Recursively-defined sets: Grammars

E →	integer
E →	(E + E)

- Grammar: set of rules for generating sentences in a computer language.
- This is a grammar for simple expressions:
  - every integer is an expression.
  - if E_1 and E_2 are expressions, so is $(E_1 + E_2)$.
- Set of legal sentences in this grammar is a recursively-defined set.

E →	integer
E →	(E + E)

Here are some legal expressions:

2

(3 + 34)

((4+23) + 89)

((89 + 23) + (23 + (34+12)))

Here are some illegal expressions:

(3

3 + 4

- Parsing: given a grammar and some text, how do we determine if that text is a legal sentence in the language defined by that grammar?
- For many grammars such the simple expression grammar, we can write efficient programs to answer this question.
- Next slides: parser for our small expression language
  - Caveat: code uses CS211In object for doing input from a file, so it is not an Ur-Java program.
  - However, you should understand the structure of the code to see the parallel between the language definition (recursive set) and the parser (recursive function)

Helper class: CS211In

- Read the on-line code for the CS211In class
- Code lets you
  - open file for input:
    - CS211In f = new CS211In(String-for-file-name)
  - examine what the next thing in file is: f.peekAtKind()
    - Integer?: such as 3, -34, 46
    - Word?: such as x, r45, y78z (variable name in Java)
    - Operator?: such as +, -, *, (,) , etc.
  - read next thing from file:
    - integer: f.getInt()
    - Word: f.getWord()
    - Operator: f.getOp()

- Useful methods in CS211In class:
  - f.check(char c):
    - Example: f.check('*'); //true if next thing in input is *
    - Check if next thing in input is c
      - If so, eat it up and return true
      - Otherwise, return false
  - f.check(String s):
    - Example of its use: f.check("if");
      - Return true if next thing in input is word if

Parser for expression language

```
static boolean expParser(String fileName) {//returns true if file has single expression
        CS211In f  = new CS211In(fileName);
        boolean gotIt = getExp(f);
        if (f.peekAtKind() == f.EOF)//no junk in file after expression
           return  gotIt;
        else //file contains some junk after expression, so return false
           return false;
}
static boolean getExp(CS211In f) {//reads one expression from file
        switch (f.peekAtKind()) {
          case f.INTEGER: //E -  integer
                                {f.getInt();
                                 return true;
                                 }
          case f.OPERATOR: //E -  (E+E)
                                 return f.check('(') && getExp(f) && f.check('+') &&
                                                                  getExp(f ) && f.check(')'));
           default:   return false;
        }
}
```

Note on boolean operators

- Java supports two kinds of boolean operators:
  - E1 & E2:
    - Evaluate both E1 and E2 and compute their conjunction (i.e.,"and")
  - E1 && E2:
    - Evaluate E1. If E1 is false, E2 is not evaluated, and value of expression is false. If E1 is true, E2 is evaluated, and value of expression is the conjunction of the values of E1 and E2.
- In our parser code, we use &&
  - if "f.check('(') returns false, we simply return false without trying to read anything more from input file. This gives a graceful way to handling errors.
  - don't worry about this detail if it seems too abstruse…

Modifying parser to do SaM code generation

- Let us modify the parser so that it generates SaM code to evaluate arithmetic expressions: (eg)

```
2          : PUSHIMM 2
             STOP
(2 + 3)    : PUSHIMM 2
             PUSHIMM 3
             ADD
             STOP
```

Idea

- Recursive method getExp should return a string containing SaM code for expression it has parsed.
- To-level method expParser should tack on a STOP command after code it receives from getExp.
- Method getExp generates code in a recursive way:
  - For integer i, it returns string "PUSHIMM" + i + "\n"
  - For (E1 + E2),
    - recursive calls return code for E1 and E2
      - say these are strings S1 and S2
    - method returns S1 + S2 + "ADD\n"

CodeGen for expression language

```
static String expCodeGen(String fileName) {//returns SaM code for expression in file
        CS211In f = new CS211In(fileName);
        String pgm = getExp(f);
        return pgm + "STOP\n"; //not doing error checking to keep it simple
}
static String getExp(CS211In f) {//no error checking to keep it simple
        switch (f.peekAtKind()) {
          case f.INTEGER: //E = integer
                                return "PUSHIMM" + f.getInt() + "\n";
           case f.OPERATOR: //E= (E+E)
                                        {     f.check('(');
                                              String s1 = getExp(f);
                                              f.check('+');
                                              String s2 = getExp(f);
                                              f.check(')');
                                              return s1 + s2 + "ADD\n";
                                        }
           default:   return "ERROR\n";
        }
}
```

Exercises

- Think about recursive calls made to parse and generate code for simple expressions
  - 2
  - (2 + 3)
  - ((2 + 45) + (34 + -9))
- Can you derive an expression for the total number of calls made to get getExp for parsing an expression?
  - Hint: think inductively
- Can you derive an expression for the maximum number of recursive calls that are active at any time during the parsing of an expression?

Implementing recursive methods

- Ur-Java implementation model already supports recursive methods.
- Key idea:
  - each method invocation gets its own frame
  - frame for method invocation *I* has storage for
    - method parameters
    - method variables
    - returned value (methods invoked by method invocation *I* deposit their return value here)
  - caveat: frame actually contains a few other things as well but we will ignore them for now