Programming Assignment 1:
Parsing and Abstract Syntax

Points: 100

Overview

Preparation

We recommend using either IntelliJ or DrJava as your Java IDE for all programming assignments. You can download DrJava in the form of the executable jar file `drjava.jar` from http://www.cs.rice.edu/~javaplt/drjavarice. When you accept the assignment, GitHub Classroom creates a repository for you that in initialized to contain all of the support files for the project including this project description. DrJava only runs using Java 8. We recommend using the Oracle Java 8 OpenJDK with DrJava. OpenJDK 8 works well on some machines but does not appear to be as solid an implementation on Windows as the Oracle Java 8 OpenJDK. If you use IntelliJ as your IDE, you can choose any compatible JDK, but note that your programs will be tested using Java 8. Beware of the fact that the assignments you submit must run from the command line and *cannot* use named packages because our test programs do not prefix any class names with package names. Edit the `README` file for your repository to give an overview of your solution, most notably the API that it supports. We recommend that you place your Java source files in the folder named `src` that is already present in your repository. We also recommend placing your compiled class files in a separate folder named `classes` that is already present in your repository. In DrJava, you can define a project with `src` as the source directory and `classes` as the folder where the compiler places the `.class` files corresponding to the source program. Like more elaborate professional IDEs, DrJava supports compiling and testing projects.

DrJava is distributed as a Java `jar` file at the web site http://drjava.org as well our local site (http://www.cs.rice.edu/~javaplt/drjavarice) but the local site typically has more recent builds. You can run DrJava on a Linux or Mac system (with Java JDK 8 installed) by typing

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

assuming that `drjava.jar` is located in the current directory. DrJava is not a registered Mac application, so some recent versions of Mac OS X make running DrJava difficult. On Windows machines (with a properly installed Java JDK 8), you can run the `drjava.jar` file simply by double-clicking on the icon. If the Registry has the wrong file type associated the `.jar` files, then double-clicking `.jar` icons will not work. In this case, we recommend running DrJava inside the Windows "Command Prompt" or Powershell using the same command as shown above for Linux.

Documentation for DrJava is available at drjava.org. If you have further questions about DrJava, ask them on Piazza.

Testing You are expected to write unit tests for all of your non-trivial methods or functions *and submit your test code as part of your program*. DrJava provides integrated support for JUnit which makes this task easy. IntelliJ supports integrated testing using Junit as well.

Task Your task is to write a parser in Generic Java (Java 8+) that recognizes expressions in the pedagogic functional language Jam, which will be defined subsequently. The course staff has provided a lexer in Java for this assignment. This lexer returns a larger set of symbols than what appears in the Jam definition. These extra symbols anticipate future extensions to Jam in later projects. 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.

Java Support Library In the provided support files, the parser's input token stream is generated by a `Lexer` object supporting the 0-ary 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: (i) a zero-ary constructor that converts standard input to a `Lexer` and (ii) 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 type `Token`) and abstract syntax trees (the java interface type `AST`).

The API That Your Program Must Support

All classes must be in the default (top-level) package. Using named packages will break our compilation and grading scripts, causing a catastrophic loss of points on the assignment.

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 **`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 folder ../../Assignments/1/java.

If your parser encounters an erroneous input (including illegal tokens), simply print an explanation of the syntax error and abort parsing the file by throwing a **`ParseException`**. 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.

Input Language Specification

The following paragraphs define the parser's input language and the subset that your parser should recognize as legal Jam programs.

The Meta-Language The parser's input language is described using Extended Backus-Naur Form (EBNF). When reading the notation, keep the following in mind:

- A symbol in the meta-language always starts with a capital letter. Therefore, **`Symbol`** is a symbol in the-meta language, whereas **`literal`** is the actual text "literal". Note that this convention depends on the fact that no literals in Jam begin with a capital letter.
- A phrase ***P*** enclosed in braces **`{ P }`** is optional, *i.e.* both ***P*** and the empty string match **`{ P }`**.
- A phrase ***P*** followed by **`*`** is iterated zero or more times, *e.g.* **`Def*`** means the concatenation of zero or more **`Def`** phrases:

  `Def`
  `Def Def`
  `...`
  and so on, including the empty string.
- A phrase ***P*** followed by **`+`** is iterated one or more times, *e.g.* **`Def+`** means the concatenation of one or more **`Def`** phrases:
  `Def`
  `Def Def`
  `...`
  and so on, but **not** the empty string.
- 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. For example, **`{ A B }*`** is matched by zero or more concatenations of A

B: **`A B`**
`A B A B`
and so on, as well as the empty string, but **not** **`A`** or **`A B A`**.

- If something that is interpreted as part of the meta language should be interpreted literally as part of the language we are describing, it will be enclosed in *single* quotes. Therefore, **`'A'`** is the actual letter "A", not the symbol **`A`** in the meta language, and ***`P`*** **`'*'`** is the phrase ***`P`*** followed by an actual asterisk, and does not mean that the phase ***`P`*** should be repeated zero or more times, as described above.

The Input Language The parser's input language (modulo a few changes described in the section (**Left-associative Binary Operators**) is **`Input`** where:

```
Input ::= Token*
Token ::= AlphaOther AlphaOtherNumeric* | Delimiter | Operator | Int
```

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

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 **`=`**.

```
Lower             ::= a | b | c | d | ... | z
Upper             ::= 'A' | 'B' | 'C' | 'D' | ... | 'Z'
Other             ::= ? | _
Digit             ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Alpha             ::= Upper | Lower
AlphaOther        ::= Alpha | Other
AlphaOtherNumeric ::= AlphaOther | Digit
Delimiter         ::= ( | ) | [ | ] | , | ;
Operator          ::= '+' | - | ~ | '*' | / | = | != | < | > | <= | >=
| & | '|' | :=
```

Note: The terminal symbols 'A', **`'B'`**, ... are enclosed in single quotation marks because **`A`**, **`B`**, ..., would automatically be the names of non-terminal symbols in our notation system. For essentially the same reason, we enclose the symbols **`'+'`**, **`'*'`**, and **`'|'`** in single quotation marks; these symbols without quotation marks are metasymbols in the notation scheme described above.

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 IdList to Exp
Term        ::= Unop Term
              | Factor { ( ExpList ) }
              | Empty
              | Int
              | Bool
Factor      ::= ( Exp ) | Prim | Id
ExpList     ::= { PropExpList }
PropExpList ::= Exp { , Exp }*
IdList      ::= { PropIdList }
PropIdList  ::= Id { , Id}*
```

Definitions

```
Def ::= Id := Exp ;
```

Primitive Constants, Operators, and Operations

```
Empty ::= empty
Bool  ::= true | false
Unop  ::= Sign | ~
Sign  ::= '+' | -
Binop ::= Sign | '*' | / | = | != | < | > | <= | >= | & | '|'
Prim  ::= number? | function? | list? | empty? | cons? | cons | first
| rest | arity
```

Identifiers

```
Id ::= AlphaOther {AlphaOther | Digit}*
```

Please note that **`Id`** does **not** contain the anything that matches **`Prim`** or the keywords **`if`**, **`then`**, **`else`**, **`map`**, **`to`**, **`let`**, **`in`**, **`empty`**, **`true`**, and **`false`**.

Numbers

```
Int ::= Digit+
```

The preceding grammar requires one symbol lookahead in a few situations. The Java lexer in our support code includes a method **`Token peek()`** that 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).

The Java file **`lexer.java`** defines all of the abstract syntax classes required to represent Jam programs using the composite design pattern. There is one constructor for each fundamentally 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.

For example, suppose the Jam program under consideration is

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

The abstract syntax representation for this program phrase is:

```
new BinOpApp(
       new Op("+"),
       new App(new Variable("f"), new AST[] { new Variable("x") }),
       new BinOpApp(new Op("*"), new Variable("x"), new
IntConstant(12))
     )
```

This construction does not tell the whole story. The lexer is guaranteed to always return the *same* object for a given operator. Similarly, there is only copy of each variable encountered by the lexer. Hence, there is only one copy of the **`Operator`** "+", one copy of the **`Operator`** "*", one copy of the **`Variable`** "x", *etc*. As a result, the **`==`** operator in Java can be used to compare operators and variables. Note that the **`Operator`**s "+" and "-" can be used either as unary or binary operators.

Left-associative Binary Operators

The preceding grammar for Jam has an annoying flaw. To make the language easy to parse using recursive descent (top-down) parsing, we used right recursion in the definition of arithmetic expressions (**`Exp`**). (Left recursion forces infinite looping in top-down parsing.) But right recursion yields right-associative ASTs (where computation must be performed right to left), while the usual mathematical convention in arithmetic expressions is that binary operators are left-associative (where computation must be performed left to right). There is a widely-used trick in writing grammars for top-down parsing using syntax diagrams that avoids this problem. The trick is replace right recursion in a simple production (only one alternative on the right hand side) by iteration: an edge looping back to the input edge of the diagram. We used this trick in generating the syntax diagrams for **`ExpList`** and **`IdList`** for our original grammar. Given our original Jam grammar, we can convert the production

```
Exp         ::= Term { Binop Exp } | ...
```

to

```
Exp          ::= BinaryExp | ...
BinaryExp    ::= Term { Binop BinaryExp }
```

at the cost of excluding chains of binary operations terminating in an expression other than a term. Then this revised grammar can be represented using syntax diagrams that define **BinaryExp** iteratively (just like **PropExpList** in the original grammar).

The preceding definition of **BinaryExp** generates the same set of strings as the extended rule:

```
BinaryExp ::= Term { BinOp Term }*
```

which is ambiguous about left or right associativity. When iteration is used to recognize a **BinaryExp**, the parsing process will construct a left-associative tree since the tree is constructed left-to-right (each new **Binop** adds a new root to the AST). This convention make the syntax diagram formulation of our Jam grammar deterministic and unambiguous.

The file SyntaxDiagrams contains iterative syntax diagrams corresponding to the original left associative version of the Jam language, which precisely describes the syntax of Jam and how we want the parser to behave. Of course, this language eliminates a few quirky inputs from the language based on our original grammmar. Your parser should treat these quirky inputs as erroneous.

It is possible to write syntax diagrams that treat binary expressions iteratively and still allow arbitrary expressions at the end of the binary expression. But the corresponding grammar and syntax diagrams are much more complex. So we have not followed this design choice in creating the revised syntax for Jam. In fact, we view our Jam grammar as a case study showing that pure context-free grammars are not quite the right formalism for expressing easily parsed program syntax. Syntax diagrams where iteration is always left-associative (equivalent to extended context-free-grammars where iteration always expands into left-associative parse trees) are much better. I am not aware of a rigorous exposition of such extended BNF grammars in the literature; I suspect that most theoretical computer scientists would not find such an exposition very interesting even though the standard notion of parsing language specified by a context-free grammar is an imperfect formalization of practical parsing.

Write your parser to produce left-associative ASTs for arithmetic expressions. If your parser implements either of the "revised" syntax diagram definitions given above (which are equivalent) using **while** loops for iteration and constructs the corresponding abstract syntax trees in a functional style (no mutation of trees in theconstruction process), your parser will naturally build syntax trees that are left-associative.

Testing and Submitting Your Program

The `src` folder in the provided code base includes some Junit4 test classes which are not comprehensive. You need to add more test cases; you need to check every production in the grammar. If a production involves iteration or recursion, you need to test the base cases as well as simple inductive constructions.

Editr the **`README`** file for your repository so that it

- outlines the organization of your program,
- specifies any compatible additions that you made the provided API (not generally necessary), and
- specifes what testing process you used to confirm the correctness of your program.

Note that the README file is a GitHub “MarkDown” file, text augmented by a few simple formatting annotations. MarkDown (MD) is much less powerful than HTML but much easier to use. The webpage **markdown-cheat-sheet** is a good summary for learning MarkDown and a useful reference.

We do not recommend using separate test data files because it creates more opportunities for incidental errors when testing your solution. Unless you are testing how well your program scales to large inputs, they are generally unnecessary. All JUnit test classes submitted for grading should be compatible with a generic solution that conforms to the public interface provided by the stub files in the provided support code. In other words, we should be able to compile and run your unit tests against a reference solution used for grading, which has the same public interface as the support code. Your test classes should be defined in files matching the pattern **`"*Assign#Test.java"`**, where **`"#"`** is the current assignment number. For example, **`"MyAssign2Test.java"`** would be a valid name for Assignment 2 unit tests. If you want to include other tests that you do not want graded (e.g., to test your private interfaces), then simply use a different suffix than "Test" for the test class names (e.g., **`"Assign1Test#.java"`**). If you have additional utility classes that are required dependences of your unit test classes, please name those classes using the pattern **`"*Assign#TestSupport.java"`**, which signals that the class is required to support your unit tests, but is not a unit test class that should be independently launched. You may also include test support files with names matching the pattern **`"*AssignAllTestSupport.java"`**, which can contain common test support code that can be reused across multiple assignment submissions without needing to rename the file. Note that our class solutions to the assignments do not rely on such utility classes.

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, the comment should provide the grammar rules describing that form.

You obviously do not need to “submit” your program, since you are creating it inside a repository on the ownership of the course staff. We can easily access all student repositories.

Implementation Hints

The **`toString()`** methods for each **`AST`** class effectively provide an unparser for **`AST`**s. You can directly compare test input strings with output **`AST`**s by unparsing the **`AST`**s (using **`toString()`**). The two strings should match up to differences in whitespace, new lines, and parentheses using for grouping which may force you to slightly reformat your input strings.