Compilers Fall 2015 Project 2

Due We 10/21/2015 at 8:30am.

Project Milestone 2: AST generator

Implement an AST generator for TACK. Your program should parse a TACK file, construct an AST, and print the AST back out. As an example, given the following input TACK code:

```
# Hello-world
main = fun () -> int {
  print("Hello, world!\n");
  -> 0;
}
```

Your program should print the output AST with the following format and indentation:

```
Program
  FunDef
    FunId main
    FunType
      RecordType
      PrimitiveType int
    BlockStmt
      CallStmt
        CallExpr
          FunId print
          StringLit "Hello, world!\n"
      ReturnStmt
        IntLit 0
```

Each line of output shows one AST node. Some of the AST nodes have an attribute, which is printed in the same line after the node name, as in **FunId main**. Each AST node can have zero or more children, which are printed with two extra spaces of indentation in depth-first, left-to-right order. For example, the node **FunDef** has three immediate children **FunId**, **FunType**, and **BlockStmt**. Table 1 describes all AST nodes for TACK.

The following archive file contains input TACK programs and corresponding output ASTs: http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/pr2/test.tar

It is a good idea to also write additional test cases of your own, to maximize test coverage.

Please turn in the entire, self-contained code for your AST generator, including a README with instructions for how to compile and run. It is required that you use Antlr4 for the parser.

Hints and tips

This project milestone involves a substantial amount of coding. You should get started early! The following steps are a strategy that will maximize your chances at succeeding in this milestone.

1. Read the Antlr introduction, especially the section about abstract syntax trees.
   http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/antlr-intro.pdf

2. Read sections 5.1-5.3 in the Dragon book, especially the tree normalizer example in Figure 5.13 (Page 321). This example assumes that you used a grammar that has left-recursion eliminated, which is common in top-down parsing. To get an AST from such a grammar, you would proceed in two steps. First, using embedded actions in that grammar, you generate a "raw AST" with nodes for the helper ("head" / "tail") productions. Second, using a visitor, you rewrite the "raw AST" into the final, normalized, AST.

3. If you don't yet know how the visitor design pattern works, look it up.

4. Read and understand the code in the AST generator example on the class webpage.
   http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/example-ast-gen.tar

5. Starting from the code in `example-ast-gen.tar`, add just enough additional code to deal with the test case `001.tack`.
   http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/pr2/test.tar

6. Repeat the previous step for one test at a time, always adding just enough features for the next test. In this step, you should reuse your code from pr1, or if you prefer, you can also use the example solutions from pr1. The "one test at a time" approach helps maximize your partial credit in case you run out of time.
   http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/pr1-example-solutions.tar

7. Check the Tack language specification for any cases you haven't yet covered. For each such case, write a test, and then implement the features for it to succeed.
   http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/tack-spec.pdf

If you have difficulties with the project, do not wait until the last moment to ask questions. Instead, you should make use of your fellow students, the class mailing list, or the instructor's and grader's office hours. To give you a feeling for the coding effort, here are the number of lines of code in the example solutions:

Lines	File name	Description
60	`README`	Usage instructions
25	`Main.java`	Driver
471	`AstNode.java`	All the AST node classes
216	`Tack.g4`	Grammar with semantic actions
86	`Visitor.java`	Superclass for tree traversal
308	`SyntaxTreePrinter.java`	AST printer
292	`TreeNormalizer.java`	AST normalizer

Node class	Attribute	Children	Superclass
Program		FunDef+	AstNode
FunDef		FunId FunType BlockStmt	AstNode
ArrayType		Type	Type
RecordType		FieldType*	Type
FieldType		FieldId Type	Type
PrimitiveType	name		Type
FunType		RecordType Type	Type
VarDef		VarId Expr	Stmt
AssignStmt		Expr Expr	Stmt
BlockStmt		Stmt*	Stmt
CallStmt		Expr	Stmt
ForStmt		VarId Expr BlockStmt	Stmt
IfStmt		Expr BlockStmt BlockStmt?	Stmt
ReturnStmt		Expr?	Stmt
WhileStmt		Expr BlockStmt	Stmt
InfixExpr	op	Expr Expr	Expr
PrefixExpr	op	Expr	Expr
CallExpr		Expr Expr*	Expr
CastExpr		Expr Type	Expr
FieldExpr		Expr FieldId	Expr
SubscriptExpr		Expr Expr	Expr
ParenExpr		Expr	Expr
FunId	name		Expr
VarId	name		Expr
FieldId	name		AstNode
ArrayLit		Expr*	Expr
RecordLit		FieldLit*	Expr
FieldLit		FieldId Expr	AstNode
BoolLit	value		Expr
IntLit	value		Expr
NullLit			Expr
StringLit	value		Expr

Table 1: Concrete AST node classes for Tack. All the superclasses are abstract: `AstNode` is the root of the hierarchy, and `Type`, `Stmt`, and `Expr` are abstract subclasses of `AstNode`.