How to run it:

There should be a 'compilers.jar' -file at the root of the ZIP, next to the 'doc' and 'src' folders. Run it with `java -jar compilers.jar "path-to-file-containing-input"`

For example, inside the extracted ZIP: `java -jar compilers.jar ./doc/test.txt`

MiniPL token patterns as regex / regdef

The tokens are identified using a switch statement, with kind of a variable lookahead. What I mean by variable lookaheas is that the lexer normally reads input one character at a time, but for some tokens, like string literals, it needs to look ahead until the end of that string to make sure if it's complete or not. To be able to do this, it keeps track of the current actual position of the 'read head', and the current 'lookahead' position, so it can return back to it, or set the 'real' position to the lookahead position to continue reading after the current token.

I have defined the following token types for the Lexer to recognize:

Single character tokens:

All are matched 'as is'. So the lexer first checks for each single character token when it reads a new character (unless it has already started to check a longer one), and if it is that character, it creates the corresponding token. This works as it follows the maximal munch practice. As these token are only one character, (and they are not allowed as starting points for other tokens) there is no possibility that they are anything else. (Meaning there is not much point to give regex patterns for these)

```
// Misc. character tokens
LEFT_PAREN
RIGHT_PAREN
COMMA
DOT
SEMICOLON
COLON

// Arithmetic
MINUS
PLUS
STAR
SLASH

// Logical operators
EQUAL
GREATER
LESS
OR
AND

EOF
```

Multi-character tokens:

```
// Literal values (strings, integers and booleans)
STRING_LIT - /"[^"]*"/
NUMBER - /[0-9]+/
FALSE - (currently not implemented, but if it is: /false/)
TRUE - (same as above, but /true/)

// The 'names' of variables
IDENTIFIER - /[A-z]+/

// Reserved keywords (Starts the same way as IDENTIFIER, but before the
lexer decides that it is an IDENTIFIER, it checks the current token
against
a map of reserved words.)
VAR
FOR
END
IN
DO
READ
PRINT
ASSERT

// Types (All checked literally, same as above)
INT
STRING
BOOL

// Checked using additional lookahead character
ASSIGN
SPREAD
```

The token names should be pretty much self explanatory, so there is only the corresponding regex given.

A modified context-free grammar suitable for recursive-descent parsing (eliminating any LL(1) violations); modifications must not affect the language that is accepted.

There are some grammar rules sketched in the 'grammar.txt' -file.

---

abstract syntax trees (AST), i.e., the internal representation for Mini-PL programs alternatively give a syntax-based definition of the abstract syntax

---

Error handling approach and solutions used in your Mini-PL implementation (in its scanner, parser, semantic analyzer, and interpreter).

Scanner/Lexer

The scanner keeps track of the current line in the source code and will print the line number where the error was detected, along with a message. For example:

```
[line 8] Error : Unexpected token: .
```

The scanner will keep scanning after that token/character, and it can report multiple errors (everything it comes across in the input string). It also still outputs the scanning result, so if there is any reason to, it can be used by a parser. The output will omit the erroneus characters, and only has the valid tokens in it, so it might actually be a valid program that still does something meaningful, although there is no guarantee of that by the lexer.

Parser

The parser has a 'panic mode' that it will enter when it comes across a syntax error. It will try to find the next place where it can continue parsing, (called synchronizing the parser) while avoiding raising errors that are actually caused by the last error it encountered.

This enables the parser to correctly inform about errors further down the code, so the user will see all relevant error messages currently applying to their source code. It will use the line numbers that have been saved into the Tokens to give some information to the user about where the error lies.

Semantic analysis

Interpreter

The interpreter will check types, for example arithmetic operands have to be numbers. It will raise an error to inform the user of the mistake and line number.

Work log

Date	**Time**	**Work hours**	**Description**
xx.01.2022	xx:xx - xx:xx	~10h	Placeholder for work done on January (Scanner/Lexer impl)
xx.02.2022	xx:xx - xx:xx	~10h	Placeholder for work done on February (Scanner/Lexer impl)

Date	Time	Work hours	Description
11.03.2022	11:00 - 15:00	~4h	Started writing documentation and building this work log
12.03.2022	12:00 - 18:xx	~5h	Fixed bug in Lexer, add error handling documentation for Lexer, add expressions
13.03.2022	11:40 - xx:xx	~7h	The Parser and Interpreter parts are working. They can do basic arithmetic, print literals or variables, variables can be defined by the user, variables can be reassigned by the user

Testing

test.txt -file was used to test that the Lexer is working as expected. It contains a few simple expressions in the MiniPL language. The Lexer print outs each token and it's value.

Contents of text.txt:

```
23 + 4;
print "this should be printed";
print 25 + 25 + 50;

var first := 1;
var second := 2;
print first + second;
first := 5;
print first + second;
```

Missing features

I did not implement the type system, or the control structures, such as loops as I ran out of time.