1 Review Scanner

- The scanner reads characters and yields tokens.
- It checks the wellformedness of the tokens.
- The class Scanner has a method nextToken(), which yields the input tokens one at a time.
- This class is implemented by jex.lex, from which JLex generates Scanner.java.

2 Review Parser

- The parser reads tokens by calling nextToken().
- It checks, whether the input is syntactically correct.
- It builds an internal data structure of class Tree, the abstract syntax tree.
- The method parse() in class Parser yields this tree.
- This class is implemented by jex.cup, from which JavaCup generates Parser.java.

3 PrettyPrinter, Analyzer, Interpreter

- These are all processes working on the abstract syntax tree.
- The PrettyPrinter prints the abstract syntax tree in readable form.
  - This can support debugging of the compiler.
- The Analyzer relates usage of a name to its definition.
  - It also checks whether all used names are defined.
  - If we do type-checking at compile-time the analyzer does the type-checking as well.
- The Interpreter executes the program.
- Each of these processes is implemented by a visitor.

4 Object-oriented approach

- Disadvantage: One process is spread over many source files.

5 Visitor

- Advantage: A process is in one class. This makes it easy to develop and maintain processes separately.
- Disadvantage: A bit more complicated, but the larger the application, the smaller the difference.

6 Another Extension Problem: A generic HTML-Parser

- We saw that visitors made the extension by new processors easy.
- A generic parser should be usable in different applications.
- Different applications may have different tree representations.
- In our parser the tree representation is fixed. There are calls of the form
  `new Operation(pos, left, right, op).`
- These calls are spread all over the parser.
- A generic parser needs to be able to generate different representations
- General principle: Encapsulate things that vary.
- Idea here: Encapsulate creation in a class.
- This class is called a *factory*, because it creates things.

7 Factories

- A factory is a class which is responsible for creating objects.
- Example: A factory for expression trees:

```
class TreeFactory {
    public Tree mkOperation(int pos, Tree left, Tree right, int op) {
        return new Tree.Operation(pos, left, right, op);
    }
    public Tree mkNumLit(int pos, int i) {
        return new Tree.NumLit(pos, i);
    }
}
```

- Now in the parser, instead of writing

  `new Tree.Operation(pos, left, right, int op);`

  we write

  `treeFact.mkOperation(pos, left, right, int op);`

Now treeFact is an attribute of the parser. To this end we have to change the parser.

```
action code
{: TreeFactory treeFact;
:};
parser code
{: TreeFactory treeFact;
   public Parser(Scanner scanner, TreeFactory treeFact) { ... }
:};
init with
{: action_obj.treeFact = treeFact;
:};
```

and we inititialize the parser now with

```
new Parser(scanner, new TreeFactory());
```


8 Extending: A second Tree

- Assume, we want to have another version where only the structure is important, numbers and operators and positions do not matter.
- We first have to declare a type OtherTree

```
class OtherTree extends Tree {
    class OtherOperation {
        OtherTree left, right;
        public OtherTree Operation(OtherTree left,
            OtherTree right) { ... }
    }
    class OtherNumLit {
        public OtherTree NumLit() { }
    }
}
```

9 Extending: A second TreeFactory

- We write a factory for OtherTree

```
class OtherTreeFactory extends TreeFactory {
        public Tree mkOperation(int pos, Tree left, Tree right, int op) {
                return new OtherTree.Operation((OtherTree) left,
                        (OtherTree) right);
        }
        public Tree mkNumLit(int pos, int i) {
                return new OtherTree.NumLit();
        }
}
```

  and construct a new factory in the main program

```
new Parser(scanner, new OtherTreeFactory());
```

- We could circumvent the casts, if we had generic types (templates in C++).

10 Factories: Other Uses

- We also have to use factories, if we have different versions of our compiler, with different trees, but we want to use the same parser.
- A software package, running under multiple window systems, can use a window factory to create windows (menu bars, menus, scrollbars, ...).
- This is probably the most typical use.
- Generally, if we have a related products
  - trees
  - windows

  and different implementations we can use a factory for creating these products.
- This is easier than switches and new statements.
- It is easy to extend (write another factory) and we do not need to change existing code.
- It is easier to assure, that we don't mix implementations.