

## Building the AST

- We have an RCLASTNode that augments the ASTNode class with some extra things we might need
  - In our AST we will make use of the AbstractViewableASTNode class, so we should make our own version of *that*
    - This will be the class we use
- Something like:

```
public abstract class AbstractViewableRCLASTNode
    extends ast.AbstractViewableASTNode implements RCLASTNode {
```
- But what should be *in* this class?
  - We should implement handy things –
    - Like the error() method
    - The static “indent()” method from PrettyPrint....
    - The type() family...

## AST Design example

- Language specific
  - Many different decompositions of the class hierarchy and object instance hierarchy are possible
  - We look at this *after* thinking about semantic analysis & code generation
- Engineering Rule-of-thumb
  - We are going to have (perhaps) lots of extra interfaces
    - PrettyPrint
    - Code Gen, Type Check etc
  - Aggregate into our own little class – RCLASTNode
    - Make *this* an interface we will implement!
  - Since we don't know about code generation, we'll need to come back and implement this later...

## Some Handy Methods

```
protected void indent(int indentLevel) {
    for (int i = 0; i < indentLevel; i++) {
        System.out.print(" ");
    }
}

public String type() { return "not-Done-Yet!"; }
public String type(String sib) { return "not-Done-Yet!"; }
public String type(String sib1, String sib2)
    { return "not-Done-Yet!"; }
```

- Now if we try to make use of the type() methods we will get an informative string back!
  - Until we over-ride them when we actually implement them!
- Note the software engineering here

## RCLASTNode

```
public interface RCLASTNode extends ast.ASTNode {
    // Pretty prints this node at the given indentation level
    public void prettyPrint(int indentLevel);
    public void typedPrettyPrint(int indentLevel);
    // Handy
    public void error(String msg);
    // Traverse the AST, performing semantic checks
    public void checkSemantics();
    // Return type - 1&2 pars for type rules.
    public String type();
    public String type(String s);
    public String type(String s1, String s2);
}
```

## While Loops

- We will need to implement the following methods:
  - PrettyPrint and TypedPrettyPrint
  - CheckSemantics
- We've already seen the pretty print method (for “if” statements)
  - We will print “while” and then invoke the pretty print method for the first child (the boolean valued expression)
  - Then the second child
- Ahhh – this is common to *all* pretty printing
  - Time to add a method to our most base class that does this!
    - This is how Object Oriented design *really* works – we never start off with a “perfect” class hierarchy and move forward – we need to evolve things!

## Class Hierarchy

- Simplicity is best – let's just follow the guidelines in the lecture notes and see what falls out
- Engineering rule-of-thumb
  - Build one class to represent each of the non-terminals, and for each significant alternative
    - This means we will have (for example) nodes to represent “true” and “false” in the <literal> rule
- Look at *while loops* as an example
  - While loop is a kind of <compound statement>, so we'll look at that too.

## Child Print Methods...

- Back to AbstractViewableRCLASTNode and add:

```
protected void printChild(int childIndex, int indent) {
    Util.myAssert(childIndex < numChildren(), "Index out of bounds");
    ((RCLASTNode) getChildAt(childIndex)).prettyPrint(indent);
}
protected void typedPrintChild(int childIndex, int indent) {
    Util.myAssert(childIndex < numChildren(), "Index out of bounds");
    ((RCLASTNode) getChildAt(childIndex)).typedPrettyPrint(indent);
}
```
- Again, note the SE approach – check everything, everytime!

## Compound Statement

- Compound Statements will be subclasses of Statements...
- We don't need to implement much at all..

```
public class CompoundStatementNode extends StatementNode {
    public void prettyPrint(int indent) {
    }
    public void typedPrettyPrint(int indent) {
    }
    public String toString() {
        return ("<compound statement>");
    }
}
```

## Semantics in AbstractViewableRCLASTNode

- OK, so something like:

```
public void checkSemantics() {
    this.before();
    for (int i = 0; i < numChildren(); i++) {
        ((RCLASTNode) getChildAt(i)).checkSemantics();
    };
    this.after();
}
protected void before() {};
protected void after() {};
```

- Now for many nodes we just need to implement the before or after methods....

## Pretty Print for While Loops

```
public void prettyPrint(int indent) {
    Util.myAssert(numChildren() == 2,
        "WhileStatementNode must have two children");
    System.out.print("while (");
    printChild(0, indent);
    System.out.println(") do");
    printChild(1, indent+2);
}
```

- Simple – and a minimum of effort!
- What about the semantics though?

## While loop semantics check

- Given that, something like this will do for type checking while loops (just check child#0 is a boolean!):

```
protected void after() {
    if (typeOfChild(0).equals("Bool") ||
        typeOfChild(0).equals("unknown")) {
        // OK! No need to do anything here
    } else {
        ((RCLASTNode) getChildAt(0)).error("expected Bool condition");
    }
}
```

- In the same way we implemented printChild(), we'll implement typeOfChild()....

## Semantics Checking

- This was described in lectures as “walking the tree”
  - We have the node (while loop) and its children
  - Recall (Data Structures) that we can do the processing of information on the tree node first, then children; or the other way around.
    - Either way, the “walk” will be kind-of the same
- SE issue (again):
  - Design the checkSemantics() method in the base class to:
    - Invoke a Before() method
    - Run checkSemantics() on all the children
    - Invoke an After() method