CMSC 330 Project #1 Fall 2012

Date due: Wednesday, October 3, 10:00:00 p.m.

# 1 Introduction

In this project you are to write a Ruby program that will read Java tokens as input, and perform simple syntax highlighting. Tokens are the fundamental syntactic entities of a programming language, and the first phase of compilation or interpretation, which is scanning or lexical analysis, involves reading the program to be compiled or interpreted and breaking it up into its constituent tokens. Each language has its own specific definition of tokens, but typical tokens include operators, identifiers, character strings, integers, floats, etc. Compilers, interpreters, debuggers, and other programs that have to read and understand programs or fragments of program text typically use regular expressions to describe tokens.

Syntax highlighting refers to the ability of some text editors, such as Emacs or the Eclipse text editor for example, to display information using different fonts and colors, depending upon the types of items that appear. For example, syntax highlighting of a file in a programming language might display reserved words in one color or font, comments in a different color or font, character string literals in another color or font, etc. In this program you won't be implementing a text editor, but rather your program will read input representing Java tokens and output HTML code which, if viewed in a web browser, would display the Java tokens in different colors and fonts. You will have to write various Ruby regular expressions to recognize the different types of Java tokens that may appear in the input.

In grading we will check your program's output to see whether it is correct, but we won't be displaying it in a web browser. However, you can view its output in a browser, which may be a way to easily see whether your program is working to at least a certain extent or not.

It's important to point out that if this was a real program it would probably not be written in the way you will have to write it. Recognizing the validity of complex input such as Java source code would be done in practice using parsing techniques and tools, but the project is just an exercise in learning and using regular expressions, and also of course getting experience coding in Ruby.

A separate handout has been placed on the class webpage (under "Administrative and information") that details the course project grading and submission policies, and also describe how to set up your account on the Grace machines. Once you have set up your account (you must have done so first), to start working on the project just log into the Grace machines and use the commands:

```
cp -r ~/330public/proj1 ~/330
cd ~/330/proj1
```

The directory `proj1` that you will be copying contains the public test inputs and outputs, two other project files mentioned below, and a (hidden) file `.submit` that will allow the project submission program to submit your code. As above, cd to the directory `proj1` created in your extra course disk space, and do your programming there, as your project will need to be in that directory when you submit.

Note that as of the time this project assignment is being posted the project hasn't been set up on the submit server yet, but that will be done in a few days. However, the `proj1` directory created above contains the public test inputs, and their corresponding outputs, so you can determine yourself exactly whether your program works for those inputs or not.

# 2 Program input, output, and identification of tokens

## 2.1 Input

Your program is to be able to recognize ten different types of tokens in its input, as described below. In the discussion below "whitespace character" refers to a blank space, tab, newline, or carriage return character, and "whitespace" refers to a sequence of one or more consecutive whitespace characters.

Your program must read from its standard input a sequence of zero or more lines, each of which will end with a newline, and each of which can contain zero or more characters before the terminating newline.

### 2.1.1 Java tokens

The following are the types of tokens that your program must be able to identify, and their descriptions. Note that the token type names will be included in your program's output.

**ID:** ID refers to a Java identifier, which begins with a Java letter; the Java letter is followed by any sequence of zero or more Java letters or digit characters (or both). Java letters are considered to be the uppercase and lowercase English alphabet characters, as well as the underscore (`_`) and dollar sign (`$`) characters. The digit characters are just `0` through `9`.

**KEY:** KEY refers to a Java keyword, of which, for purposes of this project, the following 50 exist:

| | | | | |
|---|---|---|---|---|
| `abstract` | `continue` | `for` | `new` | `switch` |
| `assert` | `default` | `goto` | `package` | `synchronized` |
| `boolean` | `do` | `if` | `private` | `this` |
| `break` | `double` | `implements` | `protected` | `throw` |
| `byte` | `else` | `import` | `public` | `throws` |
| `case` | `enum` | `instanceof` | `return` | `transient` |
| `catch` | `extends` | `int` | `short` | `try` |
| `char` | `final` | `interface` | `static` | `void` |
| `class` | `finally` | `long` | `strictfp` | `volatile` |
| `const` | `float` | `native` | `super` | `while` |

The keywords are all case–sensitive and must be spelled exactly as above. Note that although the keywords also satisfy the description of Java identifiers ID above, these 50 words must be categorized as keywords.

**BOOL:** BOOL refers to a Java boolean literal, of which there are two: `true` and `false`. Boolean literals are also case–sensitive, and must also be spelled exactly as shown. The boolean literals would also satisfy the description of Java identifiers ID above, but must be categorized instead as BOOL.

**INT:** INT refers to Java literal integers, which are nonempty sequences of octal, decimal, or hexadecimal digits that follow the restrictions below. The octal digits are `0` through `7`, while the decimal digits are `0` through `9`, and the hexadecimal digits are `0` through `9` and also the lowercase letters `a` through `f` and the uppercase letters `A` through `F`.

The base of the number is determined by the beginning of the literal:

- Octal numbers begin with a leading `0`, which is followed by one or more octal digits. For example, `0237` is a valid integer literal, but `058` is not, since `8` is not an octal digit.
- Hexadecimal numbers begin with either the character sequence `0x` or the sequence `0X`, which is followed by one or more hexadecimal digits. For example, `0x4a8` and `0xACED` are valid integer literals.
- Any other consecutive sequence of one or more digits is a decimal integer literal. For example, `330` is a valid integer literal, but `33a` is not.

Note that `0` is also a valid integer literal whose base is immaterial. Also note that by the above definition, signed integer literals are not possible in this project.

Any integer constant, of any base, including `0`, may optionally be followed by a long type modifier, which is either an uppercase `L` or a lowercase `l`. For example, `0x1aL` and `330l` are both valid integer constants.

**FLOAT:** FLOAT refers to Java real literals, which are sequences of one or more decimal digits with an optional decimal point, optionally followed by an exponent part. At least one digit must appear before the exponent if it is present. Either the decimal point or the exponent may be omitted, but not both. An exponent consists of an exponent marker (either `e` or `E`), optionally followed by a sign (`+` or `-`), followed by one or

more decimal digits. The entire literal (including the exponent if present) can optionally end with one of the letters `f`, `F`, `d` or `D` to denote a single or double precision constant. Note also that signed float literals are also not possible.

**CHAR:** CHAR represents a Java character literal, which is either a single character enclosed in single quote marks (''), or is an escape sequence enclosed in single quote marks. Any character other than a newline or a single quote may appear as a single character in a character literal. Escape sequences begin with a backslash (\), which is followed by any character other than a newline. All of the following are examples of valid character literals: `'a'`, `'\a'`, `'\n'`, `'\\'`, and `'\''`, while `'''` is not.

**STR:** STR represents Java character strings. A Java string is a sequence of zero or more characters enclosed by double–quote marks ("). Any Java character literal (defined under CHAR above, without surrounding single–quote marks) may appear between the double quotes of a Java string, other than a double quote mark itself. A double quote mark **may** be included in a Java string by using the escape sequence `\"`. Such an escape sequence cannot be the double quote mark that terminates the Java string.

**OPER:** OPER represents Java operators, of which the following 37 exist:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| `!` | `&` | `*=` | `-` | `/=` | `<<=` | `>` | `>>>` | `^=` | `~` |
| `!=` | `&&` | `+` | `--` | `:` | `<=` | `>=` | `>>>=` | `\|` | |
| `%` | `&=` | `++` | `-=` | `<` | `=` | `>>` | `?` | `\|=` | |
| `%=` | `*` | `+=` | `/` | `<<` | `==` | `>>=` | `^` | `\|\|` | |

**COMM:** COMM represents Java comments, which can just be considered to be long tokens. Only one type of comment is possible in this project, which is a single–line comment that begins with `//` and consists of all of the characters following that, until the newline marking the end of the input line. Consequently a newline cannot be present in a comment of this type.

**OTHER:** OTHER represents any sequence of characters that doesn't match any of the token types listed above.

The input to your program will be very similar to a sequence of tokens in real Java, but there are some differences in tokens as described here compared to tokens in real Java. Your program should recognize tokens as described above, **rather than** what would be the case for real Java, in all cases where these would differ, so if you have any questions about how tokens should be classified you should refer to the descriptions above rather than to a Java reference (and rather than checking what a Java compiler would accept).

## 2.2 Output

Your program must write all of its results to its standard output. For every token type **other than** OTHER, your program must print an HTML `span` element, which is just a string of the form:

`<span class="`*type*`">`*output-token*`</span>`

where *type* represents the type name of the Java token (one of the words ID, KEY, BOOL, INT, etc.), and *output–token* represents a slightly–modified version of the token itself, produced from the token that was read by performing any of the following four replacements, if applicable:

| character | replacement text |
|---|---|
| `&` | `&` |
| `<` | `<` |
| `>` | `>` |
| `"` | `&quot;` |

For example, if the input contains the INT token `330`, the output that must be produced for this part of the input should be:

```
<span class="INT">330</span>.
```

If the input contains the STR token `"Ruby"`, the output produced for it should be

```
<span class="STR">&quot;Ruby&quot;</span>
```

Tokens of type OTHER should be printed (almost) exactly as they appear in the input, meaning **without** a span element, but **with** the four replacements mentioned above performed, if applicable. Note that since whitespace matches the OTHER token, and tokens of type OTHER should be printed exactly to the output exactly as they appeared in the input (other than the four replacements, which obviously would not apply to whitespace), all whitespace in the input will be preserved exactly in the output.

### 2.3 Command–line argument

In order that the output of your program will be valid HTML that can be read by a web browser, it must be preceded by a prologue and followed by an epilogue. If the single argument `-n` appears on the command line then the printing of the prologue and epilogue must be suppressed, otherwise they must both be printed. Additional or invalid command–line arguments (anything other than `-n`) should simply be ignored.

The prologue that should be printed consists of the following exact text (which will be placed on the Grace machines, where you can copy it without needing to type it manually, which could lead to errors). Unless the option `-n` appears, the prologue must be the very first output that is printed.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>source file</title>
<style type="text/css"><!--
BODY   { background: white; }
.KEY   { color: blue; font-weight: bold; }
.STR   { color: green; }
.INT   { color: red; }
.FLOAT { color: purple; }
.COMM  { color: green; font-style: italic; }
.CHAR  { color: fuchsia; }
.BOOL  { color: teal; font-weight: bold; }
.OPER  { color: aqua; }
.ID    { color: maroon; }
// more can be added
--></style>
</head>
<body>
<pre>
<tt>
```

Note that the output produced by your program must begin **immediately** after the opening `<tt>` tag, meaning following it on the same line, as the example below illustrates.

The epilogue that should be printed as your program's very last output (unless the option `-n` is given) is the exact following lines (this will also be placed on the Grace machines):

```
</tt>
</pre>
</body>
</html>
```

### 2.4 Identification of tokens

In many programming languages there is no specific separator between tokens. Tokens may be separated by whitespace, and in many languages many types of tokens may simply be adjacent. In this project whitespace is matched by OTHER, therefore tokens are always considered to be adjacent. The first token always starts at the **beginning** of an input line. The second token starts with the very first character that does not belong to the first token, and so on. Therefore determining where a token starts just depends upon where the previous token ended. Note that there are no multiline tokens; the longest possible token would be just one entire line.

To determine where a token ends, your program must determine the **longest** token that matches from a given starting position. In order to do this, it must try to match the input against all of the token types, and the longest match is the one that should apply. In order for this process to work the OTHER token must be always be the last to be matched against, and it should only match one character. The only exception to the above is the comment token COMM. Rather than the longest token that matches from a given starting position, if it matches it always just matches everything up until the end of the input line.

Examples are:

- If the input contains just the thirteen characters `intergalactic`, then the token is an ID and contains all thirteen characters. Note that `int` would be a valid KEY, but it is shorter than `intergalactic`, so the token is an ID.
- If the input contains just the three characters `int`, then the matched token is a KEY and contains all three characters. Note that `int` is not a valid ID because it was stated above that keywords cannot be identifiers.

## 3 Development suggestions and hints

- Before starting to code, write a variety of examples of tokens of different forms, according to the definitions above, to be sure you have a good understanding of the inputs your program will have to process.
- Create and store regular expressions for any common syntactic elements (syntactic elements that appear in several regular expressions in your program) in variables, then use those variables in pattern matching operations or in forming other regular expressions. This will not only make your program much more readable, but it will be much easier to test, and simpler to adjust if you find you need to modify some of your regular expressions. Similarly, even if a regular expression is only used once, if it's too large, break it up into smaller ones, and test them separately.
- Consider using Ruby's `/x` regular expression option, which allows whitespace, newlines, and comments to be included in a regular expression for readability. For example:

```
/(cat|    # match my favorite pets
  dog)/x
```

- It's suggested that you first write regular expressions describing the syntax of the token types (starting with the simplest ones first, and testing them before going on)– which may be constructed from other smaller regular expressions as mentioned above– before trying to determine the type of token. In other words, initially just write code to test the various regular expressions describing the different token types, since the rest of your program won't work if these are incorrect, before trying to handle the longest match requirement.
- Note that if you have a Ruby program that both reads command–line arguments and standard input, when you call `gets()` it will try to read values from `ARGV`. The solution would be to either shift `ARGV` as command–line arguments are read, so it's empty before starting to read standard input, or, just use a loop of a form like `while (line= STDIN.gets())` to read standard input, rather than `while (line= gets())`.
- If you write methods that should return a true or false value, remember that a Ruby 0 is not false.
- If more than one alternative could match a string in a Ruby regular expression using alternation, the leftmost one will be the one that does. For example, in the fragment `"pineapples" =~ /(app|store|apple)/`, the substring "app" of "pineapples" will match the regular expression, rather than "apple".

# 4 Project requirements and submitting your project

1. Your program should read all of its input from its standard input and write all of its output to its standard output. Of course in UNIX standard input may be redirected from a file, or standard output may be redirected to a file.
2. You may use any Ruby language features in your project that you would like, regardless of whether they were covered in class or not, so long as your program works successfully using the version of Ruby installed on the OIT Grace Cluster and on the CMSC project submission server.
3. Your submitted program file **must** be in a file named "`proj1.rb`", otherwise the submit server will not recognize it, consequently its score will be zero. Although Ruby programs can be broken up into multiple source files (not shown in class), your program should consist of just this one source file.
4. Your output should match the expected public test outputs exactly. To check this, you can use the UNIX `diff` command. For example:

   ```
   proj1.rb < public1.input > my-output
   diff my-output public1.output
   ```

   If no output at all is produced by `diff`, your output matches the expected output and is correct.
5. You aren't required to use the `-w` option for your project, but it is nevertheless recommended, as it will just assist you in finding potential bugs.
6. Your program **must have** a comment near the top that contains your name, TerpConnect login ID (which is your directory ID), your university ID number, and your section number.

   The Campus Senate has adopted a policy asking students to include the following statement on each major assignment in every course: "I pledge on my honor that I have not given or received any unauthorized assistance on this assignment." Consequently you're requested to include this pledge in a comment near the top of your program source file. See the next section for important information regarding academic integrity.
7. Note that although you are not being graded on your source code or style (see the separate project grading handout), if the TAs cannot read or understand your code they cannot help should you have to come to office hours, until you return with a well–written and clear program.
8. To submit your program, just type the single command `submit` from the `proj1` directory that you copied above, where your program file `proj1.rb` should be. You will have to enter your UMCP directory ID and password. Then log into the submit server to verify that your submission worked there.

# 5 Example

The output below was produced by running the program with input redirected from the file shown on the right. Try storing your programs' output such as this in a file with the extension `.html` and displaying it in your web browser.

Note that the tokens in this example just happened to be arranged in the form of a valid Java class, just for clarity, but that may not be the case in all inputs your program has to process. In fact, it's more likely that its input will just be a random sequence of Java tokens. Your program should not attempt to check whether the tokens in its input are in the form of a valid Java class; this would require parsing techniques that have not been covered.

```
class Example {
  int x;
  static String s;
  { x= 330; }
  static {
    s= "We love Ruby!";
  }
}
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>source file</title>
<style type="text/css"><!--
BODY   { background: white; }
.KEY   { color: blue; font-weight: bold; }
.STR   { color: green; }
.INT   { color: red; }
.FLOAT { color: purple; }
.COMM  { color: green; font-style: italic; }
.CHAR  { color: fuchsia; }
.BOOL  { color: teal; font-weight: bold; }
.OPER  { color: aqua; }
.ID    { color: maroon; }
// more can be added
--></style>
</head>
<body>
<pre>
<tt><span class="KEY">class</span> <span class="ID">Example</span> {
  <span class="KEY">int</span> <span class="ID">x</span>;
  <span class="KEY">static</span> <span class="ID">String</span> <span class="ID">s</span>;
  { <span class="ID">x</span><span class="OPER">=</span> <span class="INT">330</span>; }
  <span class="KEY">static</span> {
    <span class="ID">s</span><span class="OPER">=</span> <span class="STR">&quot;We love Ruby!&quot;</span>;
  }
}
</tt>
</pre>
</body>
</html>
```

## 6 Academic integrity

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.