Applying Recursion:

Grammars and Parsing

Time flies like an arrow.
Fruit flies like a banana.

NOT Weiss: ch 11
Parsing

• Parse\(^1\): *v.* To resolve into its elements, as a sentence, pointing out the several parts of speech, and their relation to each other by government or agreement; to analyze and describe grammatically.

• Parsing used everywhere:
  – Understanding user input
  – Processing data
  – Compilers (e.g., parsing Java programs)
  – …

• Notes:
  – Weiss chapter 11 covers parsing, but assumes too much… we will not use Weiss for this topic. You can read it if you like.
  – We are still working in our primitive Java, with only static methods and variables.

\(^1\)Webster's Revised Unabridged Dictionary
Grammars & Languages

• A language is a set of valid sentences
• A grammar specifies which sentences are valid
• e.g. 2-year-old-language (2YOL):
  
  go town!    go down town!
  break wood half!    break house!
  go up town!    cut bread!
  saw house half!    go home!
  saw wood!
  cut bread half!
+ (too many to list)

• What is are the rules of grammar?
Baby Steps

• **Rules:** always two or three words, followed by ‘!’

  Sentence : Word Word Word !
  Sentence : Word Word !
  Word : *town*
  Word : *go*
  Word : *hammer*
  ...

• **Simplified notation:**

  Sentence : Word Word Word ! | Word Word !
  Word : *town* | *go* | *hammer* | ...

• Whitespace is irrelevant
• This grammar can be used to parse all the sentences…
• … but it also generates nonsense sentences:
  e.g. *town town town !*
A Better Grammar for 2YOL

• Better rules:
  – *go* always used with a place
  – *town* can be modified with *up* or *down*
  – actions (*saw, hammer, cut*) always used with things
  – action-sentences can be modified with *half*

• Better Grammar:
  
  Sentence : *go* Place ! | Action Thing ! | Action Thing half !
  Place : *home* | *town* | PlaceModifier *town*
  PlaceModifier : *up* | *down*
  Action : *cut* | *saw* | *hammer*
  Thing : *bread* | *house* | *wood*
Recursive Grammars

• 2YOL+ : The 2-year-old learns the word and:
  go home and cut bread half and go town!

• Modified Grammar (1st attempt):
  Sentence : Sentence and Sentence
  Sentence : go Place ! | Action Thing ! | Action Thing half !
  Place : home | town | PlaceModifier town
  PlaceModifier : up | down
  Action : cut | saw | hammer
  Thing : bread | house | wood
Recursive Grammars

• 2YOL+ : The 2-year-old learns the word *and*:
  *go home and cut bread half and go town!*

• Modified Grammar (1st attempt):
  Sentence : Sentence *and* Sentence
  Sentence : *go* Place ! | Action Thing ! | Action Thing *half* !
  Place : *home* | *town* | PlaceModifier *town*
  PlaceModifier : *up* | *down*
  Action : *cut* | *saw* | *hammer*
  Thing : *bread* | *house* | *wood*

• Allows *go home! and cut bread!*

• Getting the ‘!’ right:

TopLevelSentence : Sentence !
Sentence : Sentence and Sentence
Sentence : go Place | Action Thing | Action Thing half
Place : home | town | PlaceModifier town
PlaceModifier : up | down
Action : cut | saw | hammer
Thing : bread | house | wood

• Introduce a TopLevelSentence (non-recursive) that adds the ‘!’
Expressions (simplified)

• Grammar:

  Expression : integer  
  Expression : ( Expression + Expression )

• Legal or no?

  – (1 + 2)
  – ((3 + 5) + 2)
  – (4) + (1)
  – 1 + 1
  – (1 + 1 + (1 + (1 + 1)))))))
  – (3 +
Parsing Expressions

• Goal: read in sentences, decide if they are legal or not, and break into pieces.
• Eventual goal: do something with the pieces.
Helper: class Tokenizer

- Breaks input into tokens of various types:
  - INTEGER: such as 1, 24, 0, -3
  - WORD: such as x, r39, foo (legal Java variable names)
  - OPERATOR: such as %, *, +, ! (everything else)

- Initializing:
  - void Tokenizer.takeInputFrom(...);

- Peek at type of next token:
  - int Tokenizer.peekAtKind();

- Get next token, of a particular type:
  - int Tokenizer.getInt();
  - int Tokenizer.getWord();
  - int Tokenizer.getOp();

- Others: Tokenizer.check(...), Tokenizer.match(...)
public class Simple {
    public static void main(String[] args) {
        Tokenizer.takeInputFrom(System.in);
        getExpression();
        System.out.println("okay");
    }

    // uses Tokenizer to read in one expression
    public static void getExpression() {
        if (Tokenizer.check('(')) {
            // must be in "Exp: (Exp + Exp)" case
            getExpression();
            Tokenizer.getOp();
            getExpression();
            Tokenizer.match(')');
        } else {
            // must be in "Exp: integer" case
            Tokenizer.getInt();
        }
    }
}
When Errors Are Encountered

Welcome to DrJava.
> java Simple

(3 + x)

java.lang.Error: Attempt to read 'x' as an integer
    at Tokenizer.getInt(Tokenizer.java:62)
    at Simple.getExpression(Simple.java:18)
    at Simple.getExpression(Simple.java:14)
    at Simple.main(Simple.java:4)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:324)

>
Interesting Cases

• What happens on the following input:
  (1 / 0)
  (1) 2
  3 + 3
  (4)
Problems

• Wrong grammar:
  – Never checked if `Tokenizer.getOp() == ‘+’`
    • if (Tokenizer.getOp() != ‘+’) …
  – Never checked if all input was read or not
    • if (Tokenizer.peekAtKind() != Tokenizer.EOF) …

• Error handling:
  – Not very graceful: Tokenizer throws Errors when it encounters a problem, which typically halt the program.
  – For now, halt with error message is okay.
  – Next week: proper exception handling.
An Even/Odd Calculator

// uses Tokenizer to read in one expression
// and returns true if it evaluates to an even number
public static boolean getExpressionIsEven() {
    if (Tokenizer.check('(')) {
        // must be in "Exp: (Exp + Exp)" case
        boolean lhsEven = getExpressionIsEven();
        if (Tokenizer.getOp() != '+') throw Error("oops");
        boolean rhsEven = getExpressionIsEven();
        Tokenizer.match(')');
        return (lhsEven == rhsEven);
    } else {
        // must be in "Exp: integer" case
        int val = Tokenizer.getInt();
        return (2 * (val/2) == val);
    }
}

Result is even if either:
- both lhs and rhs are
- neither lhs or rhs are

In other words:
\[ \text{lhs} =\equiv \text{rhs} \]

Checking for even-ness using integer division trick
Tips for Recursive Programming

• Double check your algorithm:
  – Reason about base cases – did you get them all?
  – Make sure you are making progress towards base cases

• Don’t try to “unwind” in your head. Instead:
  – Write down “preconditions” and “postconditions”
  – Make sure each recursive call satisfied preconditions
  – Make sure postconditions will be satisfied at end, assuming that the recursive calls worked
  – Always assume the recursive calls will work!
public static boolean getExpressionIsEven() {
    if (Tokenizer.check('(')) {
        // must be in "Exp: (Exp + Exp)" case
        boolean lhsEven = getExpressionIsEven();
        if (Tokenizer.getOp() != '+') throw Error("oops");
        boolean rhsEven = getExpressionIsEven();
        Tokenizer.match(')');
        return (lhsEven == rhsEven);
    } else {
        // must be in "Exp: integer" case
        int val = Tokenizer.getInt();
        return (2 * (val/2) == val);
    }
}