

## CS 321 Homework 3 – due 1:30pm, Wednesday, November 17, 2004

This homework specification is copyright 2004 by Andrew Tolmach. All rights reserved.

### Parsing

Write a parser for the complete **PCAT** language. The defining grammar for **PCAT** is in Section 12 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 as a class called `Parser`, in the default (anonymous) package, which defines a constructor

```
Parser(Yylex y)
```

and a method

```
Ast.Program parse() throws ParseError
```

where `Yylex` and `ParseError` are exactly as defined in assignment 2, and `Ast.Program` is defined in the provided file `Ast.java`. Recall that class `Yylex` contains a method `yylex()` that returns an object of class `Symbol`. Your `parse()` method should use `yylex()` to obtain a sequence of `Symbol` objects representing the tokens of a supposed PCAT program. If the token stream represents a syntactically legal program, `parse()` 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 assignment).

The provided file `Ast.java` defines classes for representing the various kinds of nodes in abstract syntax trees for PCAT programs. (These classes are defined as inner classes of class `Ast`; this is just a convenient way to define a large number of classes in a single file.) You must build appropriate trees of AST nodes for all syntactically legal PCAT 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 PCAT 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 assignment.

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 three places in the AST: in the `typeName` field of a `VarDecs` (when no type is specified), in the `resultType` field of a `ProcDec` (for proper procedures, which do not return a value), and in the `returnValue` field of a `ReturnSt` (for `RETURN` statements in proper procedures).
4. The predefined constants (`TRUE`, `FALSE`, and `NIL`) should be parsed as variables.
5. The `fp-section` construct is not represented in the AST; it is turned into a list of `FormalParam` structures, which is concatenated with the lists for any adjacent `fp-sections` to form a single list.
6. Lists of statements are represented using the `SequenceSt` subclass of `St`, i.e., within the AST, single statements and lists of statements are handled uniformly. A `SequenceSt` may have zero elements (representing “do nothing”). It may also have just one element, in which case it could just be replaced by that element (but the reference parser doesn’t do so).
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. Expand `ELSIF` clauses into nested `IfSt` structures in the AST. If the `ELSE` branch is missing from an `IF`, use an empty `SeqSt` in the AST.
10. The correct precedence and associativity for operators is specified in the Language Reference Manual, Section 10.8.

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

The internal details of your parser are up to you, but it is *strongly* recommended that you implement it using the CUP parser generator. CUP can be downloaded from <http://www.cs.princeton.edu/~appel/modern/java/CUP/>. For your convenience, it has also been installed on the CS Solaris systems at `/u/cs321-01/CUP`.

To run CUP in conjunction with the `Yylex` class we developed in assignment 2, use the special directives illustrated in the provided file `pcat0.cup`, and compile your cup specification file as follows:

```
java -classpath ./u/cs321-01/CUP java_cup.Main \
    -parser Parser -symbols SymKinds -interface < pcat.cup
```

(If you have downloaded CUP yourself, substitute the directory name where you have put it in place of `/u/cs321-01/CUP`.) Use the provided special CUP-targeted version of `Symbol.java` instead of the version provided in assignment 2. Your CUP specification must define the same symbol kinds as were used in assignment 2; this is already done for you in `pcat0.cup`, which also shows how to specify a small subset of the PCAT grammar. It is permitted to include code from `pcat0.cup` in your submitted solution.

Whether or not you use CUP, 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 sequence is not known ahead of time.

If you choose to use CUP, you should submit a single file `pcat.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, using the CUP-targeted version of `Symbol.java`.

If you choose not to use CUP, you should submit a single file `Parser.java` defining class `Parser` directly. We will combine this with the provided files, using the assignment 2 version of `Symbol.java`. If you take this option, you must write the Java code yourself; you may *not* use any parser generator other than CUP.

In either case, your file should be submitted as a plain text attachment to a mail message sent to `cs321-01@cs.pdx.edu`. The subject line of your mail should include your name and the string “HW3”. Your code must work correctly with the provided `ParserDriver`, `Ast`, `ParseError` classes, with either the assignment 2 version or the CUP-targeted version of the `Symbol` class, and with the reference version of `Yylex.class` from assignment 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 the provided `ParserDriver.java`, `Ast.java`, `ParseError.java` files, whichever version of `Symbol.java` is appropriate, and `Yylex.class`, and saving your attachment. If you submit a `.cup` file, we will execute

```
java -classpath ./u/cs321-01/CUP java_cup.Main \  
    -parser Parser -symbols SymKinds -interface < pcat.cup  
javac -classpath ./u/cs321-01/CUP Parser.java \  
    ParserDriver.java Ast.java ParseError.java Symbol.java
```

If you submit a `.java` file, we will execute

```
javac Parser.java ParserDriver.java Ast.java \  
    ParseError.java Symbol.java
```

To test the resulting program on a PCAT file `foo.pcat`, we should be able to type

```
java -classpath ./u/cs321-01/CUP ParserDriver < foo.pcat
```

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.