Project 1: Scanner Visual

Tyler Burkett

For this project, you will be working on creating a scanner by hand for a more robust programming language. The language you will work on is a tweak of the B-Minor language in your text book. The provided code for this project should lead you to develop a scanner that can be used for the next project with little/no modification. The next project will have you use this scanner with a parser to generate a representation of the code as a data structure.

1 B-Sharp Language

The B-Sharp language is a superset of the B-Minor language in the text book. In other words, all aspects of the B-Minor language as described in the textbook also apply to B-Sharp, with some additions. The following are the list of features introduced with B-Sharp. For this project, we will only focus on the tokens needed by the language.

- An additional `float` type. There are also associated float literals. These literals are of the form "$d.d$", where d is one or more digits.
- An additional operator `as` for explicit casting. The casting operator has precedence between multiplication/division/modulo and exponentiation.

```
var: integer = 1;
var2: float = var as float;
```

- For logical operators, there are also corresponding keywords you can use in place of the symbols: `not` for `!`, `or` for `||`, and `and` for `&&`.

```
a: boolean;
b: boolean;
c: boolean;
...
classic_expr: boolean = a && !b || c;
new_expr: boolean = a and not b or c;
```

- (Bonus) A new looping statement; `loop-while-repeat`.

```
bool_val: boolean;
...
loop
    ...
while ( bool_val )
    ...
repeat
```

  The `while` part is optional, resulting in a `loop-repeat` statement that creates an infinite loop.

```
loop
    ...
repeat
```

2 Background Information

You will be provided with the following files. Your job is to complete both the `main.cpp` and `scanner.cpp` file to completely define a scanner for B-Minor tokens and generate a listing of tokens.

- main.cpp
- scanner.hpp
- scanner.cpp
- token.hpp

You should only need to modify `main.cpp` and `scanner.cpp` files for this project. The others are included to provide type definitions for the scanner and tokens.

To use the `Scanner` class, you will need to pass it an `istream` when constructing it, then call `next` to get a token. The `Scanner` class internally handles matching tokens using regular expressions, handling cases where multiple/none match, continuing past certain cases like whitespace, and so on.

```
    ifstream input_file;
    ...
    Scanner test_scanner = Scanner(input_file);
    token_ty next_token = test_scanner.next();

    while(not match<end_of_input>(next_token) ) {
      ...
      next_token = test_scanner.next();
    }
```

The resulting token type `token_ty` is actually a *variant*, essentially a type-safe union container. Variants allocate a single space in memory to hold multiple types of object. Each type of token is it's own struct, and the variant

```
using token_ty = std::variant<end_of_input, int_token, id_token, ...>
token_ty token{};

// Create an 'int_token' object directly in
// the memory space 'token' occupies
token.emplace<int_token>(int_token{1});

// Get a copy of the 'int_token' stored in 'token'.
// Throws an exception if 'token' is not storing an 'int_token'
auto int_token_val = std::get<int_token>(token);

// Check what is stored in 'token' at runtime
if (std::holds_alternative<int_token>(token)) {
  ...
}

// Perform different versions of a function/callable object
// on the variant 'token' without having to directly check
// what it's holding first.
struct stringify_visitor {
  std::string operator()(int_token& i) { ... }
  ...
}
auto token_string = std::visit(stringify_visitor(), token);
```

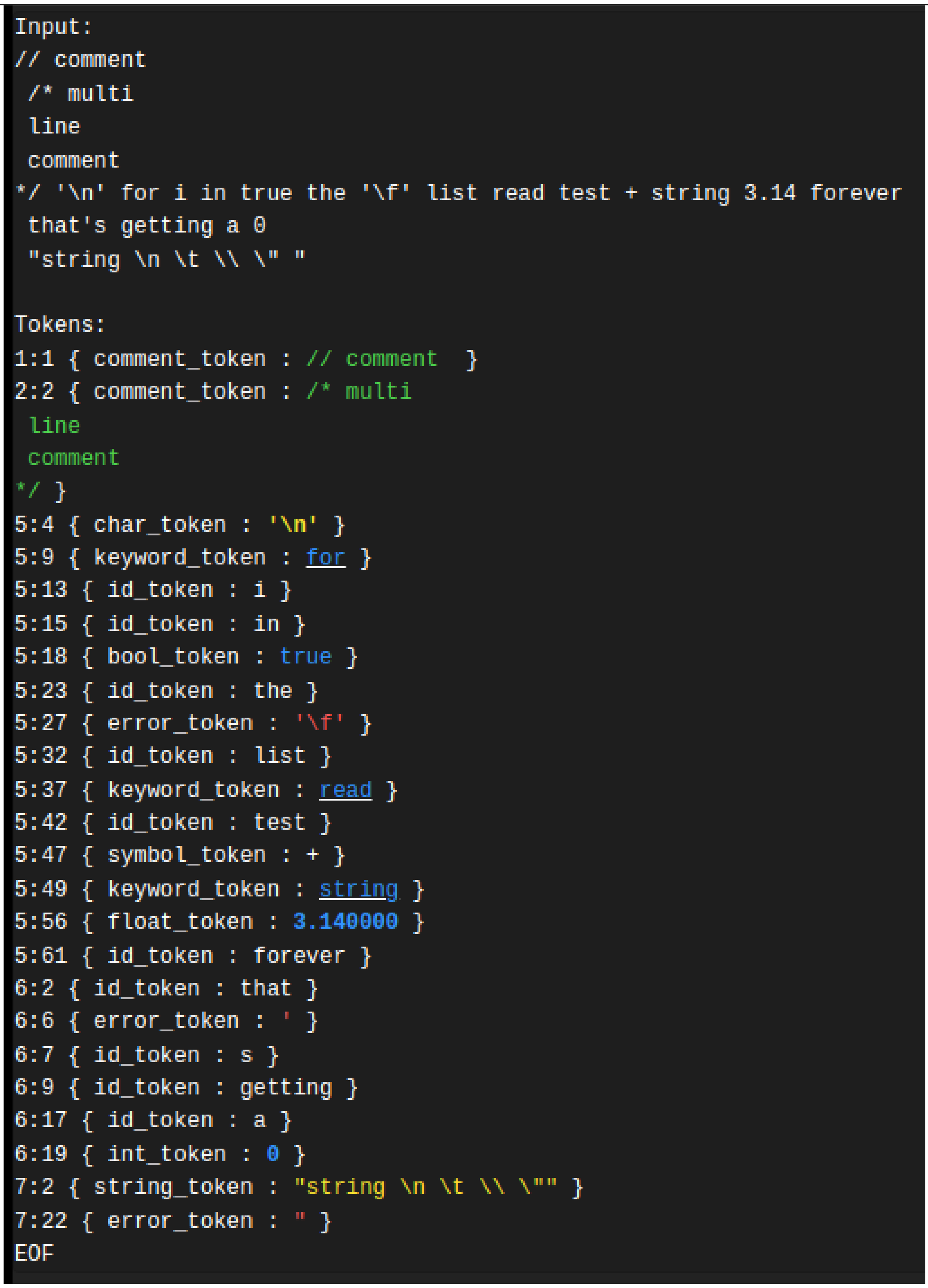
As for the `Scanner` itself, you will need to use the `regex` class in the std C++ library. With it, you will be able to easily create regular expressions for each type of token in the B-Minor language. To make writing the regular expression, you should use *raw string literals*; string literals of the form `R"(`*`text`*`)"` that capture all characters in the string without escapeing. Otherwise, you'd have to double-escape certain regex expressions (e.x. `\\\\` for a literal backslash).

```
std::string input = ...;
std::regex simple_regex(R"(([a-z]+)\\([a-z]+))");
std::smatch match_result;

// Match the entire string to the regex
if (std::regex_match(input, match_result, simple_regex)) {
    std::cout << "Matched!" << std::endl;

    // The resulting 'match_result' object contains
    // [0] : The full text matched
    // [1...] : Text captured in parenthesized parts of regex
    std::cout << "# Components = " << match_result.size() << std::
       ↪ endl;
    std::cout << "Text: " << match_result[0] << std::endl;
    std::cout << "First Half: " << match_result[1] << std::endl;
    std::cout << "Second Half: " << match_result[2] << std::endl;
}

// Match a substring of the input to the regex
auto text_copy = text;
while (std::regex_search(text_copy, match_result, simple_regex)) {
    // 'match_result' works the same as with match
    ...


    // 'match_result' also tracks what portion of the text was
    // skipped (prefix) and are yet to reached by the regex
    // (suffix)
    std::cout << "Skipped: "
              << match_result.prefix()
              << std::endl;
    // Remove the already searched text to
    // continue looking for more matches
    text_copy = match_result.suffix();
}

// Replace all substrings that match the regex
// with the substitute string
std::regex_replace(input, simple_regex, "x\\y");
```

When compiled, the program should be executable as follows.

```
./Scanner.exe PROGRAM_FILE
```

Below is an example of the output as viewed in a browser.



```
Input:
// comment
 /* multi
 line
 comment
*/ '\n' for i in true the '\f' list read test + string 3.14 forever
 that's getting a 0
 "string \n \t \\ \" "

Tokens:
1:1 { comment_token : // comment  }
2:2 { comment_token : /* multi
 line
 comment
*/ }
5:4 { char_token : '\n' }
5:9 { keyword_token : for }
5:13 { id_token : i }
5:15 { id_token : in }
5:18 { bool_token : true }
5:23 { id_token : the }
5:27 { error_token : '\f' }
5:32 { id_token : list }
5:37 { keyword_token : read }
5:42 { id_token : test }
5:47 { symbol_token : + }
5:49 { keyword_token : string }
5:56 { float_token : 3.140000 }
5:61 { id_token : forever }
6:2 { id_token : that }
6:6 { error_token : ' }
6:7 { id_token : s }
6:9 { id_token : getting }
6:17 { id_token : a }
6:19 { int_token : 0 }
7:2 { string_token : "string \n \t \\ \"" }
7:22 { error_token : " }
EOF
```

3 Requirements

3.1 Scanner Requirements

For this scanner, you may assume that a valid B-Sharp program will only contain ASCII characters. You will need to implement scanner rules for the following.

- Keywords
- Operator
- Identifiers
- Comments
- Literals
- Whitespace

Keywords are words specifically reserved by a language to denote certain language structures. These words are not valid identifiers. The following are keywords for B-Sharp.

```
as array boolean char else float for function if integer print return
  ↪  string void and not or
```

Operators are special sequences of symbols used to denote language structures (usually expressions), similar to keywords. The following are operators for B-Sharp.

```
: , ; = [ ] ( ) ++ -- - ! ^ * / % + - < <= > >= == != && ||
```

Identifiers are words which refer to specific elements/objects in a program, primarily variables and functions. A valid B-Sharp identifier is a letter ("a" - "z", lower and uppercase) followed by an arbitrary sequence of letters, numbers, and underscores.

Comments are sections of text that do not translate to a structure in the program. B-Sharp uses C-style comments (// and /* */) to denote comments.

Literals are constant values for the different data types in the programming language. The following are the different types of literals and their formatting.

- Boolean: The words "true" and "false"
- Integer: A sequence of 1 or more numbers (0 through 9).
- Character: A single ASCII character (excluding the single quote) or escape sequence contained in single quotes. A valid escape sequence consist of a backslash followed by one of the following.
  - t
  - n
  - '
- String: A sequence of ASCII characters (excluding the double quote) or escape sequences contained in double quotes. A valid escape sequence consist of a backslash followed by one of the following.
  - t
  - n
  - "
- Floating-Point: A sequence of 1 or more numbers, followed by a period, followed by a sequence of 1 or more numbers.

Whitespace consists of invisible characters in the text of a program. These include newline, space, and tabs, as well as (though less commonly used) linefeed ("\r") and feed ("\f").

For each token, you will need to implement code to perform a certain set of actions, most of which will result in you creating one of the provided token structures.

- Create regular expressions to match the following:
  - A multiline comment
  - A single line comment
  - All operators
  - All keywords
  - An identifier
  - An integer literal

- A float literal
- A string literal
- A integer literal
- A boolean literal
- A "bad" char literal (i.e. one with multiple characters in it and/or invalid escapes)
- A sequence of one or more whitespace characters

Make sure that every regular expression is matching only the beginning of the text. Everytime a token is needed, each regex will be ran on the input text to determine what token is coming up next.

- For the `next` method of `Scanner`, write code to complete it so that it will do the following:
  - When calling `next`, every token except whitespace will return some kind of token. For whitespace, the method will need to consume the matched text and try matching another token again. Define and call the `default_ignore` method for this case.
  - In most cases, creating the token is the same procedure, just with a different token type. Define and call the templated `default_rule` method for these cases.
  - For literals, you will need to process the text matched to convert it to an actual value of that type. Strings and characters will need escaped characters replaced with the actual character value, along with removing the quotes that will be matched alongside their text contents. Do this by defining and calling the `process_X_lit` methods when the corresponding literal is matched.
  - If a bad char literal is matched or none of the regular expressions are matched, return an error token. For the latter, simply extract a single character for the error token.
  - Everytime one of regular expression matches, check that any previous matches that may have been made are shorter than it before creating a token.
  - When creating tokens, make sure to use the provided `end_position_of` method to calcutate the end position of the token. Any time

anything is matched (including when we would produce an error token), make sure to remove the matched text and update `current_position` using the provided `position_after` method before returning the token.

3.2 Main Code Requirements

For the `main.cpp` file, you will need to add code to create an instance of the scanner and use it to scan a text file. Everytime a token is produced, until `end_of_input` is produced, you will need to print out a line on the console like in the example output. Code is provided to handle coloring text on the console, but you will have to complete the `token_stringify` struct to handle each type of token.

- Each token should be printed in the format:

  ```
  <line>:<column> { <token_type> : <value> }
  ```

- The value of each token should be formated as follows:
  - comments: green text
  - string literal: yellow text
  - character : bright yellow text
  - identifiers/symbols: no formatting
  - keywords: blue and underlined text
  - int/float literals: bright blue text
  - error tokens: red text
  - end of inpute: return the string "EOF" (no formatting)

4 Submission

The submission should be a zip file containing at minimum the following files.

- Makefile
- scanner.cpp

- main.cpp
- README
- tests/: a directory containing test input
- Any of the other code provided as part of the project that you used

You are free to modify any of the code provided for this project that you aren't already required to modify for this project. Keep in mind that any changes you make may affect its interoperability with code provided in future projects, and that the behavior of programs are expected to match **exactly** with what is documented in the project handout.

The Makefile should be able to perform at minimum the following commands.

- `make`. Compile the scanner program. Do NOT execute it.
- `make run IN=FILE`. Compile and execute the scanner program, give the file path for argument `IN`.
- `make clean`. Clean up all files that were generated from running make.

For this project, make sure that your Makefile compiles using C++20.

5 Resources

- Makefile manual
- CPPReference, specifically the page on Regex syntax