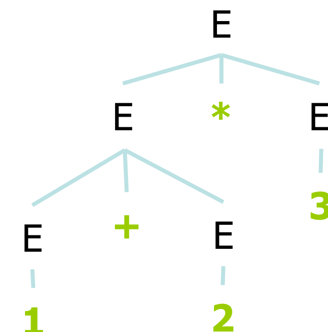


Rightmost derivation

Derivation:

E
E * E
E * 3
E + E * 3
E + 2 * 3
1 + 2 * 3

Parse tree:



Grammar rewriting

Ambiguous grammar:

$E \rightarrow id$
$E \rightarrow num$
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$

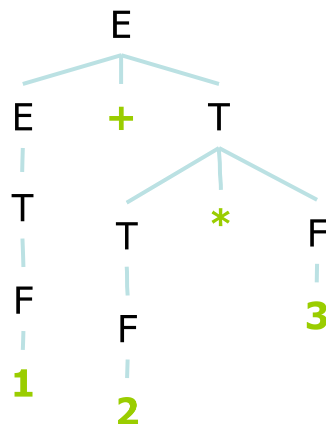
Unambiguous grammar:

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow id$
$F \rightarrow num$
$F \rightarrow (E)$

Derivation:

E
E + T
1 + T
1 + T * F
1 + F * F
1 + 2 * F
1 + 2 * 3

Parse tree:



Note the difference between a language and a grammar:
A grammar represents a language.
A language can be represented by many grammars.

Parsing methods – Top Down

- LL(k)
- "L" – left-to-right scan of input
- "L" – leftmost derivation
- "k" – predict based on "k" look-ahead tokens

- Predict a production for a non-terminal and "k" tokens

Parsing methods – Bottom Up

- LR(0), SLR(1), LR(1), LALR(1)
- “L” – left-to-right scan of input
- “R” – right most derivation

- Decide a production for a RHS and a lookup

Top Down – parsing

E

T + E

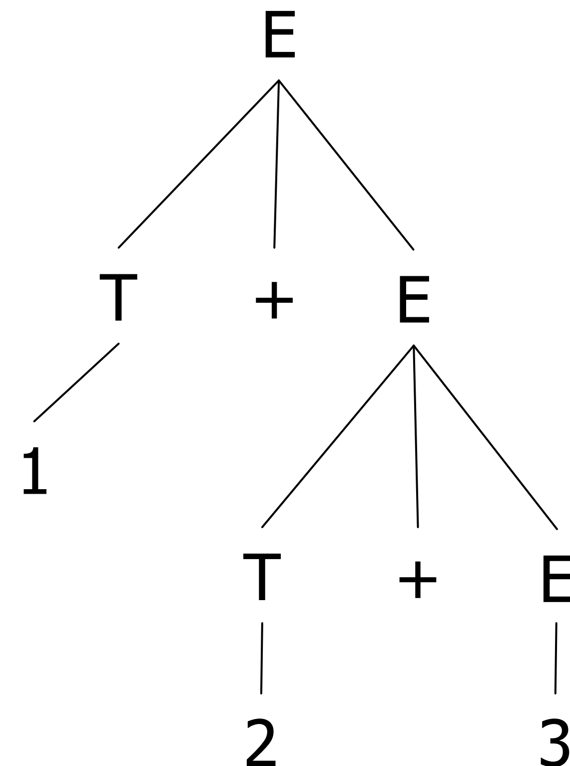
1 + E

1 + T + E

1 + 2 + E

1 + 2 + 3

$E \rightarrow T + E$
$E \rightarrow i$
$T \rightarrow i$



1 + 2 + 3

Top Down – parsing

- Starts with the start symbol
- Tries to transform it to the input
- Also called *predictive parsing*
- LL(1) example

Grammar:

S → if E then S else S
S → begin S L
S → print E
L → end
L → ; S L
E → num

if 5 then print 8 else…

Token	**:** rule
	S
if	**:** S → if E then S else S
	if E then S else S
5	**:** E → num
	if 5 then S else S
print	**:** print E
	if 5 then print E else S

…

Top Down - problems

- Left Recursion
  - $A \rightarrow Aa$
  - $A \rightarrow a$

- Non termination

  A

  Aa

  Aaa

  Aaaa

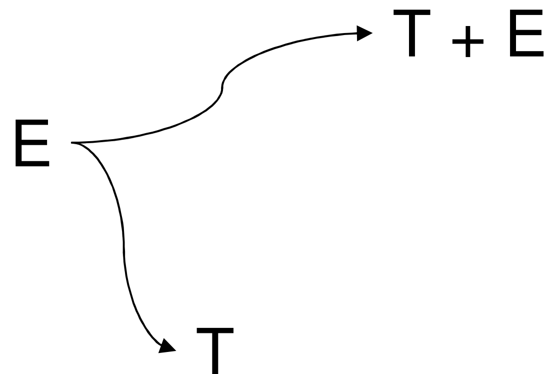
  …

  Aaaaaa…..

Top Down - problems

- Two rules cannot start with same token
- Can be solved by backtracking
- Reduce #backtracks

$E \rightarrow T + E$
$E \rightarrow T$



Top Down – solution

Two ways

- Eliminate left recursion
- Perform left refactoring

Top Down – solution

- Step I: left recursion removal

$E \rightarrow E + T$	$E \rightarrow T + E$
$E \rightarrow T$	
$T \rightarrow T * F$	$T \rightarrow F * T$
$T \rightarrow F$	
$F \rightarrow id$	
$F \rightarrow (E)$	

Top Down – solution

- Step II: left factoring

$E \rightarrow T + E$
$E \rightarrow T$
$T \rightarrow F * T$
$T \rightarrow F$
$F \rightarrow id$
$F \rightarrow (E)$

$E \rightarrow T\ E'$
$E' \rightarrow +\ E$
$E' \rightarrow \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ T$
$T' \rightarrow \varepsilon$
$F \rightarrow id$
$F \rightarrow (E)$

Top Down – left recursion

- Non-terminal with two rules starting with same prefix

Grammar:
S → if E then S else S
S → if E then S

Left-factored grammar:
S → if E then S X
X → ε
X → else S

Bottom Up – parsing

- No problem with left recursion
- Widely used in practice
- LR(0), SLR(1), LR(1), LALR(1)
  - We will focus only on the theory of LR(0)
- JavaCup implements LALR(1)

- Starts with the input
- Attempt to rewrite it to the start symbol

Bottom Up – parsing

1 + (2) + (3)

E + (2) + (3)

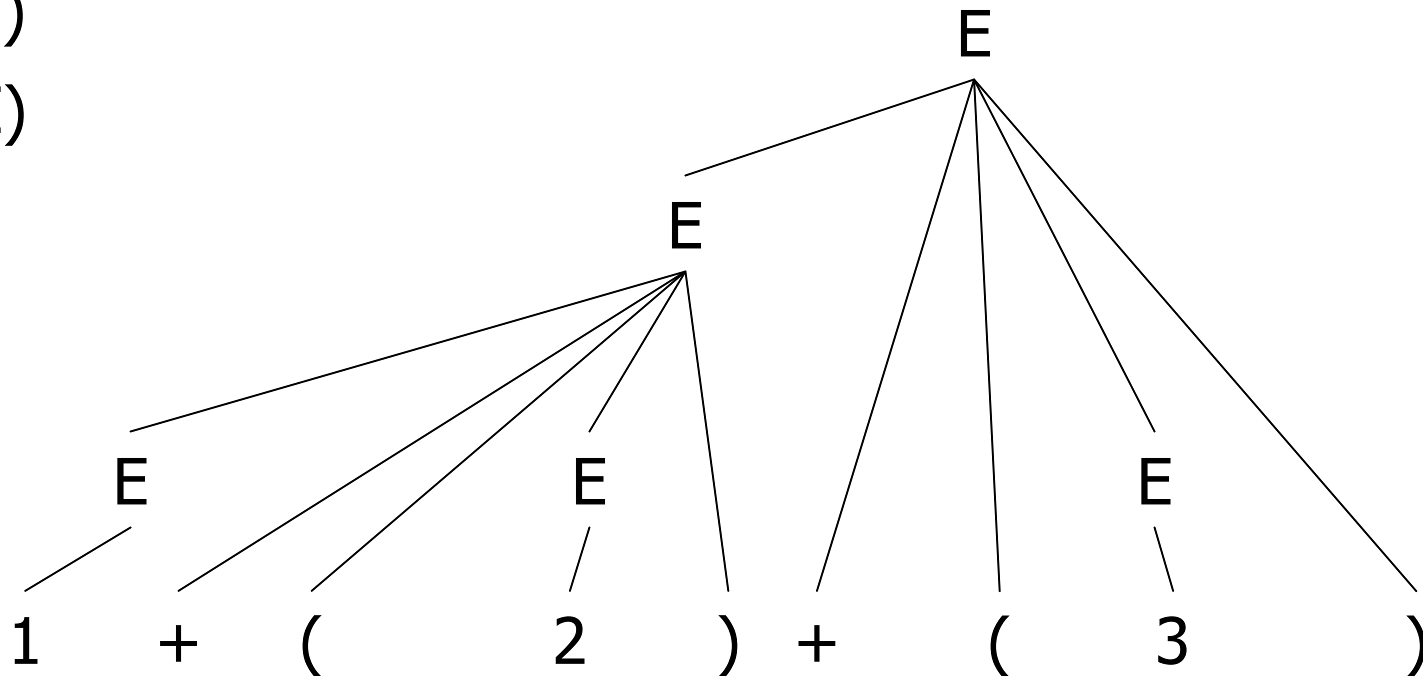
E + (E) + (3)

E + (3)

E + (E)

E

$E \rightarrow E + (E)$
$E \rightarrow i$



Bottom Up - problems

- Ambiguity

  E = E + E

  E = i

  1 + 2 + 3 -> (1 + 2) + 3 ????

  1 + 2 + 3 -> 1 + (2 + 3) ????

Summary

- Do PA1
  - Use forum
- Next week
  - Cup
  - LR(0)