If you are going to form a group for A2, please do it before tomorrow (Friday) noon

GRAMMARS & PARSING

Lecture 8

CS2110 – Spring 2014

Pointers. DO visit the java spec website

Parse trees: Text page 592 (23.34), Figure 23-31

- Definition of Java Language, sometimes useful: http://docs.oracle.com/javase/specs/jls/se7/html/index.html
- Grammar for most of Java, for those who are curious: http://csci.csusb.edu/dick/samples/java.syntax.html

Homework:

- Learn to use these Java string methods: s.length, s.charAt(), s.indexOf(), s.substring(), s.toCharArray(), s = new string(char[] array).
- Hint: These methods will be useful on prelim1! (They can be useful for parsing too…)

Application of Recursion

- So far, we have discussed recursion on integers
  - Factorial, fibonacci, a^n, combinatorials

- Let us now consider a new application that shows off the full power of recursion: *parsing*

- Parsing has numerous applications: compilers, data retrieval, data mining,…

Motivation

- The cat ate the rat.
- The cat ate the rat slowly.
- The small cat ate the big rat slowly.
- The small cat ate the big rat on the mat slowly.
- The small cat that sat in the hat ate the big rat on the mat slowly, then got sick.
- …

- Not all sequences of words are legal sentences

  The ate cat rat the

- How many legal sentences are there?
- How many legal Java programs
- How do we know what programs are legal?

http://docs.oracle.com/javase/specs/jls/se7/html/index.html

A Grammar

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

Grammar: set of rules for generating sentences of a language.

Examples of Sentence:

- boys see bunnies
- bunnies like girls

- The words boys, girls, bunnies, like, see are called *tokens* or *terminals*
- The words Sentence, Noun, Verb are called *nonterminals*

A Grammar

Sentence → Noun Verb Noun

Noun → boys

Noun → girls

Noun → bunnies

Verb → like

Verb → see

- White space between words does not matter
- This is a very boring grammar because the set of Sentences is finite (exactly 18 sentences)

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’

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

Grammar is more interesting than the last one because the set of Sentences is infinite

What makes this set infinite?
Answer:

Recursive definition of Sentence

Detour

What if we want to add a period at the end of every sentence?

Sentence → Sentence and Sentence .

Sentence → Sentence or Sentence .

Sentence → Noun Verb Noun .

Noun → ...

Does this work?

No! This produces sentences like:

$$\underbrace{\underbrace{\text{girls like boys}}_{\textbf{Sentence}}\ .\ \text{and}\ \underbrace{\text{boys like bunnies}}_{\textbf{Sentence}}\ .\ .}_{\textbf{Sentence}}$$

Sentences with Periods

PunctuatedSentence → Sentence .

Sentence → Sentence and Sentence

Sentence → Sentence or Sentence

Sentence → Noun Verb Noun

Noun → boys

Noun → girls

Noun → bunnies

Verb → like

Verb → see

- New rule adds a period only at the end of sentence.
- The tokens are the 7 words plus the period (.)
- Grammar is ambiguous:

  boys like girls

  and girls like boys

  or girls like bunnies

Grammars for programming languages

Grammar describes every possible legal expression

You could use the grammar for Java to list every possible Java program. (It would take forever)

Grammar tells the Java compiler how to understand a Java program

Grammar for Simple Expressions (not the best)

$E \rightarrow$ integer

$E \rightarrow (E + E)$

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?

Some legal expressions:

- 2
- (3 + 34)
- ((4+23) + 89)

Some illegal expressions:

- (3
- 3 + 4

Tokens of this grammar:

(+) and any integer

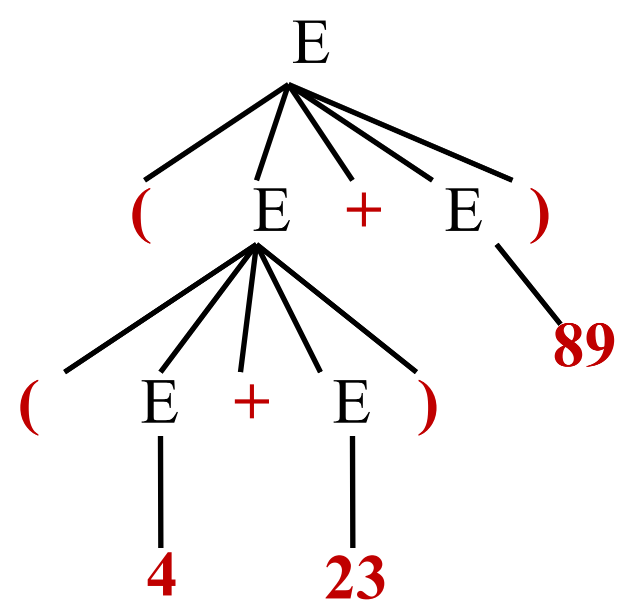
Parsing

Use a grammar 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*: much like diagramming a sentence

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



Recursive Descent Parsing

Write a set of mutually *recursive methods* to check if a sentence is in the language (show how to generate parse tree later)

One method for each nonterminal of the grammar. The method is completely determined by the rules for that nonterminal. On the next pages, we give a high-level version of the method for nonterminal E:

$$E \rightarrow \text{integer}$$

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

Parsing an E

$E \rightarrow$ integer
$E \rightarrow (E + E)$

/** Unprocessed input starts an E. Recognize that E, throwing away each piece from the input as it is recognized.
Return false if error is detected and true if none detected.
Upon return, all processed input has been removed from input. */

public boolean parseE()

before call: already processed unprocessed

(2 + (4 + 8) + 9)

after call: already processed unprocessed
(call returns true)

(2 + (4 + 8) + 9)

Specification: /** Unprocessed input starts an E. …*/

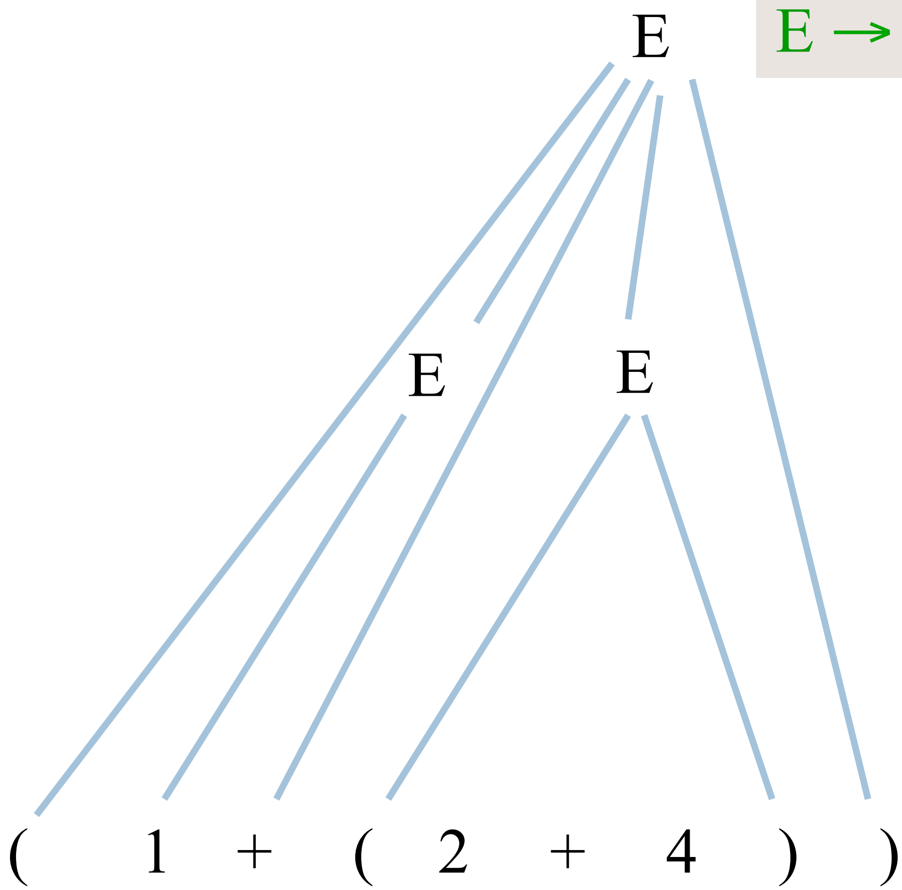
$E \rightarrow$ integer
$E \rightarrow (E + E)$

```
public boolean parseE() {
    if (first token is an integer) remove it from input and return true;
    if (first token is not '(' ) return false else Remove it from input;
    if (!parseE()) return false;
    if (first token is not '+' ) return false else Remove it from input;
    if (!parseE()) return false;
    if (first token is not ')' ) return false else Remove it from input;
    return true;
}
```

Same code used 3 times. Cries out for a method to do that

Illustration of parsing to check syntax

$E \rightarrow$ integer
$E \rightarrow (E + E)$



The scanner constructs tokens

An object scanner of class Scanner is in charge of the input String. It constructs the tokens from the String as necessary.

e.g. from the string "1464+634" build the token "1464", the token "+", and the token "634".

It is ready to work with the part of the input string that has not yet been processed and has thrown away the part that is already processed, in left-to-right fashion.

$$\underbrace{\text{(2 + (4 + 8)}}_{\text{already processed}} \quad \underbrace{\text{+ 9)}}_{\text{unprocessed}}$$

Change parser to generate a tree

$E \rightarrow$ integer
$E \rightarrow (E + E)$

```
/** … Return a Tree for the E if no error.
        Return null if there was an error*/
public Tree parseE() {
    if (first token is an integer) remove it from input and return true;

    if (first token is an integer) {
      Tree t= new Tree(the integer);
      Remove token from input;
      return t;
    }

    …
}
```

Change parser to generate a tree

E → integer
E → (E + E)

```
/** … Return a Tree for the E if no error.
        Return null if there was an error*/
public Tree parseE() {
    if (first token is an integer) … ;
    if (first token is not '(' ) return null else Remove it from input;
    Tree t1= parse(E); if (t1 == null) return null;
    if (first token is not '+' ) return null else Remove it from input;
    Tree t2= parse(E); if (t2 == null) return null;
    if (first token is not ')' ) return false else Remove it from input;
    return new  Tree(t1, '+', t2);
}
```

Using a Parser to Generate Code

- Code for 2 + (3 + 4)

```
PUSH 2
PUSH 3
PUSH 4
ADD
ADD
```

ADD removes the two top values from the stack, adds them, and placed the result on the stack

parseE can generate code as follows:

- For integer i, return string "PUSH " + i + "\n"
- For (E1 + E2), return a string containing
  - Code for E1
  - Code for E2
  - "ADD\n"

Does Recursive Descent Always Work?

Some grammars cannot be used for recursive descent

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

Other parsing techniques exists – take the compiler writing course

Syntactic Ambiguity

Sometimes a sentence has more than one parse tree

$S \rightarrow A \mid aaxB$

$A \rightarrow x \mid aAb$

$B \rightarrow b \mid bB$

aaxbb can be parsed in two ways

This kind of ambiguity sometimes shows up in programming languages. In the following, which **then** does the **else** go with?

if E1 **then** **if** E2 **then** S1 **else** S2

Grammar that gives precedence to * over +

E -> T { + T }

T -> F { * F }

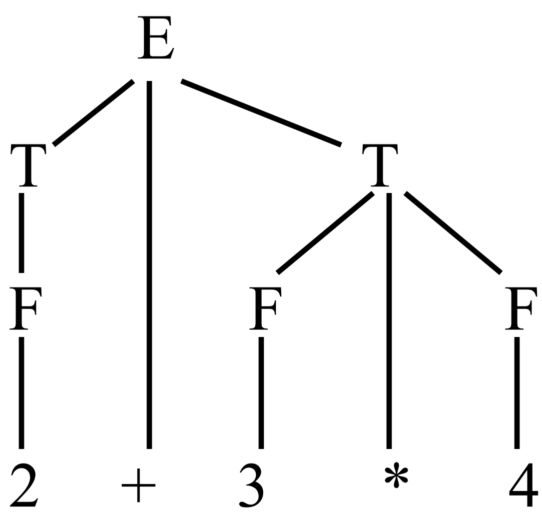
F -> integer

F -> (E)

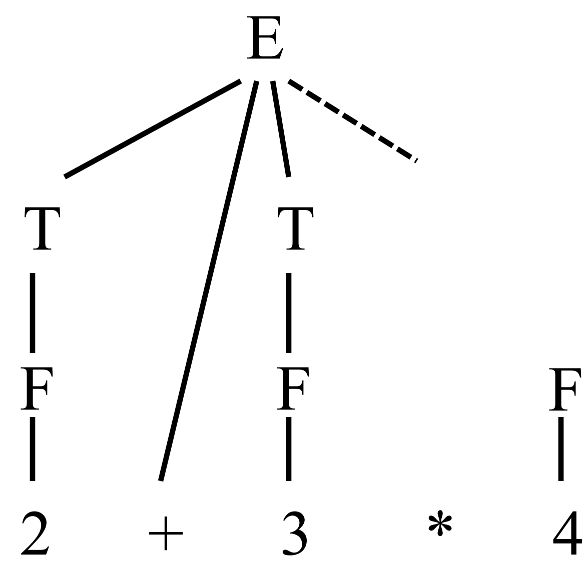
Notation: { xxx } means 0 or more occurrences of xxx.

E: Expression **T:** Term

F: Factor



says do * first



Try to do + first, can’t complete tree

Syntactic Ambiguity

This kind of ambiguity sometimes shows up in programming languages. In the following, which **then** does the **else** go with?

if E1 **then if** E2 **then** S1 **else** S2

This ambiguity actually affects the program's meaning

Resolve it by either

(1) Modify the grammar to eliminate the ambiguity (best)
(2) Provide an extra non-grammar rule (e.g. else goes with closest if)

Can also think of modifying the language (require end delimiters)

Exercises

Think about recursive calls made to parse and generate code for simple expressions

2

(2 + 3)

((2 + 45) + (34 + -9))

Derive an expression for the total number of calls made to parseE for parsing an expression Hint: think inductively

Derive an expression for the maximum number of recursive calls that are active at any time during the parsing of an expression (i.e. max depth of call stack)

Exercises

Write a grammar and recursive program for sentence palindromes that ignores white spaces & punctuation

Was it Eliot's toilet I saw?

No trace; not one carton

Go deliver a dare, vile dog!

Madam, in Eden I'm Adam

Write a grammar and recursive program for strings A^nB^n

AB

AABB

AAAAAAABBBBBBB

Write a grammar and recursive program for Java identifiers

<letter> [<letter> or <digit>]$^{0\ldots N}$

j27, but not 2j7