Suppose you are given a ‘rule’ such as

\[ a_n = n a_{n-1} \]

and also know that

\[ a_1 = 1 \]

Then we can use this to build a sequence of numbers

\[ 1, 2, 6, 24, 120, 720, \ldots \]

which you recognise as factorials. It’s easy to see this by building up from the bottom

\[ 1, 2 \times 1, 3 \times (2 \times 1), 4 \times (3 \times 2 \times 1), \ldots \]

although using this approach to show that this will really only produce factorials would take an infinite amount of time!
We could prove this in an intuitively rigorous inductive way by

1. Remark that $a_1 = 1 = (1)!$

2. Notice that if we were to assume that $a_n = n!$
then $a_{n+1} = (n+1) \times a_n$
   by our ‘rule’
   $= (n+1) \times (n!)$
   by our assumption
   $= (n+1)!$

This is rather like saying

1. I can put my foot on the first rung of a ladder.

2. IF I’m on any rung of the ladder THEN I can step onto the next rung.

This way of arguing, called induction, is very nice because

a. We don’t have to do infinitely many steps.

b. It’s “jolly obvious” that we’ve covered every case!
Instead of working from the bottom up, we could work from the top down (provided our ‘top’ is only ‘finitely high’) …

If you follow the arrows, you can see that this process first finds the BOTTOM, and then assembles the calculation as it returns to the top. Obviously, if there is no bottom then we will be waiting a jolly long time for any results!! Let’s see this process in Java code.

```java
public int fact ( int n ) {
    if ( n == 1 ) return 1 ;
    else return n * fact ( n-1 ) ;
}
```

Then our `fact ( n )` behaves just like our $a_n$, and it would be invoked by

```java
ans = fact ( 6 ) ;
```

for example - producing the same bottom-hungry routine we saw for $a_n$. 
• In fact, any sequence defined by a recurrence relation can be converted very easily into recursive code. Without making any comments about efficiency (!), recursive code is typically very short.

• As experiments, first you should run this code (previous page) to compute fact(10) and then fact(100) - but you might like to change from int to BigInteger to avoid the sad consequences of int wrap around! After that, try to find the 10th and the 100th term in the following Fibonacci sequence, and then look at the schematic of the recursive calls on the previous page to understand what’s (not) going on (and then fix it)!

\[ a_n = a_{n-1} + a_{n-2} \]
\[ a_1 = 1 \text{, and } a_2 = 1 \]

• It’s worth noting that very similar code can be used to compute \( \binom{n}{r} \), the binomial coefficients (for the intuition behind this look at Pascal’s triangle)

\[ \binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1} \text{ with } \binom{n}{n} = 1 \text{, and } \binom{n}{0} = 1 \]

and powers of a (an example of a ‘divide and conquer’ approach)

\[ a^n = (a^{n/2})^2 \text{ for } n \text{ even} , \quad a^n = a(a^{n/2})^2 \text{ for } n \text{ odd} , \text{ and } \quad a^0 = 1 \]

Note the vastly faster computation for powers when coded this way!
• It’s interesting to see how recursive methods could be implemented within the machine. The key idea is to use a stack to remember parameters and local variables across recursive calls, and for each method invocation to get its own stack frame.

A stack frame contains storage for

- local variables of the method
- parameters of the method
- return info (address and return value)
- any other bookkeeping info

is pushed with each recursive call, and popped when the method returns (leaving any value on top of the stack).

example showing the change of state of the stack of stack frames for computing the value of $2^5$ recursively (as on the previous page)
• The processor has to keep track of all this . . . one approach is to have a frame base register (FBR), so that when a method is invoked, the FBR points to its stack frame. Then when the method invocation returns, the FBR is restored to its value before the invocation (courtesy of part of the return info in the stack frame).

Computational activity only takes place in the topmost (most recently pushed) stack frame.

• A rather nice application of recursion can be found when parsing ‘sentences’ constructed according to formal grammars. These actually underlie the formal structure of programming languages (as well as many other things!).
A Grammar

Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → bunnies
Verb → like
Verb → see

• Our sample grammar has these rules:
  ▪ A Sentence can be a Noun followed by a Verb followed by a Noun
  ▪ A Noun can be ‘boys’ or ‘girls’ or ‘bunnies’
  ▪ A Verb can be ‘like’ or ‘see’

• Grammar: set of rules for generating sentences in a language
• Examples of Sentence:
  ▪ boys see bunnies
  ▪ bunnies like girls
  ▪ ...
• White space between words does not matter
• The words boys, girls, bunnies, like, see are called tokens or terminals
• The words Sentence, Noun, Verb are called nonterminals
• This is a very boring grammar because the set of Sentences is finite (exactly 18 sentences)
A Recursive Grammar

Sentence → Sentence and Sentence
Sentence → Sentence or Sentence
Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → bunnies
Verb → like
Verb → see

• This grammar is more interesting than the last one because the set of Sentences is infinite

• Examples of Sentences in this language:
  ▪ boys like girls
  ▪ boys like girls and girls like bunnies
  ▪ boys like girls and girls like bunnies and girls like bunnies
  ▪ boys like girls and girls like bunnies and girls like bunnies and girls like bunnies
  ▪ ..........

• What makes this set infinite?
  Answer:
  ▪ Recursive definition of Sentence
Sentences with Periods

PunctuatedSentence $\rightarrow$ Sentence .
Sentence $\rightarrow$ Sentence and Sentence
Sentence $\rightarrow$ Sentence or Sentence
Sentence $\rightarrow$ Noun Verb Noun
Noun $\rightarrow$ boys
Noun $\rightarrow$ girls
Noun $\rightarrow$ bunnies
Verb $\rightarrow$ like
Verb $\rightarrow$ see

- Add a new rule that adds a period only at the end of the sentence.
- The tokens here are the 7 words plus the period (.)
- This grammar is ambiguous:
  boys like girls
  and girls like boys
  or girls like bunnies
Grammar for Simple Expressions

E → integer
E → ( E + E )

• Simple expressions:
  ▪ An E can be an integer.
  ▪ An E can be ‘(’ followed by an E followed by ‘+’ followed by an E followed by ‘)’

• Set of expressions defined by this grammar is a recursively-defined set
  ▪ Is language finite or infinite?
  ▪ Do recursive grammars always yield infinite languages?

• Here are some legal expressions:
  ▪ 2
  ▪ (3 + 34)
  ▪ ((4+23) + 89)
  ▪ ((89 + 23) + (23 + (34+12)))

• Here are some illegal expressions:
  ▪ (3
  ▪ 3 + 4

• The tokens in this grammar are (, +, ), and any integer
Parsing

- Grammars can be used in two ways
  - A grammar defines a language (i.e., the set of properly structured sentences)
  - A grammar can be used to parse a sentence (thus, checking if the sentence is in the language)

- To parse a sentence is to build a parse tree
  - This is much like diagramming a sentence

- Example: Show that ((4+23) + 89) is a valid expression E by building a parse tree
Recursive Descent Parsing

- Idea: Use the grammar to design a *recursive program* to check if a sentence is in the language
- To parse an expression E, for instance
  - We look for each terminal (i.e., each *token*)
  - Each nonterminal (e.g., E) can handle itself by using a *recursive call*
- The grammar tells how to write the program!

```java
public static Node parseE(Scanner scanner) {
    if (scanner.hasNextInt()) {
        int data = scanner.nextInt();
        return new Node(data);
    }
    check(scanner, '(');
    left = parseE(scanner);
    check(scanner, '+');
    right = parseE(scanner);
    check(scanner, ')');
    return new Node(left, right);
}
```

```java
boolean parseE() {
    if (first token is an integer) return true;
    if (first token is '(') {
        parseE();
        Make sure there is a '+' token;
        parseE();
        Make sure there is a ')' token;
        return true;
    }
    return false;
}
```
Using a Parser to Generate Code

- We can modify the parser so that it generates stack code to evaluate arithmetic expressions:
  
  2  
  PUSH 2  
  STOP

  (2 + 3)  
  PUSH 2  
  PUSH 3  
  ADD  
  STOP

- Goal: Method parseE should return a string containing stack code for expression it has parsed

- Method parseE can generate code in a recursive way:
  - For integer i, it returns string “PUSH i”
  - For (E1 + E2),
    - Recursive calls for E1 and E2 return code strings c1 and c2, respectively
    - For (E1 + E2), return c1 + c2 + “ADD\n”
  - Top-level method should tack on a STOP command after code received from parseE
Does Recursive Descent Always Work?

- There are some grammars that cannot be used as the basis for recursive descent
  - A trivial example (causes infinite recursion):
    - $S \rightarrow b$
    - $S \rightarrow Sa$

- Can rewrite grammar
  - $S \rightarrow b$
  - $S \rightarrow bA$
  - $A \rightarrow a$
  - $A \rightarrow aA$

- For some constructs, recursive descent is hard to use
  - Can use a more powerful parsing technique (there are several, but not in this course)
Syntactic Ambiguity

- Sometimes a sentence has more than one parse tree
  
  \[ S \rightarrow A \mid aaxB \]
  
  \[ A \rightarrow x \mid aAb \]
  
  \[ B \rightarrow b \mid bB \]
  
  - The string aaxbb can be parsed in two ways

- This kind of ambiguity sometimes shows up in programming languages

\[ \text{if E1 then if E2 then S1 else S2} \]

Which then does the else go with?

- This ambiguity actually affects the program’s meaning

- How do we resolve this?
  
  - Provide an extra non-grammar rule (e.g., the else goes with the closest if)
  
  - Modify the language (e.g., an if-statement must end with a ‘fi’)
  
  - Operator precedence (e.g., 1 + 2 * 3 should always be parsed as 1 + (2 * 3), not (1 + 2) * 3)
  
  - Other methods (e.g., Python uses amount of indentation)