Converting Simple Regular Expresssions into a Lexer

regular expression	*lexercode*
$a \quad (a \in A)$	*if* $(current = a)$ *next else ...*
$r_1 r_2$	$code(r_1);\ code(r_2)$
$r_1 \mid r_2$	*if* $(current \in first(r_1))$ $code(r_1)$ *else* $code(r_2)$
r^*	*while* $(current \in first(r))$ $code(r)$

More complex cases

In other cases, a few upcoming characters ("lookahead") are not sufficient to determine which token is coming up.

Examples:
A language might have separate numeric literal tokens to simplify type checking:

- integer constants: *digit digit**
- floating point constants: *digit digit** . *digit digit**

Floating point constants must contain a period (e.g., Modula-2).

Division sign begins with same character as // comments.
Equality can begin several different tokens.

In such cases, we process characters and store them until we have enough information to make the decision on the current token.

Example of a part of a lexical analyzer

```
ch.current match {
  case '(' ⇒ {current = OPAREN; ch.next; return}
  case ')' ⇒ {current = CPAREN; ch.next; return}
  case '+' ⇒ {current = PLUS; ch.next; return}
  case '/' ⇒ {current = DIV; ch.next; return}
  case '*' ⇒ {current = MUL; ch.next; return}
  case '=' ⇒ { // more tricky because there can be =, ==
    ch.next
    if (ch.current == '=') {ch.next; current = CompareEQ; return}
    else {current = AssignEQ; return}
  }
  case '<' ⇒ { // more tricky because there can be <, <=
    ch.next
    if (ch.current == '=') {ch.next; current = LEQ; return}
    else {current = LESS; return}
  }
}
```

Example of a part of a lexical analyzer

```
ch.current match {
  case '(' ⇒ {current = OPAREN; ch.next; return}
  case ')' ⇒ {current = CPAREN; ch.next; return}
  case '+' ⇒ {current = PLUS; ch.next; return}
  case '/' ⇒ {current = DIV; ch.next; return}
  case '*' ⇒ {current = MUL; ch.next; return}
  case '=' ⇒ { // more tricky because there can be =, ==
    ch.next
    if (ch.current == '=') {ch.next; current = CompareEQ; return}
    else {current = AssignEQ; return}
  }
  case '<' ⇒ { // more tricky because there can be <, <=
    ch.next
    if (ch.current == '=') {ch.next; current = LEQ; return}
    else {current = LESS; return}
  }
}
```

What if we omit ch.next?

Example of a part of a lexical analyzer

```
ch.current match {
  case '(' ⇒ {current = OPAREN; ch.next; return}
  case ')' ⇒ {current = CPAREN; ch.next; return}
  case '+' ⇒ {current = PLUS; ch.next; return}
  case '/' ⇒ {current = DIV; ch.next; return}
  case '*' ⇒ {current = MUL; ch.next; return}
  case '=' ⇒ { // more tricky because there can be =, ==
    ch.next
    if (ch.current == '=') {ch.next; current = CompareEQ; return}
    else {current = AssignEQ; return}
  }
  case '<' ⇒ { // more tricky because there can be <, <=
    ch.next
    if (ch.current == '=') {ch.next; current = LEQ; return}
    else {current = LESS; return}
  }
}
```

What if we omit ch.next?

Lexer could generate a non-existing equality token!

White spaces and comments

Whitespace can be defined as a token, using space character, tabs, and various end of line characters. Similarly for comments.

In most languages (Java, ML, C) white spaces and comments can occur between any two other tokens have no meaning, so parser does not want to see them.

Convention: the lexical analyzer removes those "tokens" from its output. Instead, it always finds the next non-whitespace non-comment token.

Other conventions and interpretations of new line became popular to make code more concise (sensitivity to end of line or indentation). Not our problem in this course! Tools that do formatting of source also must remember comments. We ignore those.

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
    while (!isEOL && !isEOF) {
      ch.next
    }
  } else {
```

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
     while (!isEOL && !isEOF) {
       ch.next
     }
  } else {
    ch.current = DIV
  }
}
```

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
    while (!isEOL && !isEOF) {
      ch.next
    }
  } else {
    ch.current = DIV
  }
}
```

Nested comments: this is a single comment:
/* foo /* bar */ baz */
Solution:

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
     while (!isEOL && !isEOF) {
       ch.next
     }
  } else {
    ch.current = DIV
  }
}
```

Nested comments: this is a single comment:
/* foo /* bar */ baz */
Solution: use a counter for nesting depth

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	<=
LT:	<
EQ:	=

How can we split this input into subsequences, each of which in a token:

interpreters <= *compilers*

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	<=
LT:	<
EQ:	=

How can we split this input into subsequences, each of which in a token:

interpreters <= *compilers*

Some candidate solutions:

ID(interpreters) LE ID(compilers) - OK, longest match rule
ID(inter) ID(preters) LE ID(compilers)
ID(interpreters) LT EQ ID(compilers)

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	<=
LT:	<
EQ:	=

How can we split this input into subsequences, each of which in a token:

interpreters <= *compilers*

Some candidate solutions:

ID(interpreters) LE ID(compilers) - OK, longest match rule
ID(inter) ID(preters) LE ID(compilers) - not longest match: ID(inter)
ID(interpreters) LT EQ ID(compilers)

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	<=
LT:	<
EQ:	=

How can we split this input into subsequences, each of which in a token:

interpreters <= *compilers*

Some candidate solutions:

ID(interpreters) LE ID(compilers)	- OK, longest match rule
ID(inter) ID(preters) LE ID(compilers)	- not longest match: ID(inter)
ID(interpreters) LT EQ ID(compilers)	- not longest match: LT

Longest match rule is greedy, but that's OK

Consider language with ONLY these three operators:

LT: <

LE: <=

IMP: =>

Given sequence: <=>

lexer will split it as <=,> , return LE as token, then report unknown token error on >.

This is the behavior that we expect.

Longest match rule is greedy, but that's OK

Consider language with ONLY these three operators:

LT: <

LE: <=

IMP: =>

Given sequence: <=>

lexer will split it as <=,> , return LE as token, then report unknown token error on >.
This is the behavior that we expect.

This is despite the fact that one could in principle split the input into < and =>, which correspond to sequence LT IMP. But a split into < and => would not satisfy longest match rule, so we do *not* want it. Reporting error is the right thing to do here.

This behavior is not a restriction in practice: programmers we can insert extra spaces to stop longest match rule from taking too many characters.

Token priority

What if our token classes intersect?
Longest match rule does not help, because the same string belongs to two regular expressions
Examples:

- a keyword is also an identifier
- a constant that can be integer or floating point

Solution is priority: order all tokens and in case of overlap take one earlier in the list (higher priority). This avoids having to *subtract* language of one token from another.
Examples:

- if it matches regular expression for both a keyword and an identifier, then we define that it is a keyword.
- if it matches both integer constant and floating point constant regular expression, then we define it to be (for example) integer constant.

Token priorities for overlapping tokens must be specified in language definition.