CSC488
ASSIGNMENT 3
AST DESIGN NOTES

PETER MCCORMICK

Contents

1. Introduction 1
2. General Principles 1
3. Source Locations 1
4. Equality Comparison 2
5. Pretty Printing 2
6. Conclusion 3

1. Introduction

We had two goals which governed our modifications to the AST class hierarchy. The first was that we wanted our compiler to provide helpful and illustrative error messages for end-users, and the second, to give ourselves a high degree of confidence in the consistency and correctness of our parser grammar actions and the AST tree structures that resulted.

2. General Principles

We aimed to keep mutable state in our AST node classes to a minimum. Every action in our parsers' grammar is one line, either a constructor call via `new`, or a method call to a factory that constructs a new AST node instance. There are no setter methods for any of the arguments you specify at the time of the constructor call. This kept the grammar action code itself rather simple, and helped keep the design focus on creating a clean and logical interface to AST node generation.

While the AST is intended to be rather abstract, we did find that it followed the grammar non-terminals rather closely. Given that we wanted good error locality, and given that the semantic actions we would eventually need to take are themselves specified positionally relative to the underlying tokens of program source text, this hybrid concrete & abstract was not too onerous.

Our goal of general immutability is relaxed, however, for semantic checking purposes: we use a setParent() method call on child nodes in order to facilitate easy bi-directional navigation, and the classes related to expression nodes carry an "evaluation type" to help propagate types for type checking and symbol definition purposes.

3. Source Locations

At construction time, every AST node takes a `SourceLoc` instance, which specifies an interval of characters from the program source (given as a beginning and end pair of line and column positions.) This interval allows each node to directly relate to the exact program source from whence it came. Here is an example depicting a language statement with some of the source locations which correspond to child nodes:

```
foo := bar(x + 1, y not= true)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  AssignStmt
^^^                             lval VarRefExpn ~> IdentExpn (containing an IdentNode)
       ^^^^^^^^^^^^^^^^^^^^^^^  rval FunctionCallExpn
       ^^^                      ident IdentNode
           ^^^^^  ^^^^^^^^^^^   arguments ASTList<Expn>
                  ^^^^^^^^^^^   EqualsExpn (which is a BinaryExpn subclass)
                         ^^^^   right BoolConstExpn
```

In `compiler09.ast.SourceLocPrettyPrinter` you can see a simple implementation of pretty printing a source location given the underlying program source itself, generating an underlined highlight of a given `SourceLoc` instance. Since the base AST parent class itself implements that interface, and since every AST node constructed by the grammar actions includes a source location specific to the text interval that generated that node, whenever a semantic error is encountered we are able to easily display for the user exactly what the problematic place is from their original source code. If a variable name is being redefined in a declaration, we are able to highlight the exact `ast.IdentNode` (from within a DeclarationPart from a MultiDeclarations from a Scope from a Program) as it appeared when they wrote it.

In Semantics.java circa lines 1027-1029 you can see that the SourceLoc pretty printer is used to highlight semantic errors. Run a test by hand from the fail directory to see it in action!

4. Equality Comparison

As a sanity check, we ensured that every AST sub- class had a correct `equals` method which performed structural equality comparison (ignoring the source locations attached to each node.) The utility of this was manifest in combination with the final piece.

5. Pretty Printing

We modified the original AST design in regards to how it regenerates valid 488 language source code from a tree of AST nodes. First, we used `toString` methods on subclasses to pretty-print source fragments that represented less than a whole line of output, things like expressions or constants. Next, we added an interface `ASTPrettyPrintable` which the AST base class implements:

```
public interface ASTPrettyPrintable {
    public void prettyPrint(ASTPrettyPrinterContext p);
}
```

The `ASTPrettyPrinterContext` provides primitive methods like print(), println() and newline(), as well as enterBlock() and exitBlock() would governed the nesting depth (and thus indentation level).

In the case of the AST base class, the default implementation relies on the `toString`, which every child overrides:

```
public void prettyPrint(ASTPrettyPrinterContext p) {
    p.print(toString());
}
```

For even a complex type like MultiDeclarations, since they will be pretty-printed on a single line, all it needs to do is override toString:

```
public String toString() {
    return  "var " + elements + " : " + typeDecl;
}
```

Where `elements` (of type `ASTList<DeclarationPart>`) and `typeDecl` (of type `TypeDecl`) provided toString()'s which are used in the coercion.

Since ASTList's are used extensively throughout, sometimes when pretty printing them you would want the output to be list of comma-separated values like function arguments, while for other cases, newline separated with a uniform indentation. To support this, ASTList adds `prettyPrintCommas` and `prettyPrintNewlines`, plus a convenience helper `prettyPrintBlock` which enters a block, prints newlines, and then exits the block.

Putting all of this together, the usefulness of overriding the default `prettyPrint`, and using the pretty printer context, is seen in a multi- line block like an IfStmt:

```
public void prettyPrint(ASTPrettyPrinterContext p) {
    p.println("if " + condition + " then ");
    whenTrue.prettyPrintBlock(p);

    if (whenFalse != null) {
        p.println("else");
```

```
            whenFalse.prettyPrintBlock(p);
        }

        p.println("fi");
    }
```

Just for reference, `whenTrue` and `whenFalse` are of type Scope, and that class has a `prettyPrint` override too which handles the braces:

```
    public void prettyPrint(ASTPrettyPrinterContext p) {
        p.println("{");
        p.enterBlock();
        declarations.prettyPrintNewlines(p);
        statements.prettyPrintNewlines(p);
        p.exitBlock();
        p.print("}");
    }
```

Overall I feel that this code structure is very clean and nicely expresses intent.

With the final piece of the puzzle in place, we were ready for the big boolean check. in main.java there is a helper method `boolean verifyprettyprint(program prog)` which, given an ast generated by the parser, will pretty-print the ast, re-parse that output generating a second ast, and then check the two for structural equality. try it out by passing in the option `--roundtrip` when invoking the compiler.

6. Conclusion

We achieved what we set out to: created confidence in our implementation, and provided user friendly error diagnostics.