Compilers Construction Course Project

You will begin to develop a lexical analyzer for java programming language.

You will use two approaches:

1. **Writing the lexical analyzer without using any third-party tool.**
2. **Writing the lexical analyzer using lex (flex) tool.**

You will implement the two approaches.

Teams should be formed with a maximum of five students per a team.

- **Below the lexical structure or specification (not regular expressions) of the java programming language for which you should write your analyzer.**

Input Elements and Tokens:

```
Input:
{InputElement} [Sub]

InputElement:
WhiteSpace
Comment
Token
```

```
Token:
Identifier
Keyword
Literal
Separator
Operator
```

```
Sub:
the ASCII SUB character, also known as "control-Z"
```

White Space:

```
WhiteSpace:
the ASCII SP character, also known as "space"
the ASCII HT character, also known as "horizontal tab"    (\t)
the ASCII FF character, also known as "form feed"         (\f)
the ASCII LF character, also known as "newline"           (\n)
the ASCII CR character, also known as "return"            (\r)
the ASCII CR character followed by the ASCII LF character (\r\n)
```

Comments

There are two kinds of comments:

- `/*` *text* `*/`

  A *traditional comment*: all the text from the ASCII characters `/*` to the ASCII characters `*/` is ignored (as in C and C++).

- `//` *text*

An *end-of-line comment*: all the text from the ASCII characters `//` to the end of the line is ignored (as in C++).

Identifiers:

Identifier:
IdentifierChars but not
a Keyword or BooleanLiteral or NullLiteral

IdentifierChars:
*JavaLetter {JavaLetterOrDigit}**

JavaLetter:
any Unicode character that is a "Java letter"

note: we will use only these letters for simplicity, so the regular expression for letter is [A-Za-z_]

JavaLetterOrDigit:
any Unicode character that is a "Java letter-or-digit"

note: we will use only these letters for simplicity, so the regular expression for digit is [0-9]

Keywords:

Keyword:
(one of)

```
abstract    continue    for          new          switch
assert      default     if           package      synchronized
boolean     do          goto         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
```

Literals:

```
Literal:
IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
NullLiteral
```

Integer Literals:

```
IntegerLiteral:
DecimalIntegerLiteral
HexIntegerLiteral
OctalIntegerLiteral
BinaryIntegerLiteral
```

```
DecimalIntegerLiteral:
DecimalNumeral [IntegerTypeSuffix]

HexIntegerLiteral:
HexNumeral [IntegerTypeSuffix]

OctalIntegerLiteral:
OctalNumeral [IntegerTypeSuffix]

BinaryIntegerLiteral:
BinaryNumeral [IntegerTypeSuffix]

IntegerTypeSuffix:
(one of)
l L
```

```
DecimalNumeral:
0
NonZeroDigit [Digits]

Digits:
0
NonZeroDigit
```

NonZeroDigit:
(one of)
1 2 3 4 5 6 7 8 9

HexNumeral:
0 x *HexDigits*
0 X *HexDigits*
HexDigits:
{HexDigit}+
HexDigit:
(one of)
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

OctalNumeral:
0 *OctalDigits*

OctalDigits:
{OctalDigit}+

OctalDigit:
(one of)
0 1 2 3 4 5 6 7

BinaryNumeral:
0 b *BinaryDigits*
0 B *BinaryDigits*

BinaryDigits:
{BinaryDigit}+

BinaryDigit:
(one of)
0 1

Floating-Point Literals:

FloatingPointLiteral:
DecimalFloatingPointLiteral

DecimalFloatingPointLiteral:
Digits . *[Digits] [ExponentPart] [FloatTypeSuffix]*
. *Digits [ExponentPart] [FloatTypeSuffix]*
Digits ExponentPart [FloatTypeSuffix]
Digits [ExponentPart] FloatTypeSuffix

ExponentPart:
ExponentIndicator SignedInteger

ExponentIndicator:
(one of)
e E

SignedInteger:
[Sign] Digits

Sign:
(one of)
+ -

FloatTypeSuffix:
(one of)
f F d D

Boolean Literals:

BooleanLiteral:
(one of)
true false

Character Literals:

CharacterLiteral:
' *SingleCharacter* '
' *EscapeSequence* '

SingleCharacter:
*InputCharacter but not ' or *

The following are examples of `char` literals:

- `'a'`
- `'%'`
- `'\t'`
- `'\\'`
- `'\''`
- `'\u03a9'`
- `'\uFFFF'`
- `'\177'`
- `'*'`

String Literals:

StringLiteral:
" {*StringCharacter*} "

StringCharacter:
InputCharacter but not " or \
EscapeSequence

The following are examples of string literals:

```
""                      // the empty string
"\""                    // a string containing " alone
"This is a string"      // a string containing 16 characters
"This is a " +          // actually a string-valued constant
expression,
    "two-line string"   // formed from two string literals
```

Escape Sequences for Character and String Literals:

EscapeSequence:
\ b *(backspace BS, Unicode* \u0008*)*
\ t *(horizontal tab HT, Unicode* \u0009*)*
\ n *(linefeed LF, Unicode* \u000a*)*
\ f *(form feed FF, Unicode* \u000c*)*
\ r *(carriage return CR, Unicode* \u000d*)*
\ " *(double quote ", Unicode* \u0022*)*
\ ' *(single quote ', Unicode* \u0027*)*
\ \ *(backslash \, Unicode* \u005c*)*
UnicodeInputCharacter

UnicodeInputCharacter:
UnicodeEscape

UnicodeEscape:
\ *UnicodeMarker* *HexDigit* *HexDigit* *HexDigit* *HexDigit*

UnicodeMarker:
u *{u}*

HexDigit:
(one of)
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

Text Blocks

TextBlock:

" " " {TextBlockWhiteSpace} LineTerminator {TextBlockCharacter} " " "

TextBlockWhiteSpace:

WhiteSpace but not LineTerminator

TextBlockCharacter:

InputCharacter but not \

EscapeSequence

LineTerminator

Example:

```
//multi line string
String html = "<html>\n" +
" <body>\n" +
" <p>Hello, world</p>\n" +
" </body>\n" +
"</html>\n";
//text Block
```

String html = """

<html>

<body>

Hello, world

</body>

</html>

""";

Note: the escape sequences are the same but (“ ‘ \) don’t require escape operator before it.

The Null Literal:

```
NullLiteral:
null
```

Separators:

```
Separator:
(one of)
```

```
(    )    {    }    [    ]    ;    ,    .    ...    @    ::
```

Operators:

```
Operator:
(one of)
```

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