CS 132: Compiler Design, Fall 2007

Assignment 3
Lexing and Parsing BC#

Due: 11:59pm, Wednesday, October 3

E-mailed questions about this assignment should be sent to `cs132help@cs.hmc.edu`.

1. Use `svn copy` to get a copy of the `src/a3` directory. You can compile the code in SML/NJ `CM.make "sources.cm"; .`

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

3. Extend the files `bcs.grm` to construct a parser for BC#, producing absyn as in `absyn.sml`.

   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 along with type checking; just remember what these constraints are.

4. If a parser is written and successfully runs through `ml-antlr`, it will generate code for the `BCSTokens` structure that you'll need for the `ml-ulex` lexer. Complete the lexer in `bcs.lex`.

5. Once everything compiles, you be able to test test your code by running

   ```
   Toplevel.parse "mytest.bcs";
   ```

   where `mytest.bcs` is the name the file with your test input.

6. A number of (non-exhaustive) test files have been provided in the `src/bcstest` directory.

Alternative

`ml-yacc` is simpler than `ml-antlr` for dealing with expression precedence and associativity, but `ml-antlr` has other advantages, including automated handling of line numbers; similarly `ml-ulex` is more flexible than `ml-lex`.

The skeleton files provided assume you will be using `ml-ulex` and `ml-antlr`. If you absolutely must use an LR-style parser generator, you can replace the lexer file and grammar file, and modify the `.cm` file appropriately. I wouldn't recommend it, but it's your choice.

Submission

Along with your lexer and parser and a `Design.txt` file containing any important documentation, you should submit a final `mytest.bcs` that exercises as much of the language as possible. I will be interested to see whether your test program breaks anyone else's parsers...

Hints

- Do *not* try to sit down and write everything, and then try to compile and tests. Build very small 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 an integer 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.

- I used `ml-antlr` backtracking (`%try`) to handle the `if-then`/`if-then-else` ambiguity.

- Be very careful to check whether `CM.make` returns `true` (compilation succeeded) or `false` (compilation failed). Sometimes compilation fails but the actual error message is several lines higher.