Project 2: The MiniJava Parser

Due: Wednesday, February 7, 12:30 pm, by turn-in.

In this assignment you will extend the initial MiniJava parser and AST representation with the extensions described in the course project description handout.

You should extend MiniJava's syntax to allow the following (all of which are legal in full Java too):

- `double` is a legal (base) type.
- A floating-point literal constant is a legal expression.
- An array of a base type, e.g., `int[]`, `boolean[][][]`, and in general *`type`*`[]` where *`type`* is an arbitrary base type, is a legal (base) type. (Base types are ints, booleans, doubles, and arrays of base types. Only class types are not base types; this restriction is included only because otherwise the language becomes too hard to parse! The AST and the rest of the compiler should not depend on this restriction against arrays of class types, however.)
- A one-level array allocation expression, e.g., `new int[10]`, `new boolean[20][][]`, and in general `new` *`type`*`[`*`expr`*`]`*`dims`* where *`type`* is an arbitrary non-array base type, *`expr`* is an arbitrary expression, and *`dims`* is a possibly-empty sequence of `[]`'s, is a legal expression.
- An array dereference, e.g., *`a`*`[`*`i`*`]`, *`b`*`[`*`i`*`][`*`j`*`][`*`k`*`]`, and in general *`expr1`*`[`*`expr2`*`]` where *`expr1`* is an arbitrary atomic expression and *`expr2`* is an arbitrary expression, is a legal expression. An array dereference is also legal on the left-hand side of an assignment statement. (Atomic expressions EXclude unary and binary operator expressions and array allocation expressions.)
- An array length expression, e.g., *`a`*`.length` and in general *`expr`*`.length` where *`expr`* is an arbitrary atomic expression, is a legal expression. `length` is a reserved word in MiniJava (unlike Java).
- An or expression (using the `||` infix operator) is a legal expression.
- `if` statements do not require `else` clauses.
- For loops of the restricted form `for (`*`i`*` = `*`expr1`*`; `*`expr2`*`; `*`i`*` = `*`expr3`*`)` *`stmt`* are allowed, where *`expr1`*, *`expr2`*, and *`expr3`* are arbitrary expressions, *`i`* is an arbitrary variable (but which has to be the same variable in both the initialization and increment clauses, and *`stmt`* is an arbitrary statement.
- `break` statements are allowed. (You do not need to check in the syntax that break statements only appear inside of loops; semantic checking will enforce this.)
- A class variable declaration may be preceded by the `static` reserved word to declare a static class variable.

You should follow the precedence and associativity rules of regular Java for these extensions. It's OK to use CUP's `predecence` declarations to achieve this.

It's OK to have one shift/reduce conflict in your CUP grammar, for the "dangling else" problem. Add the "`-expect 1`" option before the `minijava.cup` argument in the `Makefile` to build `Parser/parser.java` if you decide to accept this shift/reduce conflict. (FYI, in making my sample solution, I couldn't find a way to revise the CUP grammar specification to avoid this conflict.)

You should add new AST classes and/or modify existing AST classes so that you can represent the new MiniJava constructs. You should define the appropriate `toString` operations on these classes so that they can be pretty-printed in a form that is syntactically legal and produces the same AST if it is parsed again. The other operations required of AST nodes, e.g. typechecking, evaluating, and lowering, you should implement by throwing `UnimplementedError` exceptions.

You only need to get the parser to work (and keep the extended scanner working). You do not need to do anything to enforce type checking rules or other semantic-analysis constraints on the input program.

Do the following:

1. Extend this specification of MiniJava's syntactic structure to describe the extended language, in the same style. (You can assume precedence and associativity is specified separately, and it is OK to define a grammar that is ambiguous with respect to the "dangling else" problem.)
2. Add and/or modify classes in the AST subdirectory to represent the extended language.
3. Extend `Parser/minijava.cup` to parse the extended language and construct the abstract syntax tree representing the parsed program.
4. Develop test cases that demonstrate that your extended parser and AST classes work, both in cases that should now be syntactically legal and in cases that should still be syntactically illegal. (Since the parser quits at the first error, you'll likely need several illegal test case files to test the different illegal cases.) You do not need to check for lexical errors, just syntactic errors. The `SamplePrograms` directory contains some files that should parse after you make your changes; some of the files should parse successfully with the initial version of the MiniJava compiler.

You can use the `-parse -printAST` options to the MiniJava compiler to just run the parsing phase and print out the AST that it builds. See the `test_parser` target in the `Makefile` for an example, and feel free to make your own target(s) to make running the tests you like easier and more mechanical.

Turn in the following:

1. Your extended MiniJava syntax specification.
2. Your modified `minijava.cup` file. Clearly identify your changes using comments.
3. Your new and/or modified `AST/*.java` files. Clearly identify any modifications using comments.
4. Your test cases, with names of the form *`name`*`.legal.java` for test cases that should parse successfully and *`name`*`.illegal.java` for test cases that should trigger syntax errors.
5. A transcript of running your parser and printing out the resulting AST on each of your test cases (at least).

Create a single directory containing these files, and submit them electronically by the due date.