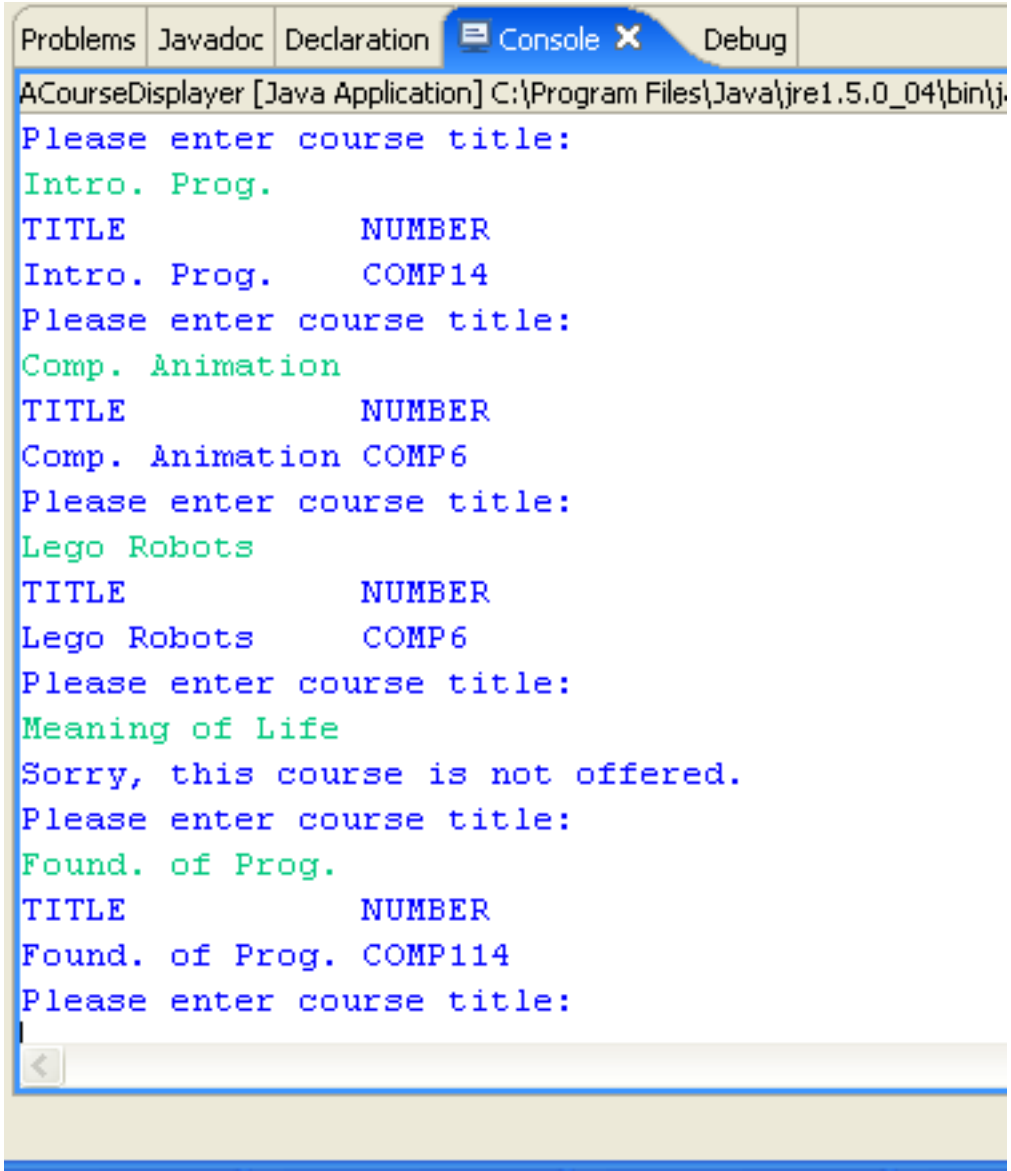
Comp 401
Grammars, Recursive Descent

Instructor: Prasun Dewan

Prerequisites

- Inheritance Abstract Classes

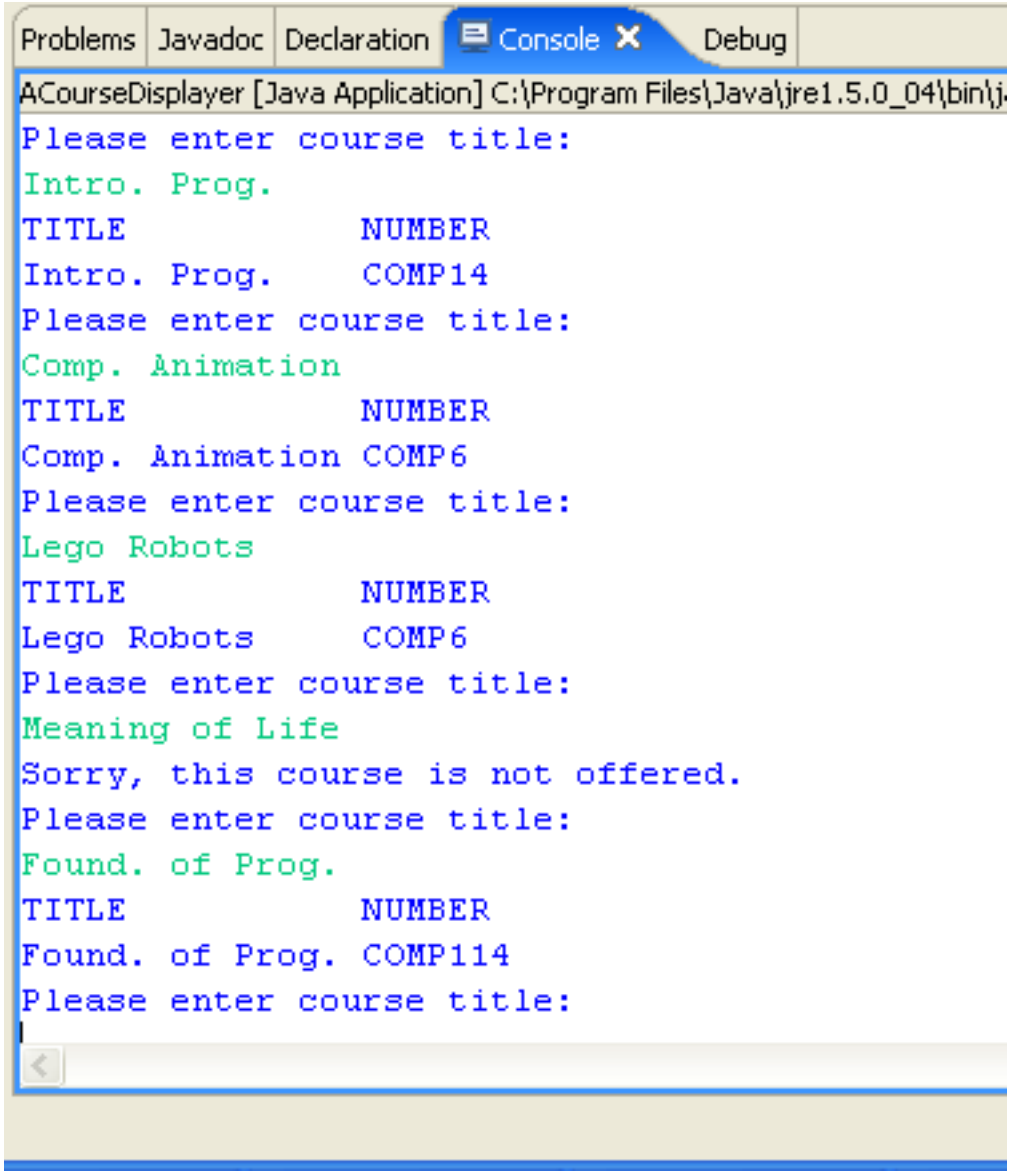
Predefined Course List



Predefined List Filled by Program

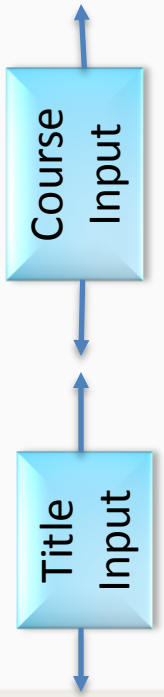
```
static CourseList courses = new ACourseList();
static void fillCourses() {
   courses.addElement(new ARegularCourse ("Intro. Prog.",  "COMP", 14));
   courses.addElement(new ARegularCourse ("Found. of Prog.", "COMP", 114));
   courses.addElement(new AFreshmanSeminar("Comp. Animation", "COMP"));
   courses.addElement(new AFreshmanSeminar("Lego Robots",  "COMP"));
}
```

Filling it Interactively?



Filling Course List Interactively

```
ACourseDisplayer [Java Application] C:\Program Files\Java\jre1.5.0_04\bin\javaw.exe (Sep 27, 2005 12:52:26 PM)
Please enter course info, terminating with a period:
RC
Intro. Prog.
COMP
14
FS
Random Thoughts
COMP
.
Please enter course title:
Random Thoughts
TITLE                NUMBER
Random Thoughts COMP6
Please enter course title:
Meaning of life
Sorry, this course is not offered.
Please enter course title:
```



Formal way of describing input syntax?

Analyzing English

<Sentence List> → <Sentence>*

<Sentence> → <Imperative Sentence> | <Interrogative Sentence> | …

<Imperative Sentence> → <Transitive Verb> <Object>

<Transitive Verb> → study | play | …

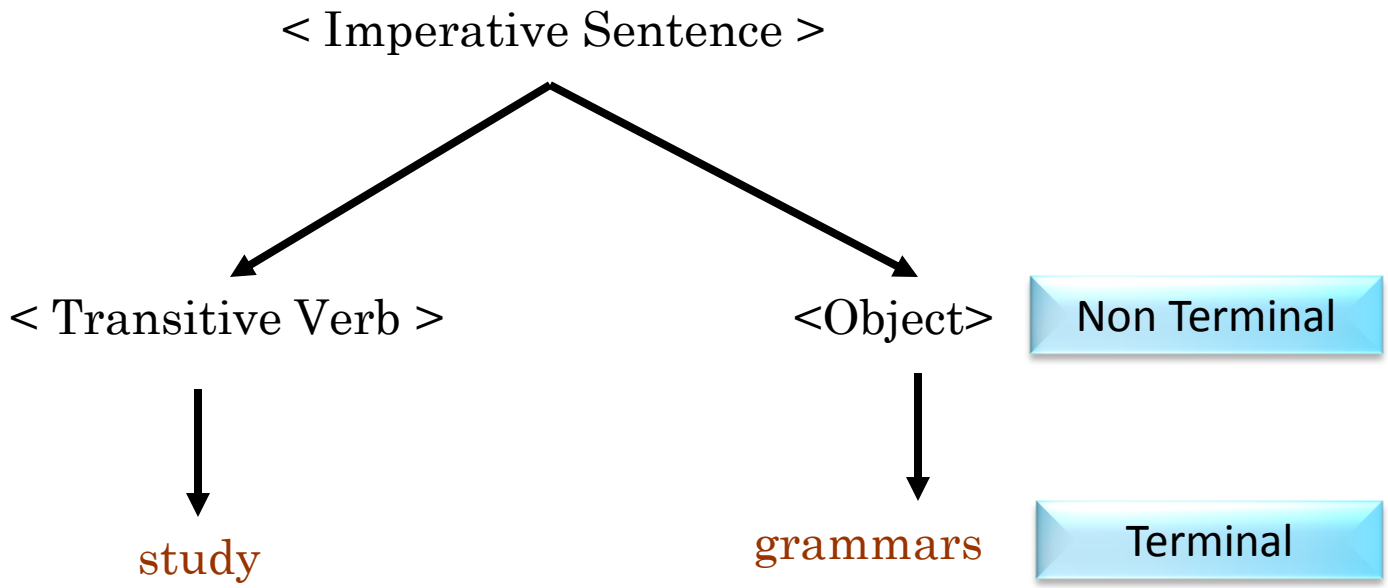
<Object> → grammars | tennis | …

Correct Imperative Sentence?

study grammars play tennis play grammars study tennis tennis 

Angle brackets?

Terminals vs. Non Terminals

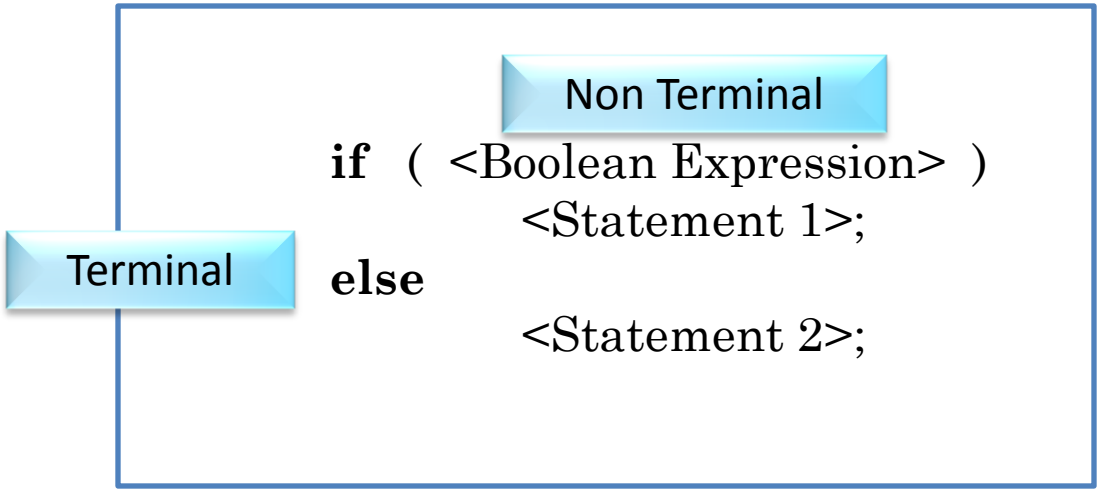


Terminal is a token produced by the scanner (spell checker) that is not further expanded into other tokens – terminates

Set of all terminals is the vocabulary on which the grammar is based

Non terminal is a place holder for a legal sequence of tokens and is associated with rules describing how to derive these sequences

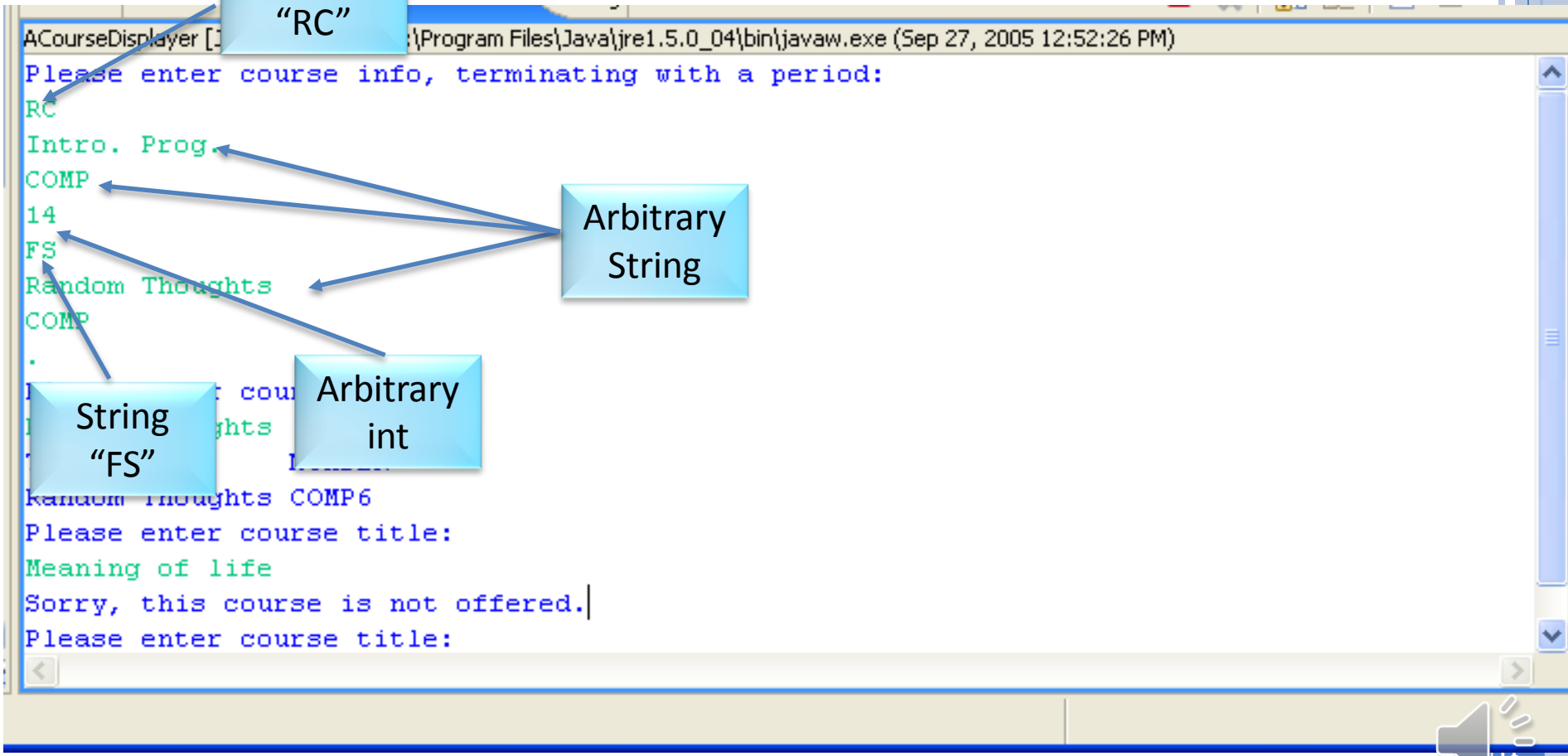
Grammar: If-Else Statement



Filling Course List Interactively

Each program that processes input defines a grammar

Terminals (Tokens)?



Analyzing Grammars

Production/Rule

Repetition (0 or more)

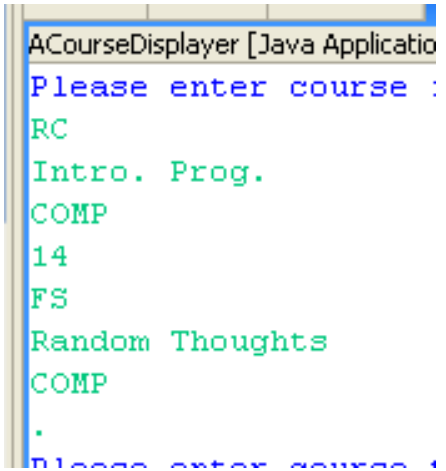
LHS

Root

Alternation

RHS

<CourseList> → <Course> *.
<Course> → <RC> | <FS>
<RC> → RC <Title> <Dept> <Number>
<FS> → FS <Title> <Dept>
<Title> → String
<Dept> → String
<Number> → Integer



First

First will be used to decide on alternation

Grammar-based Concepts

Root: A distinguished non-terminal in the grammar that is either explicitly marked or does not appear on the RHS

- E.g. <Course List>, <Sentence>

Sentence: Any sequence of terminals

- E.g.: help offer, offer help, RC COMP 14 Intro. Prog. FC

Legal sentence: A sequence of terminals derived from the root using the grammar rules.

- E.g.: offer help, RC Intro. Prog. COMP 14 FS Random Thoughts

Legal phrase: A sequence of terminals derived from some non-terminal

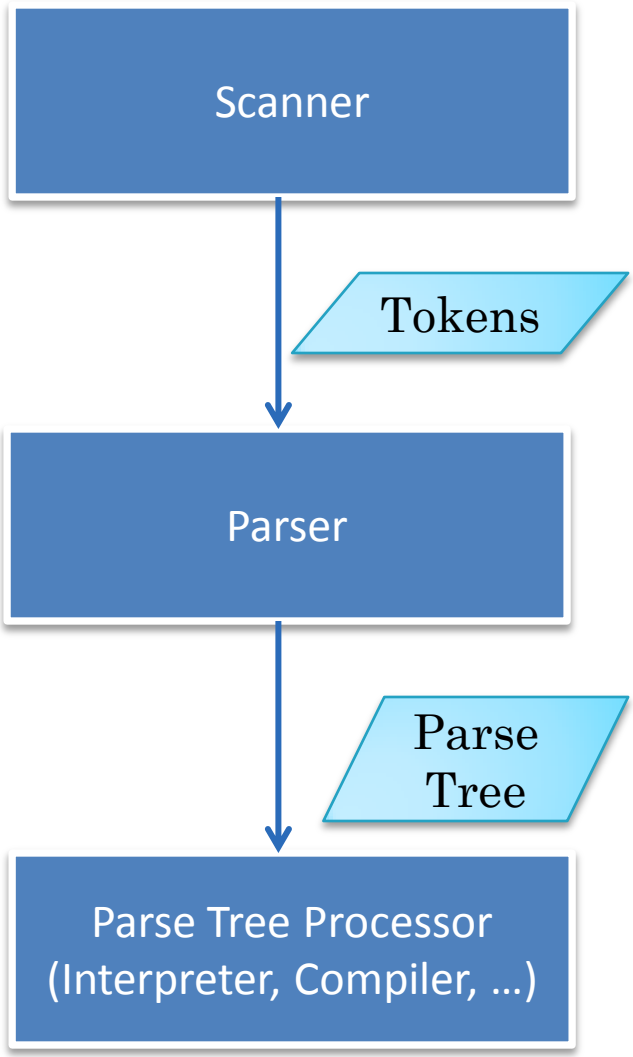
- E.g. RC Intro. Prog. COMP 14

Language: The (possibly infinite) set of legal sentences

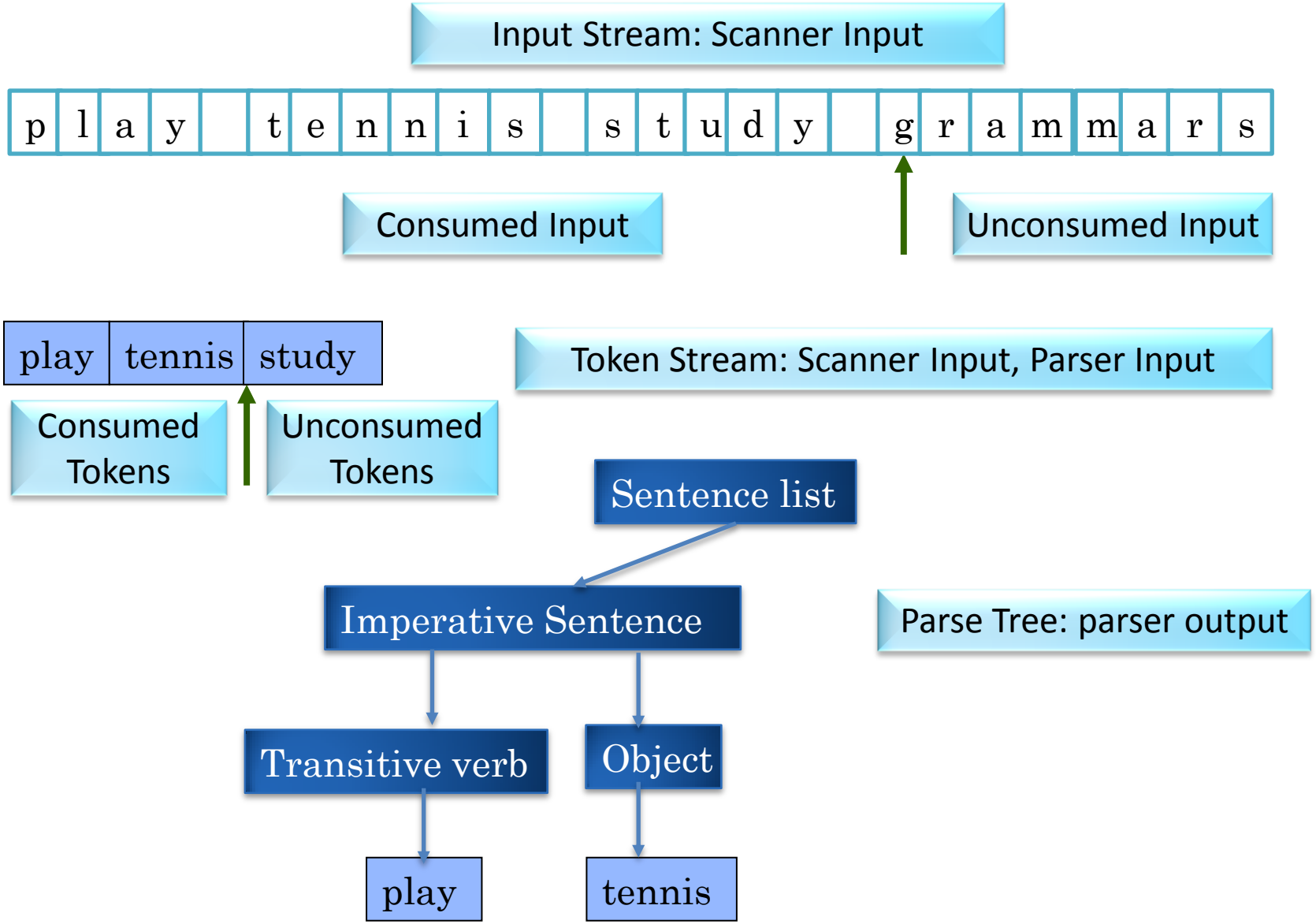
- E.g. {little boy, big boy, little girl, …}

Parser: A program that recognizes a language (set of legal sentences), that is, can distinguish between a legal and illegal sentence and optionally produces a parse tree

Parser Input and Output

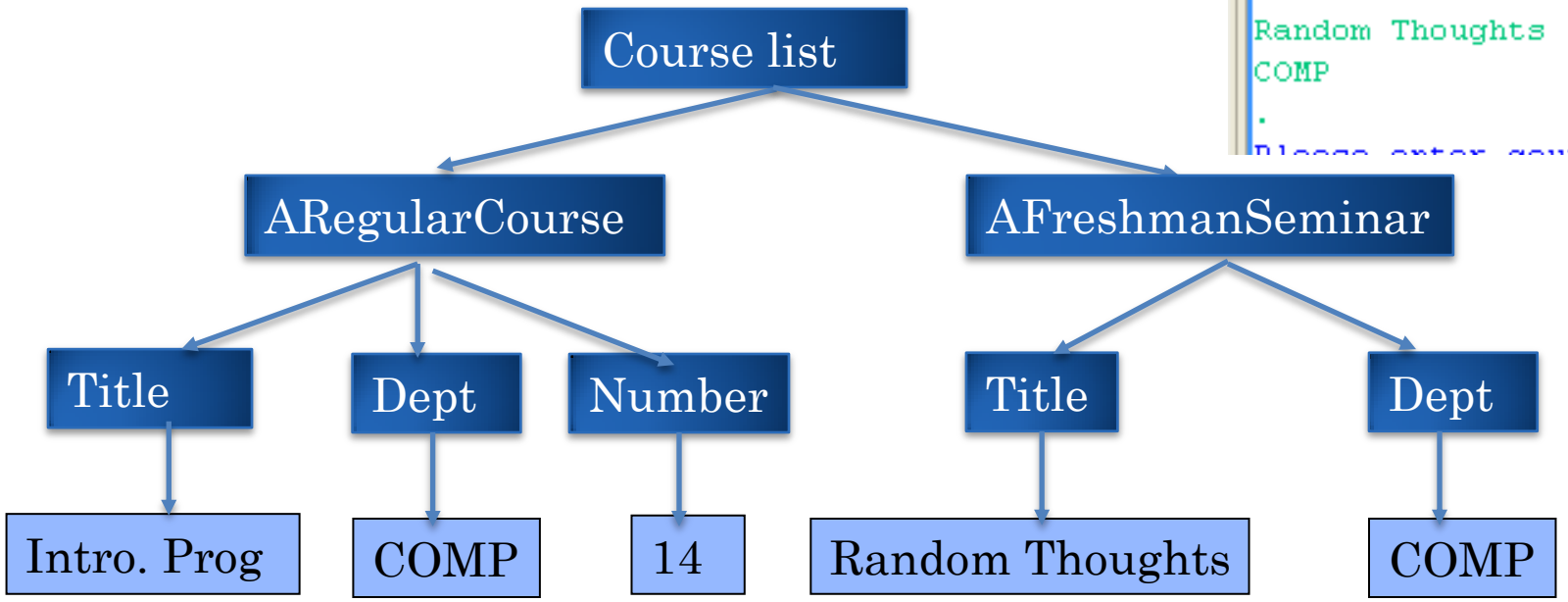


Scanning vs. Parsing

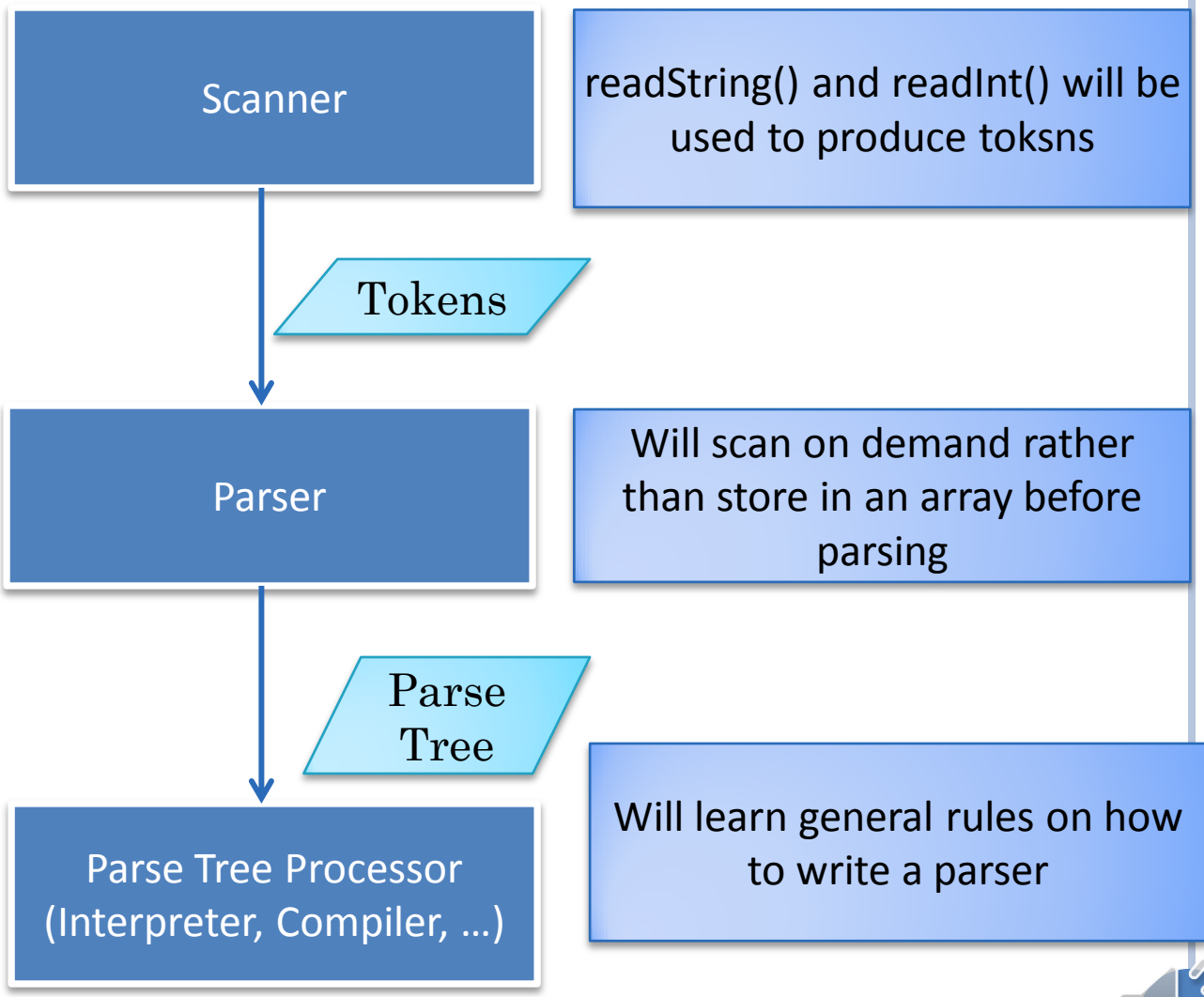


Parse Tree?

```
ACourseDisplayer [Java Applicatio
Please enter course :
RC
Intro. Prog.
COMP
14
FS
Random Thoughts
COMP
.
```



Course Parser Nature



Main class: Previous Main method

```
public class ACourseDisplayer          {
        public static void main(String[] args){
                fillCourses();
                ....
        }
```

Main class: New Main method

```
public class ACourseDisplayer          {
        public static void main(String[] args)          {
                fillCoursesInteractively(); // parser method
                ...
        }
```

<CourseList> → <Course> *

<Course> → <RC> | <FS>

<RC> → RC <Title> <Dept> <Number>

<FS> → FS <Title> <Dept>

<Title> → <String>

<Dept> → <String>

Interactive Filling: Recursive Descent

```
static void fillCours    teractively() {
        System.out    ln("Please enter course info, terminating with a period:");
        courses =    w ACourseParser()).parseCourseList();
}
```

parseCourseList()

parseCourse ()

parseRegularCourse ()

Call parseCourse ()

Call parseFreshmanSemninar ()

Call parseRegularCourse ()

parseFreshmanSeminar ()

<Cours   <Course>

<Cours   <RC> | <FS>

<RC>   C <Title   Dept> <Number>

<FS> → FS <Title> <D

<T   → <String>

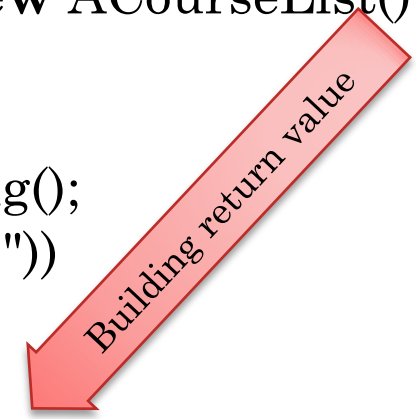
<Dept>   ring>

For each non terminal NT define a method parseNT() that returns a node in the parse tree

If NT^1 is on the LHS and NT^2 is on the RHS of a production then $parseNT^1()$ calls $parseNT^2()$ to get a child node in is return value

Processing Course Sequence

```
package courses;
import collections.CourseList;
import collections.ACourseList;
public class ACourseParser implements CourseParser {
  //<CourseList> → <Course>*
   public CourseList parseCourseList() {
     CourseList courseList = new ACourseList();
     String nextToken;
     while (true) {
          nextToken = readString();
          if (nextToken.equals("."))
                    break;
          else
                     courseList.addElement(parseCourse(nextToken));
     }
     return courseList;
  }
```



Processing Course

```
// <Course> → <RC> | <FS>
Course parseCourse (String firstToken) {
                    if (firstToken.toUpperCase().equals("FS"))
                                return parseFreshmanSeminar();
                    else if (firstToken.toUpperCase().equals("RC"))
                                return parseRegularCourse();
                    else {
                                return null;
                    }
}
```

Each method consumes and processes tokens in the token stream

First token consumed by the caller (parseCourseList()) and processed by this method (parseCourse())

Each method must know where in the token stream is the first unconsumed token

Instead of using Java Scanner to tokenize into strings and numbers, in general some custom scanner used to get tokens (terminals)

Processing Course (Review)

```
// <Course> → <RC> | <FS>
Course parseCourse (String firstToken) {
                    if (firstToken.toUpperCase().equals("FS"))
                                return parseFreshmanSeminar();
                    else if (firstToken.toUpperCase().equals("RC"))
                                return parseRegularCourse();
                    else {
                                return null;
                    }
}
```

Each method consumes and processes tokens in the token stream

First token consumed by the caller (parseCourseList()) and processed by this method (parseCourse())

Each method must know where in the token stream is the first unconsumed token

Instead of using Java Scanner to tokenize into strings and numbers, in general some custom scanner used to get tokens (terminals)

Processing FreshmanSeminar

```
//<FS>  → FS  <Title> <Dept>
Course parseFreshmanSeminar () {
                    String title = readString();
                    String dept = readString();
                    return new AFreshmanSeminar(title, dept);

}
```

Method consumed, RHS Tokens not consumed by caller

FS already consumed!

Processing Regular Course

```
//<RC>  → RC  <Title> <Dept> <Number>
Course parseRegularCourse () {
                String title = readString();
                String dept = readString();
                int number = readInt();
                return new ARegularCourse (title, dept, number);
}
```

Instead of using Java Scanner to tokenize into strings and numbers, in general some custom scanner used to get tokens (terminals)

Recursive Descent Parsing

Non Terminal		Parsing Function
<CourseList>	→ <Course> *.	CourseList parseCourseList()
<Course>	→ <RC> | <FS>	Course parseCourse(nextToken)
<RC>	→ RC <Title> <Dept> <Number>	RegularCourse parseRegularCourse()
<FS>	→ FS <Title> <Dept>	FreshmanSeminar parseFreshmanSeminar()
<Title>	→ String	String readString()
<Dept>	→ String	String readInt()

For each non terminal, create a type, if needed, and a parse function that returns a value of that type, if a value is needed

For each RHS non terminal or terminal, call the parse or scan function that returns corresponding value if needed

Parser Structure

- Each production associated with a parser method.
- Parser method returns object associated with LHS of production *if a return value expected*.
- A parser method can recursively call parser methods to parse non-terminals in the associated production.
- A parser method for some production rule can call parser methods for non-terminals in the production, thereby "recursively descending" into lower levels of the tree deriving the input.
- The first token in the unparsed input is consumed and used to choose between alternatives.
- Such a parser called: recursive descent parser
- Illustrates top-down programming, next level details handled by called methods.

Why Grammars and Parsers