C311 – Program 1: Syntax, Parsing, Abstract Syntax

Corky Cartwright

Produced: August 18, 2004— Due: 12:00 noon, Wednesday, September 8, 2004

Overview

Preparation: We suggest that you run the OwlNet `register` command for `comp311`, which will make the latest versions of drscheme and drjava available as commands in your OwlNet environment.

For more information about DrScheme, read the documentation in

`http://www.drscheme.org/tour/` http://www.drscheme.org/tour/

and the DrScheme Help menu. You can download DrScheme Version 205 from `drscheme.org`.http://drscheme.org DrScheme supports a variety of different Scheme dialects. For this class, please use the ”Pretty Big” dialect of Scheme under Professional Languages.

DrJava is distributed as a Java .jar file at the web site `drjava.org`. http://drjava.org You can run DrJava on a Unix system (with Java JDK 1.4 installed) by typing

```
java -jar drjava.jar
```

assuming that `drjava.jar` is located in the current directory. On Windows machines (with Java JDK 1.4 installed), you can run the `drjava.jar` file simply by clicking on the icon. Documentation for DrJava is available at `drjava.info`. http://drjava.org In addition, if you have questions about DrJava, send email to

`javaplt@rice.edu`. mailto:javaplt@rice.edu

Documentation on the JSR14 extension of Java is available at `www.cis.unisa.edu.au/~pizza/gj/`

We strongly recommend following the *documentation* link and reading the *GJ Tutorial.* The download site for JSR14 is

`developer.java.sun.com.`

This site requires registration. Follow the link listed under the heading ¡B¿Early Access Downloads¡/B¿.

A good article on the use of Generic Java in conjunction with unit testing is available the IBM DeveloperWorks website:

`www-106.ibm.com/developerworks/java/library/j-diag0625.html?dwzone=java`

Every student should create a `comp311` directory in his or her home directory. Each programming team should submit their solution under the userid one team member. We recommend creating a a `programs/1` directory within the directory `comp311`. All of your files for this assignment should be stored in the `programs/1` subdirectory.

TeamWork: You are strongly encouraged to do this assignment and all other assignments in this course in teams of two. When you submit your assignment, indicate the composition of your team (names, student id's, and email addresses) and to what extent you followed the pair-programming model in your `README` file.

If you cannot find a partner and would like one, send James Sasitorn (camus@rice.edu) email and he will try to find one for you. Teams of more than two members are *not* permitted.

Testing: You are expected to write unit tests for all of your non-trivial methods or functions, depending on whether you write your assignments in Java or Scheme. DrJava provides integrated support for JUnit which makes this task easy. DrScheme Version 205 supports a similar unit testing system for Scheme called (surprise!) SchemeUnit. It is available for download from

`http://sourceforge.net/projects/schematics/`

under the name `schemeunit` in the list of file releases. The downloadable file is a "plt" file that you open using DrScheme. When you open it, DrScheme adds the library to the collection of libraries maintained by DrScheme. Documentation for SchemeUnit is available at `http://schematics.sourceforge.net/schemeunit/schemeunit.html`

Chapter 2 entitled "Quick Start" (a single page!) provides enough information to write unit tests for this assignment.

Note that unit testing conflicts in some cases with the name hiding recommendations given for writing Scheme code in Comp 210 because lexically nested function definitions cannot be tested. In your Scheme programs all non-trivial functions must be accessible at the top-level.

Task: Your task is to write a parser in Generic Java or Scheme that recognizes expressions in the illustrative language Jam, which will be defined subsequently. The course staff has provided lexers in both Scheme and Java for this assignment. These lexers return a larger set of symbols than what appears in the Jam definition. These extra symbols anticipate future extensions to Jam. At this point, your parser should simply reject any program containing one of these extra symbols as ill-formed, just like it would for any other syntax error.

Scheme Version:

In Scheme, the parser's input language is a stream of tokens implemented as a parameterless "read" procedure `p` that encodes a token stream. Given the procedure `p`, the application `(p)` returns the next token in the encoded stream, which is either a Scheme symbol, a Scheme integer, or a special end-of-file object that is recognized by the predicate `eof-object?`.

The Scheme library file `lexer.ss` implements a scanner (lexical analyzer) that converts either a file or a string to a corresponding token stream. Given a string `s` specifying a file name, the application `(make-file-lexer s)` returns the corresponding token stream. Given a string `s` consisting of program text, the application `(make-string-lexer s)` returns the corresponding token stream.

The parser's output language is Jam abstract syntax, which is defined in the library file `abs.ss`. If the parser determines that some given input does *not* represent a legal Jam program, it should print an appropriate error message and terminate the parsing process. In Java, you can abort the computation by throwing a `ParseException` for which there is no `catch` handler.

Note that `lexer.ss` and `abs.ss` are *Teachpack* (library) files and should be loaded into DrScheme using the `AddTeachpack` command on the `Language` menu. You can load both libraries into DrScheme as a single TeachPack library by placing both library files in your program directory and designating the file `lexer.ss` as the TeachPack to be loaded. The module `lexer.ss` requires the module `abs.ss`, which forces it to be loaded. These library files are available on the web in the directory

`www.cs.rice.edu/\~javaplt/311/Assignments/1/scheme.`

Your parser should be expressed as a function `parse` that takes a string specifying a file name as its only argument and returns an abstract syntax tree for the Jam expression in the specified file. Your Scheme program should be stored in the file `parser.ss`, which will be recognized by the `turnin311` command and our grading program.

Java (GJ) Version:

In Java, the parser's input token stream is provided by a `Lexer` object supporting the parameterless method `readToken()`. The `readToken()` method produces a stream of `Token` objects belonging to various subclasses of the `Token` interface. The `Lexer` class supports two constructors: a zero-ary constructor that converts standard input to a `Lexer` and a unary constructor `Lexer(String fileName)` that converts the specified file to a `Lexer`. The file `lexer.java` contains the definition of the Lexer class and the collection of classes representing tokens (the Java interface `Token`) and abstract syntax trees (the java interface `AST`).

Your parser should be expressed as a class `Parser` with constructors `Parser(Reader stream)` and `Parser(String filename)` which create parsers reading the specified stream or file. The provided `Lexer` class includes two similar constructors `Lexer(Reader stream)` and `Lexer(String filename)`. The `Reader` form of these constructors is very convenient when you are testing your parser from the DrJava read-eval-print loop or from unit test methods.

The `Parser` class should contain a `public` method `AST parse()` that returns the abstract syntax tree for the Jam expression in the input stream. Our grading program will print the output using the method `System.out.println`, which invokes the `toString` method on its argument. All of the requisite abstract syntax classes including `toString` methods are defined in the file `lexer.java`. This library files are available on the web in the directory

`www.cs.rice.edu/~javaplt/311/Assignments/1/java.`

If your parser encounters an erroneous input, simply print an explanation of the syntax error and abort parsing the file (using the `error` primitive

Scheme and the throwing of a `ParseException` in Java. Recovering from a syntax error to continue parsing is a complex problem with many special cases. It is not a realistic goal for this assignment.

Choice of Language: Generic Java and Scheme are the only languages permitted for this assignment.

Input Language Specification:

The following paragraphs define the parser's input language and the subset that your parser is supposed to recognize a legal `Jam` program.

The Input Language The parser's input language is `<input>` where:

```
%% A phrase P enclosed in braces { P } is optional
%% A phrase P followed by a * is iterated zero or more times, e.g. <def>+ means
%%   the concatenation of zero or more <def> phrases
%% A phrase P followed by a + is iterated one or more times, e.g. <def>+ means
%%   the concatenation of one or more <def> phrases
%% If a phrase P enclosed in brackets { P } is followed by a * or +, the braces are
%%   simply denote grouping: the enclosed phrase is iterated as specified by the
%%   * or + symbol

  <input> ::= <token>*

  <token> ::= <alpha/other> <alpha/other/numeric>* | <delimiter> | <operator>

%% Adjacent tokens must be separated by whitespace (a non-empty sequence of
%% spaces, tabs, and newlines) unless one of the tokens is a delimiter or
%% operator.
%% In identifying operators, the lexer chooses the longest possible match.
%% Hence, "<=" is %% interpreted as a single token rather than "<" followed by "="

  <alpha/other/numeric> ::=  <alpha/other> | <digit>

  <alpha/other> ::= <lower> | <upper> | <other>

  <lower> ::= a | b | c | d | ... | z

  <upper> ::= A | B | C | D | ... | Z

  <other>  ::= ? | _
  <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  <delimiter> ::= ( | ) | [ | ] | , | ; |
  %% In subsequent assignments, "{" and "}" will be added to this list

  <operator> ::= "+" | - | ~ | "*" | / | = | != | < | > | <= | >= | & |
                 "|" | :=

%% The operators "ref", "<-",  and "!" will be added in subsequent assignments,
%% but they do not exist in Assignment 1
```

Jam The set of legitimate Jam phrases is the subset of the potential inputs consisting the expressions `<exp>` defined by the following grammar:

```
%% Expressions:
  <exp> ::= <term> { <binop> <exp> }
  | if <exp> then <exp> else <exp>
  | let <def>+ in <exp>
  | map <id-list> to <exp>

  <term> ::= <unop> <term>
   | <factor> { ( <exp-list> ) }
   | <null>
   | <int>
   | <bool>

  <factor> ::= ( <exp> )  |  <prim>  | <id>

  <exp-list>      ::= { <prop-exp-list> }
  <prop-exp-list> ::= <exp>   | <exp> , <prop-exp-list>
  <id-list>       ::= { <prop-id-list> }
  <prop-id-list>  ::= <id>    | <id> , <prop-id-list>

%% Definitions:
  <def> ::= <id> := <exp> ;

%% Primitive Constants, Operators, and Operations:
  <null>  ::= null
  <bool>  ::= true | false

  <unop>  ::= <sign> | ~
  <sign>  ::= "+" | -
  <binop> ::= <sign> | "*" | / | = | != | < | > | <= | >= | & | "|"
  <prim>  ::= number? | function? | list? | null? | cons? | cons
            | first | rest | arity
  %% In subsequent assignments ref? will be added to <prim>

%% Identifiers:
  <id>  ::= <alpha/other> {<alpha/other> | <digit>}*
            %% except for <prim> and the keywords if, then, else,
            %%   map, to, let, in, null, true, false
            %% In subsequent assignments ref will be added to this list

  <alpha/other> ::= <alpha> | <other>  %% <other> as defined above
  <alpha> ::= <upper> | <lower>        %% <upper> and <lower> as defined above


%% Numbers:
   <int>  ::= <digit>+
```

The lexer recognizes keywords, delimiters (parentheses, commas, semicolons), primitive operations, identifiers, and numbers.

The preceding grammar requires one symbol lookahead in a few situations. The Scheme lexer in the file `lexer.ss` supports a peek operation precisely for this purpose. Given a token stream `p`, an application of the form `(p` *x*`)` where *x* is any argument (*e.g.*, the symbol `'peek`) returns a copy of the next token in `p` without removing it from (or otherwise changing) `p`.

In the Java lexer, the method `Token peek()` behaves exactly like the method `Token readToken()` except for the fact that it leaves the the scanned token at the front of the input stream (in contrast `readToken()` removes the scanned token from the input stream).

Abstract Syntax Trees As mentioned above, the Scheme library `lexer.ss` defines the set of constructors used to build abstract syntax trees in Scheme. There is one constructor for each different form of expression in the definition of Jam syntax given above. In addition, there is one abstract syntax form for Jam definitions, which can easily express both concrete forms given above.

For example, suppose the Jam program under consideration is

```
f(x) + (x * 12))
```

The abstract syntax representation for this program phrase is:

```
(make-biop-exp '+
  (make-app-exp (make-id-exp 'f) (list (make-id-exp 'x)))
  (make-biop-exp '* (make-id-exp 'x) (make-num-exp 12)))
```

in the context of the following data descriptor definitions:

```
(define-struct biop-exp (rator rand1 rand2))
(define-struct app-exp (rator l-of-rands))
(define-struct id-exp (arg))
(define-struct num-exp (arg))
```

The Java file `lexer.java` defines all of the abstract syntax classes required to represent Jam programs. There is one constructor for each different form of expression in the definition of `Jam` syntax given above. Some primitive (non-recursive) abstract syntax classes are the same the corresponding token classes. There is one abstract syntax form for `Jam` definitions, which can easily express both concrete forms given above.

Testing and Submitting Your Program

The file

`www.cs.rice.edu/~javaplt/311/Assignments/1/java.`

contains a sample input program. Create a README file in the your directory `program/1` that

- gives the names and userids of *both* team members,
- outlines the organization of your program, and
- specifies what testing process you used to confirm the correctness of your program.

Your test data files should be stored in the `programs/1` directory.

Each procedure or method in your program should include a short comment stating precisely what it does. For routines that parse a particular form of program phrase, give a grammar rule(s) describing that form.

To submit your program, make sure that everything that you want to submit is located in the directory `programs/1` and type the command `turnin311 1`. The command will inform you whether or not your submission succeeded. *Only submit one copy of your program per team. If you need to resubmit an improved version your program, submit it from the same account as the original so that the old version is replaced.*

Implementation Hints Use an “unparser” to print a concrete representation for an abstract syntax tree. Then you can directly compare test input strings and output strings (up to differences in whitespace and parentheses using for grouping).