COMP261 Lecture 18

Parsing 3 of 4

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

Example: Simple expressions

- Consider the following grammar:

```
Expr ::= Num  | Add  |  Sub  | Mul  |  Div
Add  ::= "add"  "("  Expr  ","  Expr  ")"
Sub  ::= "sub"  "("  Expr  ","  Expr  ")"
Mul  ::= "mul"  "("  Expr  ","  Expr  ")"
Div  ::= "div"  "("  Expr  ","  Expr  ")"
Num  ::= an optional  sign followed by a sequence of digits:
         [-+]?[0-9]+
```

- What does a parser based on this grammar look like?
  - There is a method for each non terminal.
  - They need to follow the structure of the grammar rules.

Top Down Recursive Descent Parser

A top down recursive descent parser:

- Built from a set of mutually-recursive procedures
- Each procedure usually implements one of the production rules of the grammar.
- Structure of the resulting program closely mirrors that of the grammar it recognizes.
- Return Boolean if just checking, or parse tree.

Simple Parser:

- Look at next token
- Use token type to choose branch of the rule to follow
- Fail if token is missing or is of a non-matching type.

Requires the grammar rules to be highly constrained:

- Always able to choose next path given current state and next token

Parser for expressions

```
public boolean parseExpr(Scanner s) {
  if ( !s.hasNext() )  { return false; }          // PARSE ERROR
  String token = s.next();
  if ( token is a number ) { return true; }
  if ( token = "add" ) { return parseAdd(s); }
  if ( token = "sub" ) { return parseSub(s); }
  if ( token = "mul" ) { return parseMul(s); }
  if ( token = "div" )  { return parseDiv(s); }
  else                  { return false;  }        // PARSE ERROR
}

public boolean parseAdd(Scanner s) {
  if ( !s.hasNext() )   { return false; }         // PARSE ERROR
  String token = s.next();
  if ( token != "add" ) {return false; }          // PARSE ERROR
  token = s.next();
  if ( token != "(" )     {return false; }        // PARSE ERROR
  …                                                  What's wrong here??
```

5

Accessing the next token

- How does parseAdd access the next token, when parseExpr has already read it?
- Scanner doesn't allow you to get the next token without reading it from the input, so it's no longer there.
- Could implement an alternative scanner class with *current* and *advance* methods.

Looking at next token

- Scanner has methods that test for a particular kind of token:
  hasNextInt, hasNextFloat, hasNextBoolean, …
- Can also check for a particular string:
  - s.hasNext("string to match"):
    → is there another token, and does it match the string?

```
if ( s.hasNext("add") ) { …..
```

- Or for a regular expression:

```
if ( s.hasNext("[-+]?[0-9]+") ) { …..
```

  - true if there is another token, which is an integer

6

Accessing the next token

- Or save the next token in a field of a parser object, which contains the parser methods.
- Can keep the scanner in a field too, rather than pass it to every parser method.
- public class Parser {
  Scanner s;
  Token t = null;
  public Parser(Scanner scanner) { s = scanner; }
  public parseExp() { … }
  …
  }

Parsing Expressions (checking only)

```
public boolean parseExpr(Scanner s) {
   if (s.hasNext("[-+]?[0-9]+"))    { s.next(); return true; }
   if (s.hasNext("add"))            { return parseAdd(s); }
   if (s.hasNext("sub"))            { return parseSub(s); }
   if (s.hasNext("mul"))            { return parseMul(s); }
   if (s.hasNext("div"))            { return parseDiv(s); }
   return false;

public boolean parseAdd(Scanner s) {
   if (s.hasNext("add"))    { s.next(); } else { return false; }
   if (s.hasNext("("))      { s.next(); } else { return false; }
   if (!parseExpr(s))                      { return false; }
   if (s.hasNext(","))      { s.next(); } else { return false; }
   if (!parseExpr(s))                      { return false; }
   if (s.hasNext(")"))      { s.next(); } else { return false; }
   return true;
}
```

Parsing Expressions (checking only)

```
public boolean parseSub(Scanner s) {
   if (s.hasNext("sub"))  { s.next(); } else { return false; }
   if (s.hasNext("("))    { s.next(); } else { return false; }
   if (!parseExpr(s))                    { return false; }
   if (s.hasNext(","))    { s.next(); } else { return false; }
   if (!parseExpr(s))                    { return false; }
   if (s.hasNext(")"))    { s.next(); } else { return false; }
   return true;
}
```

same for parseMul and parseDiv

How do we construct a parse tree?

- Given our grammar:

  Expr ::= Num | Add | Sub | Mul | Div
  Add ::= "add" "(" Expr "," Expr ")"
  Sub ::= "sub" "(" Expr "," Expr ")"
  Mul ::= "mul" "(" Expr "," Expr ")"
  Div ::= "div" "(" Expr "," Expr ")"
  Num ::= an optional sign followed by a sequence of digits:
  [-+]?[0-9]+

- And an expression:

  `add(sub(10, -5), 45)`

- First goal is a concrete parse tree:

Parsing Expressions (checking only)

Alternative, given similarity of Add, Sub, Mul, Div:

```
public boolean parseExpr(Scanner s) {
   if (s.hasNext("[-+]?[0-9]+"))    { s.next();  return true; }
   if (!s.hasNext("add|sub|mul|div"))       { return false; }
   s.next();
   if (s.hasNext("("))  { s.next(); } else  { return false; }
   if (!parseExpr(s))                       { return false; }
   if (s.hasNext(","))  { s.next(); } else  { return false; }
   if (!parseExpr(s))                       { return false; }
   if (s.hasNext(")"))  { s.next(); } else  { return false; }
   return true;
}
```

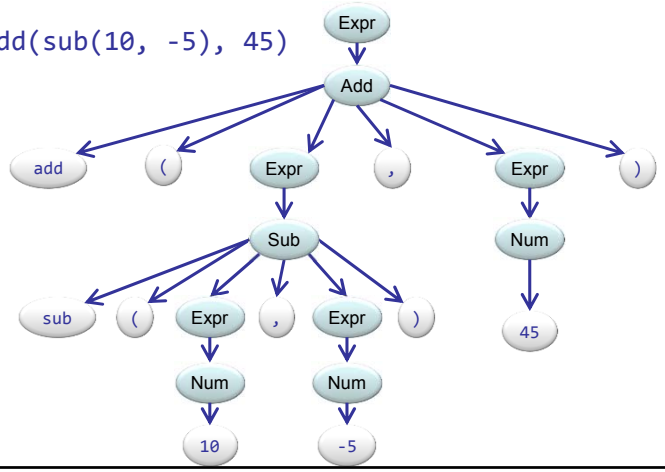
This amounts to changing the grammar to:

Expr ::= Num | Op "(" Expr "," Expr ")"
Op ::= "add" | "sub" | "mul" | "div"
Num ::= [-+]?[0-9]+

And writing the code for parseOP and parseNum inline.

How do we construct a parse tree?

`add(sub(10, -5), 45)`



Building a parse tree

- Need a data structure to represent the syntax tree.
- Can use different type for each kind of expression:
  - Expression node
    - Contains a Number or an Add/Sub/Mul/Div
  - Add, Sub, Mul, Div node
    - Contains the operator, "(", Expression and ")"
  - Number Nodes
    - Contains a number
  - Terminal Nodes
    - Contains a string
- Or: Use a general tree class with a node label to show the type of node.

Building a parse tree

```
class AddNode implements Node {
   final ArrayList<Node> children;
   public AddNode(ArrayList<Node> chn){ children =
   chn; }
   public String toString() {
      String result = "[";
      for (Node n : children){ result += n.toString();
      }
      return result + "]";
   }
}
class SubNode implements Node {
   ...
```

Building a parse tree

```
interface Node { }
class ExprNode implements Node {
   final Node child;
   public ExprNode(Node ch){ child = ch; }
   public String toString() { return "[" + child +
   "]"; }  // Brackets added to show structure.
}
class NumNode implements Node {
   final int value;
   public NumNode(int v){ value = v; }
   public String toString() { return value + ""; }
}
class TerminalNode implements Node {
   final String value;
   public TerminalNode(String v){ value = v; }
   public String toString() { return value; }
}
```

Handling errors

- Can't use false to indicate parse failure.
- Could use null, or add an "Error" node .
- Or, make the parser throw an exception if there is an error:
  - each method either returns a valid Node, or throws an exception.
  - `fail` method throws exception, constructing message and context.

```
public void fail(String errorMsg, Scanner s){
   String msg = "Parse Error: " + errorMsg + " @... ";
   for (int i=0; i<5 && s.hasNext(); i++){
      msg += " " + s.next();
   }
   throw new RuntimeException(msg);
}
```

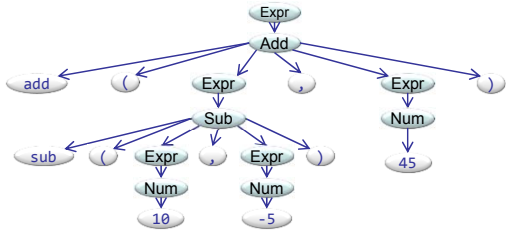
⇒ Parse Error: no ',' @... 34) , mul (

Building a parse tree

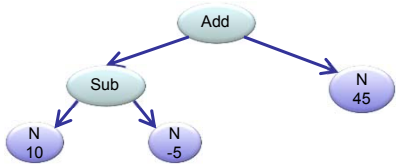
```
public Node parseExpr(Scanner s) {
   if (!s.hasNext())           { fail("Empty expr",s); }
   Node child = null;
   if (s.hasNext("-?\\d+"))    { child = parseNumNode(s);}
   else if (s.hasNext("add"))  { child = parseAddNode(s); }
   else if (s.hasNext("sub"))  { child = parseSubNode(s); }
   else if (s.hasNext("mul"))  { child = parseMulNode(s); }
   else if (s.hasNext("div"))  { child = parseDivNode(s); }
   else { fail("not an expression", s); }
   return new ExprNode(child);
}

public Node parseNum(Scanner s) {
   if (!s.hasNextInt())        { fail("not an integer", s); }
   return new NumNode(s.nextInt());
}
```

What about abstract syntax trees?

- Don't need all the stuff in the concrete parse tree!



- An abstract syntax tree:
- Don't need
  - literal strings from rules
  - useless nodes
    - Expr
    - tokens under Num



Building a parse tree

```
public Node parseAddNode(Scanner s) {
   ArrayList<Node> children = new ArrayList<Node>();
   if (!s.hasNext("add")) { fail("no 'add'", s); }
   children.add(new TerminalNode(s.next()));
   if (!s.hasNext("(")) { fail("no '('", s); }
   children.add(new TerminalNode(s.next()));
   children.add(parseExpr(s));
   if (!s.hasNext(",")) { fail("no ','", s); }
   children.add(new TerminalNode(s.next()));
   children.add(parseExpr(s));
   if (!s.hasNext(")")) { fail("no ')'", s); }
   children.add(new TerminalNode(s.next()));
   return new ExprNode(children);
}
```

Simplify the node classes

```
interface Node {}
```

```
class AddNode implements Node {
   private Node left, right;
   public AddNode(Node lt, Node rt) {
      left = lt;    right = rt;
   }
   public String toString(){return
    "add("+left+","+right+")";}
}
```

Only need the two arguments

```
class SubNode implements Node {
   private Node left, right;
   public SubNode(Node lt, Node rt) {
      left = lt;    right = rt;
   }
   public String toString(){return
    "sub("+left+","+right+")";}
}
```

```
class MulNode implements Node {
   ...
```

Numbers stay the same

```
class NumNode implements Node {
   private int value;
   public NumNode(int value) {
      this.value = value;
   }
   public String toString(){return ""+value;}
}
```

```
public Node parseNum(Scanner s){
   if (!s.hasNext("[-+]?\\d+")){
      fail("Expecting a number",s);
   }
   return new Number(s.nextInt(t));
}
```

parseAdd *etc* are simpler:

Don't need so many children:

Good error messages will help you debug your parser

```
public Node parseAdd(Scanner s) {
   Node left, right;
   if (s.hasNext("add")) { s.next(); }
   else                  { fail("Expecting add",s); }
   if (s.hasNext("("))   { s.next(); }
   else                  { fail("Missing '('",s); }
   left = parseExpr(s);
   if (s.hasNext(","))   { s.next(); }
   else                  { fail("Missing ','",s); }
   right = parseExpr(s);
   if (s.hasNext(")"))   { s.next(); }
   else                  { fail("Missing ')'",s); }
   return new AddNode(left, right);
}
```

Highly repetitive structure!!

ParseExpr is simpler

Don't need to create an Expr node that contains a node:

- Just return the node!

```
public Node parseExpr(Scanner s){
   if (s.hasNext("-?\\d+")) { return parseNum(s); }
   if (s.hasNext("add"))    { return parseAdd(s); }
   if (s.hasNext("sub"))    { return parseSub(s); }
   if (s.hasNext("mul"))    { return parseMul(s); }
   if (s.hasNext("div"))    { return parseDiv(s); }
   fail("Unknown or missing expr",s);
   return null;
}
```

Making parseAdd *etc* even simpler

```
public Node parseAdd(Scanner s) {
   Node left, right;
   require("add", "Expecting add", s);
   require("(", "Missing '('", s);
   left = parseExpr(s);
   require(",", "Missing ','", s);
   right = parseExpr(s);
   require(")", "Missing ')'", s);
   return new AddNode(left, right);
}
```

```
// consume (and return) next token if it matches pat, report error if not
public String require(String pat, String msg, Scanner s){
   if (s.hasNext(pat)) {return s.next(); }
   else { fail(msg, s);  return null;}
}
```

What can we do with an AST?

- We can "execute" parse trees in AST form

```
interface Node {
   public int evaluate();
}
class NumberNode implements Node{
   ...
   public  int evaluate() { return this.value; }
}
class AddNode implements Node{
   ...
   public int evaluate() {
      return left.evaluate() + right.evaluate();
   }
   ...
}
```

Recursive DFS evaluation of expression tree

What can we do with AST?

- We can print expressions in other forms

```
class AddNode implements Node {
   private Node left, right;
   public AddNode(Node lt, Node rt) {
      left = lt;
      right = rt;
   }
   public int evaluate() {
      return left.evaluate() + right.evaluate();
   }
   public String toString(){
      return "(" + left + " + " + ...
   }
}
```

Prints in regular infix notation (with brackets)

Nicer Language

- Allow floating point numbers as well as integers
  - need more complex pattern for numbers.

```
class NumberNode implements Node {
   final double value;
   public NumberNode(double v){
      value= v;
   }
   public String toString(){
      return String.format("%.5f", value);
   }
   public double evaluate(){ return value; }
}
```

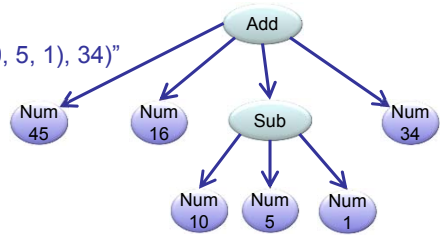
Nicer Language

- Extend the language to allow 2 or more arguments:

Expr ::= Num | Add | Sub | Mul | Div
Add ::= "add" "(" Expr ["," Expr]+ ")"
Sub ::= "sub" "(" Expr ["," Expr]+ ")"
Mul ::= "mul" "(" Expr ["," Expr]+ ")"
Div ::= "div" "(" Expr ["," Expr]+ ")"

sub(16, 8, 2, 1) = 16 – 8 – 2 – 1

"add(45, 16, sub(10, 5, 1), 34)"



Node Classes

```
class NumberNode implements Node {
  final double value;
  public NumberNode(double v){
    value= v;
  }
  public String toString(){
    return String.format("%.5f", value);
  }
  public double evaluate(){ return value; }
}
```

Node Classes

```
class AddNode implements Node {
  final List<Node> args;
  public AddNode(List<Node> nds){
    args = nds;
  }
  public String toString(){
    String ans = "(" + args.get(0);
    for (int i=1;i<args.size(); i++){
      ans += " + "+ args.get(i);
    }
    return ans + ")";
  }
  public double evaluate(){
    double ans = 0;
    for (nd : args) { ans += nd.evaluate(); }
    return ans;
  }
```