COMP 356
Programming Language Structures
Class 10: Scanning (Sections 4.1-2 of *Concepts of Programming Languages*)

Acknowledgment: These notes are based closely on an earlier version by Tim Wahls.

The **scanner** (or **lexical analyzer**) of a compiler processes the source program, recognizing tokens and returning them one at a time. (This is today's topic.)

The **parser** (or **syntactic analyzer**) of a compiler organizes the stream of tokens (from the scanner) into parse trees according to the grammar for the source languages. (Parsing will be covered in the next lecture.)

Separating the scanner and parser into different stages of the compiler:

- makes implementing the parser easier
- increases portability of the compiler
- permits parallel development
- eases tool support

1 Scanning

Scanning is essentially pattern matching—each token is described by some pattern for its lexemes.

Examples:

- the pattern for an identifier token could be: a letter followed by a sequence of letters, digits and underscores. Lexemes of this token include: i3, foo, foo_7, ...
- the pattern for a floating point numeric literal token could be: an optional sign, followed by a sequence of digits, followed by a decimal point, followed by a sequence of digits. Lexemes include: -7.3, 3.44, ...

2 Regular Expressions

Regular expressions are a notation for defining patterns. We will be using regular expressions in a tool called JFLex (described later). JFLex regular expressions operate according to the following notation:

Because `|` has lowest precedence, `ab | c` (the last example above with the parentheses removed) matches `ab` or `c`, and not `a` followed by `b` or `c`.

A *regular definition* is a named regular expression. Such names can be used in place of the regular expression in following definitions.

Examples:

```
letter = [a-zA-Z]
digit = [0-9]
id = {letter}({letter} | {digit} | _)*
num = (+ | -)?{digit}+.{digit}+
```

notation	**name**	**meaning**	**example**	**matches**
`[...]`	character class	any of a set	`[a-zA-Z]`	any letter
`*`	Kleene * Kleene closure	0 or more occurrences	`a*`	0 or more `a`'s
`+`	Kleene + positive closure	1 or more occurrences	`a+`	1 or more `a`'s
`?`	optional	0 or 1 occurrences	`a?`	`a` or ϵ
`|`	alternative		`a | b`	`a` or `b`
`(...)`	grouping		`a(b | c)`	`a` followed by `b` or `c`

Table 1: Regular expression notation for JFlex

The name of a regular definition is put in `{ }` on the R.H.S. of another regular definition. This is to distinguish the use of the named definition, for example, `{digit}`, from the simple pattern given by the letters in the name, for example, `d` followed by `i` followed by `g` followed by `i` followed by `t`.

The *language* specified by a regular expression is the set of all strings that match the pattern given by the regular expression. For example, the language specified by the pattern for `id` given above includes strings such as "`foo`", "`i3`", "`foo_7`" and so on.

A *regular language* is a language that can be specified by a regular expression.

A *context free language* is a language that can be specified by a BNF definition.

Every regular language is context free, but not vice-versa. For example:

$$\{\mathrm{a}^n\mathrm{b}^n \mid n \geq 0\}$$

is a context free language, because it is defined by the following grammar:

<S> $\rightarrow$ a<S>b | ϵ

but is not a regular language. (You will see a proof of this fact in COMP314, Theoretical Foundations of Computer Science.)

Hence, regular expressions are strictly less powerful than context free grammars (BNF definitions). However, they can be used to generate more efficient pattern matchers.

To generate a pattern matcher from regular expressions, tools such as lex and JFLex first convert regular expressions to state transition diagrams, and then generate pattern matching code from the diagrams. In this course, we will use JFlex for this purpose. JFlex is a tool for generating Java implementations of scanners from regular expression definitions of tokens. The next few sections walk through the many details of using an understanding JFlex.

3 Setting your classpath correctly

We will be using two Java programs: JFlex and CUP. The code for these programs is stored in the file `cup.jar`, available on the course webpages. Whenever you run or compile Java, and you want to include the functionality from `cup.jar`, you need to ensure that the Java *classpath* is set correctly. If you know how to do this using the environment variables, feel free to go ahead and do that. For simplicity, however, the following method is recommended: we will set the classpath explicitly each time you run `java` or `javac`. To do this, use the `-cp` option for `java` and `javac`. You need to make sure the class path includes both `cup.jar` and the current directory, which is represented by a single "`.`" character. On Windows, separate these with a "`;`". On Mac, use the separator "`:`". For example, assuming that `cup.jar` is in the current directory, on Windows you might use the complete command

```
        javac  -cp cup.jar;. *.java
```

On a Mac, the same command would be

```
        javac  -cp cup.jar:. *.java
```

In the rest of these notes, we will assume you are on a Mac, and that `cup.jar` is in the current directory.

4 The goal for our first example of scanning

We will construct a JFlex scanner (and, in the next lecture, a CUP parser) for the following simple "calculator" language. This language defines lists of arithmetic expressions:

<exprList> → <exprList> <expr> ; | ϵ
<expr> → num | <expr> addop <expr> | <expr> mulop <expr> | (<expr>)

Of course, this grammar doesn't completely define the language, since we haven't specified what lexemes correspond to the tokens. Informally, the plan is for `addop` to correspond to lexemes "`+`" and "`-`"; `mulop` to correspond to lexemes "`*`" and "`/`"; and `num` to correspond to lexemes like "`455`" and "`23.101`".

Thus, strings in this will include things like:

```
31.3+45;
(4*6.5)-31.3; 555+666;
```

As you can see, our calculator language has several tokens, such as `num`, `mulop`, "`)`" and "`;`". The language will also have a special end-of-file token, `eof`, which represents the end of an input file.

In the next lecture, we will see a way of automatically assigning a numeric code to each token. But for now, we will manually assign arbitrary numeric codes as follows:

token	**numeric code**
`eof`	0
`addop`	1
`(`	2
`mulop`	3
`num`	4
`)`	5
`;`	6

Our immediate objective is to construct a scanner whose input is a string from the calculator language (e.g. "`2+3.1;`") and whose output is a corresponding list of token codes (e.g. "`2+3;`" becomes "`4,1,4,6`"). More precisely, the output will contain numeric token codes and the actual values of the observed lexemes, so the input "`2+3.1;`" should result in:

```
token 4, value 2.0
token 1, value +
token 4, value 3.1
token 6
```

5 Manually specifying numeric token codes

To do this with JFlex, the first step is to write a Java file specifying the above numeric token codes. This is provided as `sym.java` on the course resource page. (Remember, in the next lecture we will see a way of automatically generating the numeric token codes and `sym.java`. But today, we are deliberately doing this part manually in order to understand the scanning phase of a compiler.)

6 Using a `.lex` file to describe the lexemes

The next step is to write a JFlex specification. The main job of this specification is to describe the set of lexemes corresponding to each token, using regular expressions. For example, we can use the following two regular definitions as part of our JFlex specification:

```
DIGIT = [0-9]
NUM = {DIGIT}+("."{DIGIT}+)?
```

There will be many other things in the JFlex specification file too. A JFlex specification is put in a file with a `.lex` extension, and you can look at `example.lex` (provided on the resources page) to get a feel for this now. However, let's ignore the details of the `.lex` file format at this point. Instead, assume that we have a correct `example.lex` file, and let's see how we will use that file.

7 Generating `Yylex`, the scanner class

The next step is to automatically generate a Java program called `Yylex.java`. `Yylex.java` is actually a scanner class that will help us produce output like "`token 4, value 2.0`" in the example above. We will generate `Yylex.java` using `example.lex` as the input, by running the JFlex program. This is done using the following command:

```
java -cp cup.jar:. JFlex.Main example.lex
```

(Any error messages about inability to find main or JFlex.Main indicate a problem with the classpath. Make sure `cup.jar` is in the current directory. If you are on Windows, make sure you are using "`;`" instead of "`:`".)

This is a good time to look through the file `Yylex.java` and pick up just a few ideas of what kind of functionality it provides. The most important thing is to note the constructor `Yylex(java.io.InputStream in)`, and the method `next_token()`. You don't need to read the code—just note that this constructor and method are present, and take a look at their signatures.

8 Running the scanner using `LexTest`

Finally, we are ready to complete our scanner program. The scanner program is called `LexTest.java`, and this file is also available from the resources page. Take a quick look at it now: it uses the `Yylex()` constructor and `next_token()` method discussed above.

At this point, we need to compile `Yylex.java` and `LexTest.java`:

```
javac -cp cup.jar:. *.java
```

And finally, we can run our scanner program:

```
java -cp cup.jar:. LexTest
```

Try typing in the above example ("2+3.1;"), and check that the output is what you expect. You can also try more complex examples by reading input from a file. For example, the file `input-for-LexTest.txt` is provided on the resources page, and you can use test it via the command:

```
java -cp cup.jar:. LexTest < input-for-LexTest.txt
```

Try some more complex inputs now, using input from a file as in the command above.

9 Many important details

Now that we have understood the final goal, let's go back and look at some of the details of how we got there. First, note that the constructor for class `Yylex` takes a stream reader (and you can use `System.in` for standard input) as the input stream to read the source program from. The other significant method in class `Yylex` is `next_token()`, which returns the next token. More precisely, the method returns instances of class `Symbol` from `java_cup.runtime.Symbol`.

We don't have direct access to the source code of the `Symbol` class. However, it's important to understand two of the fields of this class. They are:

- `int sym`: The token recognized (represented as a numeric code, as discussed above). For example, when a `num` is recognized for our calculator example, the numeric code would be 4.
- `Object value` - can contain instances of any Java class, and is often used to hold lexemes as instances of classes such as `String`, `Double` and `Integer`. For example, when the input "`23.1`" is processed in our calculator example above, this token would be recognized as a `num`, and its `value` field would hold a `Double`, equal to 23.1.

We also need to know about the constructors of class `Symbol`. The `Symbol(int)` constructor takes the token code (of type `int`) as its parameter. The two-argument constructor `Symbol(int, Object)` has parameters for both the token code and the value field.

Now let's understand a few more details of the format of JFlex specification files, which are also known as `.lex` files. Follow along with the `example.lex` file provided, in order to see examples of all the following details. The high-level format is as follows:
user code
`%%`
JFlex directives
`%%`
regular expression rules and actions

The contents of the *user code* section are copied directly into the generated scanner code. For example, `import`s of any needed Java classes would go here.

The *JFlex directives* section is used to customize the behavior of JFlex. Some common directives include:

```
%cup  // to specify CUP compatibility
// the following three lines make JFlex recognize EOF properly
%eofval{
  return new Symbol(sym.EOF);
%eofval}
```

Notes on JFlex directives:

- the value of `sym.EOF` must be 0 (this only matters if you construct class `sym` by hand)
- every directive must start at the beginning of a line
- the line breaks in the macro defined with `%eofval` above are critical

Any regular definitions also go in this section. These use exactly the syntax of regular definitions presented in class, except that double quotes (`" "`) are used instead of single quotes (`' '`) to distinguish metasymbols from literal characters.

The *regular expression rules and actions* give a sequence of regular expression patterns for tokens and actions to be executed when the associated actions are matched. Notes:

- each action is Java code, and Java keyword `return` can be used to specify the value returned by method `next_token()`
- whitespace (spaces, tabs, ...) in the source program is ignored by giving a pattern for white space with an empty action. Since the action contains no `return`, method `next_token()` ignores white space and keeps scanning the source program until something else is found.
- the patterns are tried in order, and the first matching pattern is used.
- a period (.) is a pattern that matches any character. This is useful for reporting lexical errors. However, this also means that the character period must be put in double quotes, i.e. use `"."` in a pattern if you mean the character `.` and not the metasymbol.
- in each action, the string of characters that matched the associated pattern (the lexeme) can be accessed using function `yytext()`