ADT: Abstract Data Type

Just like a type: Bunch of values together with operations on them.

/** an implementation of stack implements a LIFO list of values of type E. */
public interface<E> stack {
    /** = "the list is empty." */
    boolean isEmpty();
    /** Push v onto the top of the list. */
    void push(E v);
    /** Remove top value on the list and return it. 
     * Precondition: the list is not empty. */
    E pop();
}

Java interface is a great way to define an ADT!

Expression trees

Can draw a tree for \((2 + 3) \times (1 + (5 - 4))\)

```
public abstract class Exp {
    /* return the value of this Exp */
    public abstract int eval();
}
```

```
public class Int extends Exp {
    int v;
    public Int(int v) {
        this.v = v;
    }
    public int eval() {
        return v;
    }
}
```

```
public class Add extends Exp {
    Exp left;
    Exp right;
    public int eval() {
        return left.eval() + right.eval();
    }
}
```

Pointers to material

- Parse trees: text, section 23.36
- Grammar for most of Java, for those who are curious: docs.oracle.com/javase/specs/jls/se7/html/jls-18.html
- Tree traversals --preorder, inorder, postorder: text, sections 23.13 .. 23.15.
prefix and postfix notation proposed by Jan Lukasiewicz in 1951

Postfix notation is often called RPN for Reverse Polish Notation

Postfix notation is easy to compute. Process elements left to right.
1. Number? Push it on a stack
2. Binary operator? Remove two top stack elements, apply operator to it, push result on stack
3. Unary operator? Remove top stack element, apply operator to it, push result on stack

Expression trees

public class Add extends Exp {
  public abstract int eval();
  public abstract String pre();
  public abstract String post();
}

/* Return the value of this exp. */
public int eval() {return left.eval() + right.eval();}

/* Return the preorder. */
public String pre() {return "+ " + left.pre() + right.pre();}

/* Return the postorder. */
public String post() {return left.post() + " + " + right.post();}

Motivation for grammars

- The cat ate the rat.
- The cat ate the 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.
- How many legal sentences are there?
- How many legal Java programs?
- How do we know what programs are legal?

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

Examples of Sentence:
- boys see bunnies
- bunnies like girls

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:
girls like boys . and boys like bunnies .

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 end of sentence.
Tokens are the 7 words plus the period (.)
Grammar is ambiguous:
- boys like girls
- and girls like boys
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 "parse" a Java program.

docs.oracle.com/javase/specs/jls/se7/html/jls-2.html#jls-2.3

Grammar for simple expressions (not the best)

E \rightarrow \text{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 ')'.

Some legal expressions:
- 2
- (3 + 34)
- ((4+23) + 89)

Some illegal expressions:
- (3
- 3 + 4

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 + 4)
- ((4+23) + 89)

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 a string is a sentence is in the language)

To parse a sentence is to build a parse tree: much like diagramming a sentence

E \rightarrow \text{integer} \\
E \rightarrow (E + E)

Example: Show that 

\[(4+23) + 89\]

is a valid expression E by building a parse tree.

Ambiguity

Grammar is ambiguous if it allows two parse trees for a sentence. The grammar below, using no parentheses, is ambiguous. The two parse trees to the right show this. We don't know which + to evaluate first in the expression 1 + 2 + 3

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

/** 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 no errors. */

public boolean parseE()

before call: already processed unprocessed

\[( 2 + ( 4 + 8 ) + 9 )\]

after call: already processed unprocessed

(call returns true)

\[( 2 + ( 4 + 8 ) + 9 )\]
**Unprocessed input starts an E.**

```java
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**

![Diagram of parsing process](image)

---

**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.

- 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.

```
( 2 + ( 4 + 8 ) + 9 )
```

---

**Change parser to generate a tree**

```java
/** ... 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) {
    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);
}
```

---

**Code for a stack machine**

```
Code for 2 + (3 + 4)
```

```
PUSH 2
PUSH 3
PUSH 4
ADD
ADD
ADD: remove two top values from stack, add them, and place result on stack
```

It's postfix notation! 
```
2 3 4 + +
```

---
Code for a stack machine

<table>
<thead>
<tr>
<th>Code for 2 + (3 + 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH 2</td>
</tr>
<tr>
<td>PUSH 3</td>
</tr>
<tr>
<td>PUSH 4</td>
</tr>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>ADD: remove two top values from stack, add them, and place result on stack</td>
</tr>
</tbody>
</table>

It's postfix notation! 2 3 4 ++

Use parser to generate code for a stack machine

<table>
<thead>
<tr>
<th>Code for 2 + (3 + 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>parseE can generate code as follows:</td>
</tr>
<tr>
<td>PUSH 2</td>
</tr>
<tr>
<td>PUSH 3</td>
</tr>
<tr>
<td>PUSH 4</td>
</tr>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>ADD: remove two top values from stack, add them, and place result on stack</td>
</tr>
</tbody>
</table>

It's postfix notation! 2 3 4 ++

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  

2 + 3 * 4  
Try to do + first, can’t complete tree

Does recursive descent always work?

Some grammars cannot be used for recursive descent

Trivial example (causes infinite recursion):

\[ S \rightarrow b \]
\[ S \rightarrow S a \]

Can rewrite grammar

\[ S \rightarrow b \]
\[ S \rightarrow b A \]
\[ A \rightarrow a \]
\[ A \rightarrow a A \]

For some constructs, recursive descent is hard to use

Other parsing techniques exist – take the compiler writing course

Syntactic ambiguity

Sometimes a sentence has more than one parse tree

\[ S \rightarrow A | a x x B \]
\[ A \rightarrow x | a A b \]
\[ B \rightarrow b | b B \]

axxb 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

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)
Summary: What you should know

- preorder, inorder, and postorder traversal. How they can be used to get prefix notation, infix notation, and postfix notation for an expression tree.
- Grammars: productions or rules, tokens or terminals, nonterminals. The parse tree for a sentence of a grammar.
- Ambiguous grammar, because a sentence is ambiguous (has two different parse trees).
- You should be able to tell whether string is a sentence of a simple grammar or not. You should be able to tell whether a grammar has an infinite number of sentences.
- You are not responsible for recursive descent parsing

Exercises

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

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Was it Eliot's toilet I saw?</td>
<td>No trace, not one carton</td>
</tr>
<tr>
<td>Go deliver a dare, vile dog!</td>
<td>Madam, I'm Adam</td>
</tr>
</tbody>
</table>

Write a grammar and recursive program for strings $A^*B^n$

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>ABB</td>
</tr>
<tr>
<td>AAAAAAABBBBBBBB</td>
<td>AABB</td>
</tr>
</tbody>
</table>

Write a grammar and recursive program for Java identifiers

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

j27, but not 2j7