ECE251 Final Project, Part 1, Fall 2010

Patrick Lam

Due: Friday, November 19

Brief Overview

The big picture: you will create a source-to-source translator. This translator will read WIG, a domain-specific language for describing web services, and output CGI scripts in C. I will give you a skeleton and tips along the way.

The first step is to create a parser. This should build on the knowledge that you gained doing Lab 1. I'll provide the Abstract Syntax Tree code, so you just need to modify the grammar file as necessary. In particular, you need to:

- add token definitions;
- stratify the expression grammar;
- fix the dangling-else ambiguity;
- add code to create an AST using the AST classes I provide.

There are some parts of ANTLR that you may not have explored in Lab 1, so I'll be providing more information on how to use them in the next few days.

You can find just the grammar at `http://ece251.patricklam.ca/svn/plam/skel/Wig.g` and the skeleton at `http://patricklam.ca/plt/files/project-skeleton-vN.tar.gz`. This time, I'm also going to make the Subversion repository for the skeleton available at `http://ece251.patricklam.ca/svn/plam/skel`. So, if you can figure out how to use `svn merge`, you can pull the differences as I publish them.

The skeleton contains a `ca.uwaterloo.ece251.WIG` main class which calls your parser to read a WIG input, and produces a reformatted version of that input on standard output.

We'll mark part 1 of the project using test cases. We will publish most of the test cases as well as expected outputs. I haven't yet decided whether or not to use hidden test cases.

Populating a List in ANTLR

I used the Kleene star * a lot in my grammar. I then needed to get lists. Here's one way of doing it. It may not be optimal, but it does work. So let's say that you have a production:

```
s : r*;
```

and the AST contains a constructor `S(List<r> rs)`. You need to create the list `rs` somehow. First, create the list, then add to it while parsing the `r`'s.

```
s returns [S s]:
 @init {
   sl = new LinkedList<R>();
 }
   (rr=r { sl.add($rr.v); })*
      { $s = new S(sl); };
```

I hope that makes sense! You'll find another example in the provided `Wig.g` file at the `inputs` production. There's also a way to share lists between different rules, but you shouldn't need to do that.

The Task

Here is some more detail on what you have to do.

Token Definitions. You need to put in the proper regular expressions for the tokens `ID`, `INT_LITERAL`, `STRING_LITERAL`, and `WS`. This should take 30 minutes max.

Stratify the expression grammar. The grammar in your `Wig.g` contains a single production for `exp`. This is ambiguous: ANTLR doesn't know, for instance, that it should give * higher precedence than +. Stratifying the grammar gets rid of the ambiguity and encodes precedence. See Lecture 9 (p3) for an example of a stratified grammar. This might take a bit longer for you to figure out what's going on, but not more than 1 hour.

Dangling-else. The grammar I've given you has the classic dangling-else ambiguity. Search in your notes for information on the proper incantation to get ANTLR to disambiguate the else properly.

Create ASTs. Insert parser actions to create the AST classes (i.e. those in `ca.uwaterloo.ece251.ast`) corresponding to the parse tree. I've provided a couple of examples in the grammar, for instance for one of the `exp` alternatives and for the `schema` production. Fill in parser actions for all of the rest of the productions in the grammar. Then the `WIG` driver will produce the right abstract syntax tree and print it out. This part is a bit tedious, but shouldn't take more than 3 hours.

Summary. For part 1, all of these tasks require only changes to `Wig.g`, so that file is all you need to submit. We will compile it with the skeleton and run it on our test cases. I'm not too worried about you over-fitting your solution to the test cases here, since the AST has to work for the second part of the project.