Introduction to Language Theory and Compilation Exercises

Computer Session 1: `lex/flex` scanner generator

Reminders

Extended regular expressions (ERE)

Expression	Accepted language
`r*`	0 or more `r`s
`r+`	1 or more `r`s
`r?`	0 or 1 `r`
`[abc]`	`a` or `b` or `c`
`[a-z]`	Any character in the interval `a`...`z`
`.`	Any character except `\n`
`[^s]`	Any character but those in `s`
`r{m,n}`	Between `m` and `n` occurrences of `r`

Expression	Accepted language
`r1 r2`	The concatenation of `r1` and `r2`
`r1 | r2`	`r1` or `r2`
`(r)`	`r`
`^r`	`r` if it starts a line
`r$`	`r` if it ends a line
`"s"`	The string `s`
`\c`	The character `c`
`r1(?=r2)`	`r1` when it's followed by `r2`

Scanner

A *scanner* is a program that reads text on the standard input and prints it on standard output after applying operations. For example, a filter that replaces all `a`s with `b`s and that receives `abracadabra` on input would output `bbrbcbdbbrb`.

Specification format

You can find JFlex on `http://jflex.de`.

The manual is available at `http://jflex.de/manual.html`. A `JFlex` (a Java implementation of flex) specification is made of three parts separated by lines with `%%`:

- **Part 1**: arbitrary programming code to be prepended in the output Java scanner program
- **Part 2**: regular expression definitions and arbitrary Java code (between `%{` and `%}`) to be inserted at the start of the scanner program
  - The JFlex options (%class **Name**, %unicode, %line, %column, %standalone, %cup, ...)
  - The regular expression definitions are used as "macros" in part 3.
  - The Java additional code of the scanner (`%{ code }%`, `%init{ code executed before the parsing }init%`, `%eof{ code executed after the parsing }eof%`, ...)
- **Part 3**: translation rules of the following shape: `Regex  {Action}`
  - `Regex` is an *extended regular expression* (ERE)
  - `Action` is a *Java code snippet* that will be executed each time a *token* matching `Regex` is encountered.
  - The regular expressions defined in Part 2 can be used by putting their names in curly braces `{ }`.

Variables and special actions

When writing *actions*, some special variables and macros can be accessed:

- `yylength()` contains the *length* of the recognized token

- `yytext()` is a the actual string that was matched by the regular expression.
- `yyline` is the line counter (requires the option %line).
- `yycolumn` is the column counter (requires the option %column).
- `EOF` is the End-of-file marker.

Warning for Mac users: On Mac, files which do not end with an empty line can make the lexer "forget about" the last line. When you test your lexer, make sure that your test files end with an empty line.

Meta states

You can define inclusive or exclusive (use **x**state instead of state) states with the command:

```
%xstate states list separated by a comma;
```

Each state has to contain some regular expressions. An action can be going into another state by using the function `yybegin(Name of the destination state)`. The first state used is defined by JFlex and it called **YYINITIAL**. For instance:

```
⋮
xstate YYINITIAL, PRINT;
%%
<YYINITIAL> {
   "print" {yybegin(PRINT);}
}
<PRINT> {
   ";" {yybegin(YYINITIAL);}
   .   {System.out.println(yytext());}
}
```

Executable

To obtain the scanner executable There are two cases. If JFlex is system-wide known:

1. Generate the scanner code with `jflex myspec.flex`
   which creates `Lexer.java` (%class option)
2. Compile the code generated by `JFlex` into a class file: `javac Lexer.java`
   which creates `Lexer.class`
3. Run it with `java Lexer <input file>`

If it is not known, use the `.jar` directly:

1. Generate the scanner code with `java -jar jflex-1.8.2.jar myspec.flex`
   which creates `Lexer.java` (%class option)
2. Compile the code generated by `JFlex` into a class file: `javac Lexer.java`
   which creates `Lexer.class`
3. Run it with `java Lexer <input file>`

Exercises

Ex. 1. Write a scanner that outputs its input file with line numbers in front of every line.

Ex. 2. Write a scanner that outputs the number of alphanumeric characters, alphanumeric words and alphanumeric lines in the input file.

Ex. 3. Write a scanner that only shows the content of comments in the input file. Such comments are enclosed within curly braces `{ }`. You can assume that the input file does not contain curly braces inside comments.

Ex. 4. Write a scanner that transforms the input text by replacing the word "compiler" with "nope" if the line starts with an "a", with "???" if it starts with a "b" and by "!!!" if it starts with a "c".

Ex. 5. Write a *lexical analysis function* that recognizes the following *tokens*:

- Decimal numbers in scientific notation (i.g. -0.4E-1)
- C99 variable identifiers (start by an alpha, followed by arbitrary number of alphanumeric or underscore)
- Relational operators (<, >, ==, `!`=, >=, <=, `!`)
- The `if`, `then` and `else` keywords

Your function should output "`TOKEN NAME: token`", *e.g.* "`C99VAR: myvariable`" and return an object `Symbol` containing the token type, value and position (line and column) for each recognized token. You can use the `Symbol` and `LexicalUnit` classes provided on the Université Virtuelle.

This way, you designed a simple lexer that you can reuse for the project.