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

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 is 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 is 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.

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

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}+
```

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** in 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.

3 Using JFLex

JFlex is a tool for generating Java implementations of scanners from regular expression definitions of tokens.

To configure your account for using JFlex (and CUP, which is used in the next lecture), first download the file `cup.jar` from the course web page (see the links on the resource page for today). Remember which directory this file is stored in. Next, use a text editor to create/edit a file called `.profile` in your home directory, add the following content to your `.profile`, and save it:

```
CLASSPATH=~/cup.jar:.
export CLASSPATH
PATH=.:$PATH
export PATH
```

If you saved `cup.jar` somewhere other than your home directory, the first line should include the appropriate directory. For example, if you saved `cup.jar` to your `Desktop` directory, the first line should be:

```
CLASSPATH=~/Desktop/cup.jar:.
```

Your PATH and CLASSPATH environment variables are now set correctly for all of the software we will use in this course.

A JFlex specification is put in a file with a .lex extension (e.g. example.lex). Assuming that example.lex contains a valid JFlex specification, the command:

```
java JFlex.Main example.lex
```

generates the scanner code and places it in file Yylex.java. (Any error messages about inability to find main or JFlex.Main indicate that your CLASSPATH environment variable is not set correctly.)

The scanner code is in class `Yylex`. The constructor for class `Yylex` takes a stream reader (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. For use with CUP (a parser generator), `next_token()` should return instances of class `Symbol` from `java_cup.runtime.Symbol` (which requires that CUP is installed and your CLASSPATH is set correctly).

Class `Symbol` has two significant fields (data members):

- `int sym` - the token recognized (represented as an integer)
- `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`. CUP generates parsers from attribute grammars, and this `value` field will become an intrinsic attribute of the token in the attribute grammar.

The one-argument constructor of class `Symbol` takes the token (of type `int`) as its parameter, while the two-argument constructor for class `Symbol` has parameters for the values of both of these fields.

The token values should be defined as integer constants in class `sym`. CUP generates this class automatically, or it can be constructed by hand if CUP is not being used.

We will construct a JFlex scanner and a CUP parser for the following simple "calculator" language. This language defines lists of arithmetic expressions, and the output from our parser will be the value of each expression.

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

Format of a JFlex specification file:

user code
%%
JFlex directives
%%
regular expression rules and actions

For an example, see file example.lex on the course web page.

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()`
- white space (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. `"."`, 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()`