Milestone #1: Front-end

Overview

This milestone serves to introduce you to the language for which you will build a compiler, to refresh your memory of working with language representations, and to introduce some static analysis. More specifically, this milestone requires that you complete the front-end for the compiler that you will be building throughout the quarter. To complete this milestone you must implement the static semantic analysis phase.

Input & Abstract Syntax Tree

A partially complete front-end for your compiler is available on the course website. This front-end parses an input source file, builds a parse tree, and converts the parse tree into an abstract syntax tree that represents the input program.

Note: To support student interest in different languages, the provided front-end has been structured such that you can use whichever language you prefer in the implementation of your compiler. Specifically, the provided front-end will parse the input file and build a parse tree using `ANTLR`. The provided front-end can then generate and print the abstract syntax tree in JSON format or generate the abstract syntax tree as a collection of Java objects.

You have a choice to make, based on the provided code, for this milestone.

1. **Java:** If you choose to implement your compiler in Java, then you can use the in-memory Java object representation of the abstract syntax tree (see the `ast` directory) to perform the static semantic analysis. You are free to modify the provided files however you wish; you may replace them entirely if you want, they will not be offended.

2. **Not Java:** If you choose to implement your compiler in a language other than Java (or even if you want to use Java but want to create your own abstract syntax tree), then you can use the front-end to generate JSON. You can then write a program (in your language of choice) that will read the JSON representation, build an abstract syntax tree, and perform the static semantic analysis. (If your preferred language does not provide a JSON library, then this approach will take more effort, so budget your time accordingly.)

3. **ANTLR:** The more adventurous might opt to use ANTLR directly to generate code for their language of choice (ANTLR does support a few languages in addition to Java). If you wish, you can use the provided grammar file and then write a visitor to translate the parse tree into an abstract syntax tree (or the visitor can perform the static semantic analysis directly on the parse tree). This approach does require learning how to work with ANTLR, which is a valuable skill, but doing so would primarily provide exposure to the API and not the tool itself.

Static Semantics

After successfully parsing a program, your front-end must validate the program to ensure that it conforms to the static semantics of the language. Your validity checks must include those mentioned in the description of the language overview (linked from the course website).

Your program should report an error if an appropriate `main` function is not defined.

The check for a proper `return` requires a bit of bookkeeping. To get a feel for what a real compiler might do, your validity check on `return` must recognize each of the following as valid. There are, of course, additional patterns that one could match (such as those that Java does), but we have limited time (though you are free to extend your front-end as you see fit).

```
fun foo(int i) int         # standard case
{
   return i;
}

fun bar(int i) int         # control flow case
{
   if (i > 0)
   {
      return 1;
   }
   else
   {
      if (i < 0)
      {
         return -1;
      }
      else
      {
         return 0;
      }
   }
}
```

Removing any of the `return` statements above should result in an error.

Command-line Options

Throughout the term, you will extend your compiler to support a number of features. It will be desirable to allow the user to specify if a feature should be used or not. Keep this in mind when working with and extending the provided code.