CSC 345 Lab – Abstract Syntax Tree Code

Overview

In this lab you will write code to generate an abstract syntax tree (AST). You will need to define AST node classes. You will need to write a parser for an expression grammar. Instead of writing all the code for a scanner we will create a queue of tokens and use that to drive processing.

Grammar

You will be writing code for the following grammar:

Expr → Factor ExprEnd
ExprEnd → + Factor ExprEnd
ExprEnd → ε
Factor → id

The start symbol for the grammar is Expr.

AST Node Classes

Create the following AST node classes.

- Expr – Should be an abstract class. There should be one abstract method named show. Here is the abstract method:
  abstract void show();
- Id – Should inherit from Expr. It should have one variable of type String to store the id's name. Create a one-parameter constructor to initialize the member variable. Override the show method so that it prints the id name.
- Sum – Should inherit from Expr. It should have two variables of type Expr (one each for the left and right sides of the sum operation). Create a two-parameter constructor to initialize the member variables. Override the show method so that it calls show on its child nodes.

Parser Class

Create a class named Parser.

- Define an enum member named TOKEN that has the values ID, PLUS.
- Declare a Queue<TOKEN> member variable named program.
- Create a match method. Here is the method header:
  public boolean match(TOKEN expectedToken)
  Use the Queue.peek method to get the next token. Use Queue.remove method to consume the next token. Return true if it matched and false otherwise.

- Create methods to do recursive descent parsing on the expression grammar. Need methods for factor(), exprEnd(), and expr() (recursive descent parsing).
    - These methods should return Expr.
    - Use the Queue.peek method to get the next token.
    - When creating an Id or INTLITERAL node use the token name (just call toString() on the token to get the token name). Note: A normal scanner would get the string from the token buffer.
- Create a method named parse that takes a queue of TOKEN as a parameter. Here is the method header:
  public void parse(Queue<TOKEN> program)
    - Initialize the program member variable with the parameter.
    - Declare a local variable of type Expr to store the root of the AST returned by the start symbol method (see below).
    - Call the start symbol method to begin parsing (it will return the AST root as an Expr instance).
    - Call show() on the generated AST.

Main Class

In the main method you should do the following:

- Create and instance of Queue and populate it with TOKENs. The queue of tokens is functioning as the program in this lab. For example, add the following tokens to the queue: ID PLUS ID. Another example would be: ID PLUS ID PLUS ID.
- Create an instance of Parser passing in the queue.
- Call the Parser.parse method to parse the program.

Java Queue Hints

We will be using the Java Queue interface. Use the Java LinkedList class as the queue instance. Using the LinkedList class gives a first-in first-out (FIFO) queue instance.

- Here is code to create declare and create a new Queue instance (Parser.TOKEN is an enum that will defined further down):
  Queue<Parser.TOKEN> queue;
  queue = new LinkedList<>();
- Queue.add method – Add a token to the queue.
- Queue.remove method – Remove the next token from the queue (removes in FIFO order).
- Queue.peek method – Returns the next token in the queue without removing it. This method allows us to "peek" inside the queue to see what is there.