CS 321 Homework 3 – due 1:30pm, Thursday, March 1, 2012

Parsing

Write a parser for the complete **fab** language. The defining grammar for **fab** is in Section 13 of the Language Reference Manual; a copy is also available on the web page in the file `concrete.txt`. Use this grammar as a guideline for writing your parser.

Your parser must be implemented using the `CUP` parser generator to produce a `Parser` class. The generated parser will use the lexical analyzer from homework 2 (either your own or the reference version provided) to obtain a sequence of `Symbol` objects representing the tokens of a supposed **fab** program. If the token stream represents a syntactically legal program, your parser should generate the corresponding abstract syntax tree; otherwise, it should throw an appropriate instance of the `ParseError` exception. The provided file `ParserDriver` illustrates how the `Parser` class can be used (and how it will be tested for grading this homework).

The provided file `Ast.java` defines classes for representing the various kinds of nodes in abstract syntax trees for **fab** programs. (These classes are defined as inner classes of class `Ast`; this is just a convenient way to organize a large number of related classes.) You must build appropriate trees of AST nodes for all syntactically legal **fab** programs. More precisely, your parser should produce exactly the same AST as the reference parser provided in files `Parser.class`, `CUP$Parser$actions.class`, and `SymKinds.class`; more details about this are given below. The `toString()` methods defined on all AST node classes can be used to obtain a readable, printable representation of the AST in a standardized format, suitable for making comparisons between parser implementations. For syntactically invalid **fab** programs, your parser must raise an exception on the first syntax error discovered; it should not attempt error recovery. The text associated with your parser's exceptions need not match the reference parser exactly.

The "correct" form of the parser's output, i.e., the correct mapping from concrete to abstract syntax, is defined by the behavior of the reference parser. In most cases, this behavior should be obvious; here are a few noteworthy points:

1. The AST is capable of describing programs that are not type-correct; type-checking will be done in a later homework.

2. To help make error messages from such a type-checker meaningful, each AST node contains a `line` field; this should be the source line number associated with the construct. For constructs spanning several lines, the line number containing the *first* token should be used.

3. A `null` object is permitted in only four places in the AST: in the `super_name` field of a `RecordTypeDec` (when no `extends` clause is given), in the `typeExp` field of a `ConstDec` or `VarDec` (when no type is specified), and in the `returnValue` field of a `ReturnSt` (for `RETURN` statements that do not return a value).

4. Expand `elsif` clauses into nested `IfSt` structures in the AST. If the `else` branch is missing from an `if`, use a `BlockSt` containing a `Block` with an empty statement list.

5. The grammar for statements is ambiguous because of possible "dangling" `else` clauses; the correct disambiguation is described in the Language Reference Manual, Section 12.6.
6. If the result type of a function is omitted, supply `NamedType(0,"unit")` in the AST.
7. If the `by` clause in a `for` statement is omitted, supply 1 in the AST.
8. If the count expression is omitted in an array initializer, supply 1 in the AST.
9. The correct precedence and associativity for operators is specified in the Language Reference Manual, Section 11.8.
10. Uses of variables and constants are both parsed as `VarLvalue`s.

Your error messages need not match the reference version exactly, but at a minimum they should indicate the nature of the error and reflect the approximate source line number at which the error occurred.

Implementation and Program Submission

You must use the `CUP` parser generator to implement your parser. `CUP` can be downloaded from `http://www2.cs.tum.edu/projects/cup/`; you want the JAR file labeled `CUP 11a beta 20060608`. The remainder of these instructions assume that `java-cup-11a.jar` is in your current working directory.

To run CUP in conjunction with the `Yylex` class we developed in homework 2, compile your `.cup` specification file as follows (assuming a unix-like shell environment where `\` is used to continue a long line):

```
java -classpath java-cup-11a.jar java_cup.Main  \
        -parser Parser -symbols SymKinds -interface < fab.cup
```

This will produce files `Parser.java` and `SymKinds.java`; consult the `CUP` documentation for details. Use the newly provided version of `Symbol.java`, which is designed to work with CUP, instead of the version provided in homework 2. Your CUP specification must define the same terminals (symbol kinds) as were used in homework 2; this is already done for you in `fab0.cup`, which also shows how to specify a small subset of the **fab** grammar. (Note that running `fab0.cup` through CUP will produce 39 warning messages about declared but unused terminals; these are to be expected, since the part of the grammar that uses these terminals has not been filled in yet!) It is permitted to include code from `fab0.cup` in your submitted solution.

Your parser *must* generate an AST structure using the constructors defined in `Ast.java`. Note that for each node type that takes a sequence of children, there is a variant constructor that allows these children to be specified as a `List`, rather than as an array; this simplifies parsing, where the length of the seqeunce is not known ahead of time.

To compile the CUP-generated parser code in the context of the test driver, make sure that a working solution to homework 2 (either yours or the reference solution) is in the current directory in file `Yylex.class` and type:

```
javac -classpath .:java-cup-11a.jar Parser.java \
     ParserDriver.java Scanner.java Ast.java \
     ParseError.java Symbol.java
```

To test the resulting program on a **fab** file `foo.fab`, type:

```
java -classpath .:java-cup-11a.jar ParserDriver < foo.fab
```

Note: To generate the reference parser for comparison purposes, use the same `javac` command line as above, but omit `Parser.java`, instead making sure that the provided reference copies of `Parser.class`, `CUP$Parser$actions.class`, and `SymKinds.class` are in the current directory.

You should submit a single file `fab.cup` containing your `CUP` specification. (Remember, if you need to define any additional auxiliary classes, you can put them at the top of your `CUP` file.) We will process your CUP file using CUP option settings specified above, producing `Parser.java` and `SymKinds.java`. These will be combined with the provided files.

Your file should be submitted as a plain text attachment to a mail message sent to `cs321-03@cecs.pdx.edu` with subject line prefix "`[hw3]`". Be sure to include your name in a comment within your submitted code file. Your code must work correctly with the provided `ParserDriver`, `Scanner`, `Ast`, `ParseError`, and `Symbol` classes, and with the reference version of `Yylex.class` from homework 2. You may *not* modify these classes, and you should not submit any code for them. We will process your submission by creating a fresh directory, copy in `java-cup-11a.jar`, the provided `ParserDriver.java`, `Scanner.java`, `Ast.java`, `ParseError.java`, `Symbol.java`, and `Yylex.class` files, and saving your attachment. We will then process your `.cup` file, compile the resulting Java code, and run the parser driver using the incantations given above. Note that we will be using automated mechanisms to read, compile, and test your programs, so adherence to this naming and mailing policy is important! As usual, you may lose points if you fail to submit your program in the correct way.