Parser Output

- So far, our parser simply returns when there has been a successful parse:
  - Need to make it generate an output – an Abstract Syntax Tree (AST).
- We don’t actually need to do this:
  - We can place all our semantic analysis, code generation etc. in the parser code itself.
  - This is not necessarily a good idea:
    - Less modular
    - More error prone
- The AST is the central data structure of our compiler:
  - The parser builds it
  - The remaining phases annotate it and re-arrange it (perhaps)
- The rest of the compiler is a series of modules that simply walk the AST, performing their intended tasks.
  - Time to review your Data-Structures lectures!

AST - Description

- There is no “built-in” meaning to the AST:
  - Each compiler we build will have a unique kind of parse tree.
- Example Pascal program:
  
  ```
  program foo(output);
  var I : integer;
  begin
    for I = 1 to 100 do 
      writeln(I);
  end .
  ```

- Our parse trees will be extremely simple: we will build a child tree for each syntactic element as we find it, left-to-right.

Tree View

```
program foo(output);
var i : integer;
begin
  for i = 1 to 100 do 
    writeln(i);
end .
```

- Within each of these categories, we simply build more nodes
- So if we have 10 variables, we have 10 children of the “vars” node…

Our AST

- We will make use of a simple AST
  ```
  public interface ASTNode {
    public Iterator childIterator();
    // Adds a child to the list of children
    public void addChild(ASTNode child);
    // Removes the child from the list of children
    public void removeChild(ASTNode node)
      throws ASTNodeNotFoundException;
  ...
  }
  ```

Our AST

- We will make use of a simple AST
  ```
  public interface ASTNode {
    ...
    // returns the number of children
    public int numChildren();
    // returns the parent node or null if this node is a root node
    public ASTNode currentParent();
    // sets the parent of this node
    public void setParent(ASTNode parent);
    // returns information on what the node represents
    public String toStr();
  }
  ```
**AST Infrastructure**

- Implementors of `MutableTreeNode` can be rendered in a `JTree`
- You must extend either `AbstractViewableASTNode` or `AbstractASTNode` for every node type you define
- Suggestion: for your parser derive your AST nodes from `AbstractViewableASTNode`
  - Easier to debug
  - Only slightly less efficient

**Parse Tree HOW-TO**

- For each syntactic category, decide how you’ll lay out the tree
  - Decide what information needs to be stored in each node
  - Think clearly, draw the picture, make sure all the information you’ll need is in there
- Build a concrete implementation that conforms to the interface.
- Program defensively
  - if you expect only n child nodes then test this assertion
- Annotate the syntax specification with tree building notes
  - Remember: there is a 1:1 mapping between the syntax and the parser code
- In the parser code, insert the necessary tree building code.
- After calling the parser, it should yield a parse tree
- Note well – this is an `Abstract` syntax tree
  - It does not contain terminal symbols (that can be deduced)
  - It will only contain terminal symbols like identifiers

**Example**

```java
public void prettyPrint(int currentIndent) {
    /* Pretty Prints the While loop like so:
        while expr do
            statement;
      i.e.: body is indented two spaces...
    */
    for(int i=currentIndent;i>0; i--)
        System.out.print(" ");
    //we must have exactly 2 child nodes
    Utility.assert(numChildren() == 2);
    Iterator iterator = childIterator();
    System.out.println("while ");
    ([Expression][Statement]) prettyPrint();
    System.out.println("do");
    ([Statement][Statement]) prettyPrint(currentIndent + 2);
}
```

**Viewing an AST**

- The above figure shows the resulting window after invoking the main method in `AbstractViewableASTNode`
- In all `ASTNode` implementations write a suitable `toString()` method
  - this text will appear in the tree view
  - an ideal debugging tool for your parser and AST!

**Parse Tree infrastructure**

- You will need to write a `node class` for each programming language construct
  - There will be many classes, but all will be very simple
  - This is object-oriented programming!
- You will need to implement (override) several methods in this class, but we will flesh those details in later
  - Always use a standard interface e.g. `ASTNode`
- Start with a pretty print method for each syntactic category
  - E.g. a Pascal `while` loop class might have a `prettyPrint()` method that:
    - Prints the keyword “while”
    - Invokes the `prettyPrint()` method for the boolean valued expression that controls the loop
    - Prints the keyword “do”
    - Skips to a new line, invokes `prettyPrint()` on the statement list that is the body…
  - Note that you need to pass the `prettyPrint()` method the current indentation level