Assignment 3: Lexing and Parsing BC#
Due: 11:59pm, Tuesday, February 15, 2011

1. Read the BC# Reference Manual. Most of the language is a subset of C#.

2. In your or your pair’s directory (see instructions in the previous assignments if you’re working with a new partner), use `svn copy` to create a private copy of the `src/a3` directory.

3. The files you are given are almost exactly the same as for the parsing part of the previous assignment, except that `Mintree.hs` has been renamed to `Absyn`. Your job is to extend these files so that the `ParseTest` program can parse any program in the BC# language. This will involve:
   - Updating `Lexer.x` to detect appropriate tokens;
   - Updating `Absyn.hs` to define suitable types that can represent the abstract syntax of BC# programs.¹
   - Updating `Parser.y` with suitable grammar rules for BC#, to produce appropriate abstract syntax trees.
   - Extending `Conflicts.txt` to explain why any remaining shift-reduce conflicts in your grammar description are “harmless.” (E.g., the dangling-else problem.)

Remember that your parser should not try to do typechecking, or to disambiguate overloaded notations (e.g., whether + is addition or string concatenation, or whether `s[n]` is getting an element from an array or a character from a string). Many constraints (such as that the name of the constructor be the same as the name of the class) might be easier to check later during type checking; just remember what these constraints are.

4. For each student (even if you’re working in a pair), add at least three non-trivial test inputs in `cs132s11/tests/goodparse` that should parse, and at least three test inputs in `cs132s11/tests/badparse`. Name these files using your HMC CS username and a strictly positive integer, e.g., `stone1.bcs`, `stone2.bcs`, .... (Please start counting from 1 in each directory.)

¹Recall: for abstract syntax, we don’t care about syntactic details like whether classes end with a semicolon or not, whether integers were written in decimal or hexadecimal, where the grouping-parentheses occurred in arithmetic expressions, and whether expressions were terms.
Hints

- Do not try to sit down and write everything, and then try to get the grammar to compile. Build very small lexers/parsers, get them to compile, test them, and once you're happy, move on by adding one more feature to the grammar. Trying to do everything at once will lead to a world of hurt.

For example, you might start with a parser that expects a program to be a single expression, where an expression can only be a boolean constant or an identifier. Add just these tokens and grammar rules to the parser, extend the lexer to generate these tokens, and try some test cases. If everything works, you can add string and boolean constants. If this works, add prefix unary operators, and so on.