Compiler Theory and Practice

Coursework

Juan Scerri
123456A

March 31, 2024

A coursework submitted in fulfilment of study unit CPS2000.

Contents

Listings

Report

1 | Lexer

1.1 | Design & Implementation

The lexer was split into three-main components. A DFSA class, a generic table-driven lexer, and a lexer builder.

The DFSA

The DFSA class is an almost-faithful implementation of the formal concept of a DFSA. Listing 1, outlines the behaviour of the DFSA. Additionally it contains a number of helper functions which facilitate getting the initial state and checking whether a state or a transition category is valid. These helpers specifically, `getInitialState()` is present since after building the DFSA there is no guarantee the initial state used by the user will be the same.

```
14 class Dfsa {
15    public:
16     Dfsa(
17         size_t noOfStates,
18         size_t noOfCategories,
19         std::vector<std::vector<int>> const&
20             transitionTable,
21         int initialState,
22         std::unordered_set<int> const& finalStates
23     );
24
25     [[nodiscard]] int getInitialState() const;
26
27     [[nodiscard]] bool isValidState(int state) const;
```

```
28      [[nodiscard]] bool isValidCategory(int category) const;
29
30      [[nodiscard]] bool isFinalState(int state) const;
31
32      [[nodiscard]] int getTransition(
33          int state,
34          std::vector<int> const& categories
35      ) const;
36
37    private:
38      const size_t mNoOfStates;       // Q
39      const size_t mNoOfCategories;   // Sigma
40      const std::vector<std::vector<int>>
41          mTransitionTable;                          // delta
42      const int mInitialState;                       // q_0
43      const std::unordered_set<int> mFinalStates;    // F
44  };
```

Listing 1: DFSA Class Declaration (lexer/DFSA.hpp)

The only significant difference is the `getTransition()` functions. In fact, it accepts a vector of transition categories instead of a single category.

This is because a symbol e.g. 'a', '9' etc, might be valid for multiple categories. For instance 'a' is considered to be both a letter and a number in hexadecimal.

The DFSA for accepting the micro-syntax PArL is built as follows.

Let $\mathfrak{U}$ be the set of all possible characters under the system encoding (e.g. UTF-8).

The will use the following categories:

- $L \coloneqq \{\mathtt{A}, \ldots, \mathtt{Z}, \mathtt{a}, \ldots, \mathtt{z}\}$
- $D \coloneqq \{\mathtt{0}, \ldots, \mathtt{9}\}$
- $H \coloneqq \{\mathtt{A}, \ldots, \mathtt{F}, \mathtt{a}, \ldots, \mathtt{f}\} \cup D$
- $S \coloneqq \{\alpha \in \mathfrak{U} \colon \alpha \text{ is whitespace}\} \setminus \{\mathtt{LF}\}$

Note: LF refers to line-feed or as it is more commonly known '\n' i.e. new-line.

Together these categories form our alphabet Σ:

$$\Sigma \coloneqq L \cup D \cup S \cup \{\texttt{.}, \texttt{\#}, \texttt{\_}, \texttt{(}, \texttt{)}, \texttt{[}, \texttt{]}, \texttt{\{}, \texttt{\}}, \texttt{*}, \texttt{/}, \texttt{+}, \texttt{-}, \texttt{<}, \texttt{>}, \texttt{=}, \texttt{!}, \texttt{,}, \texttt{:}, \texttt{;}, \text{LF}\}$$

Now, the following drawing describe the transitions of the DFSA. For improved readability the DFSA has been split across mulitple drawings. Hence, in each drawing initial state 0 refers to the *same* initial state (a DFSA has one and only one initial state).

Additionally, each final state is annotated with the token type it should produce.

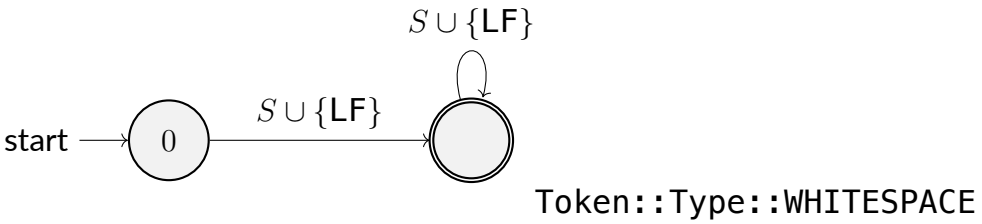

Figure 1: States & transitions for recognising whitespace

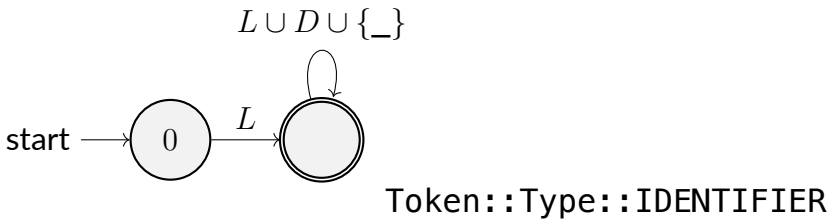

Figure 2: States & transitions for recognising identifiers/keywords

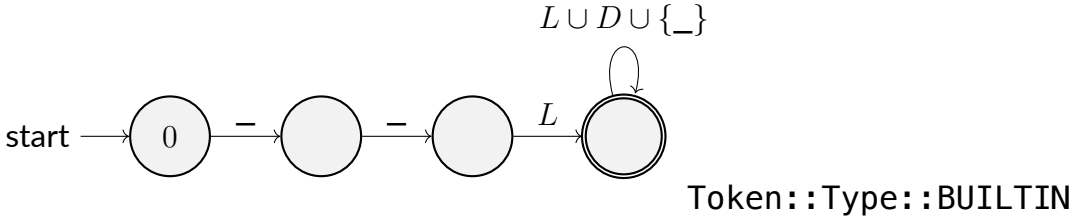

Figure 3: States & transitions for recognising builtins

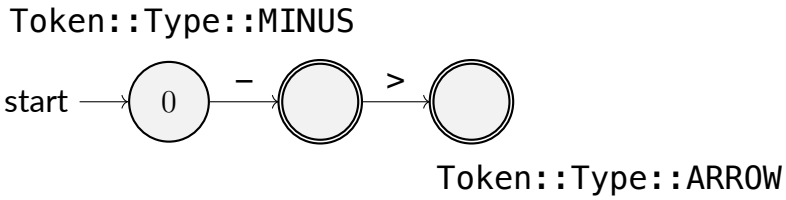

Figure 4: States & transitions for recognising minus and arrow (–>)

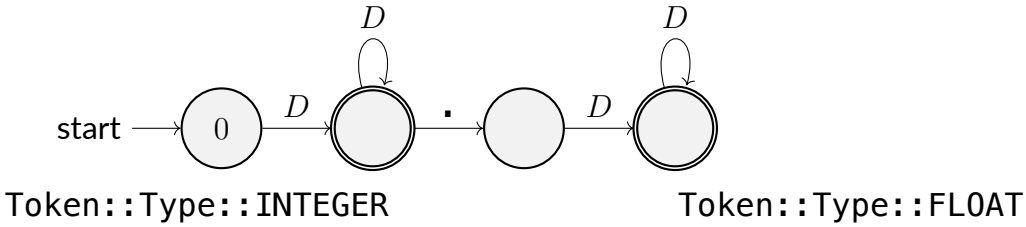

Figure 5: States & transitions for recognising integers and floats

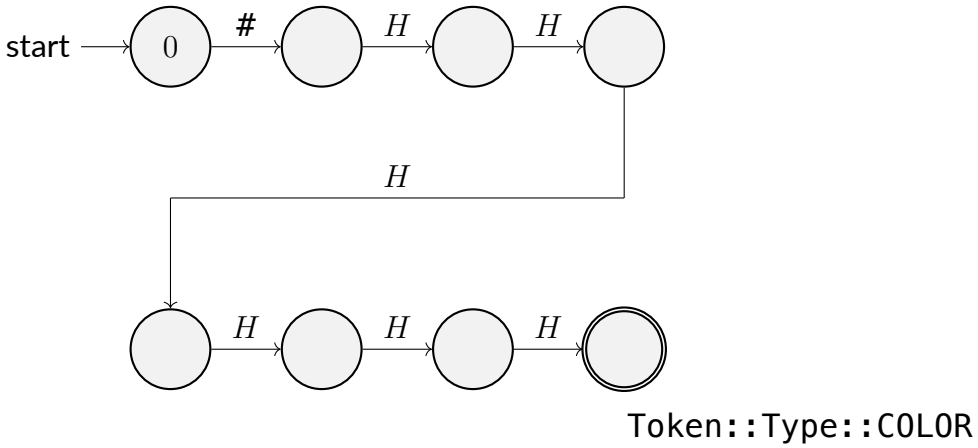

Figure 6: States & transitions for recognising colours

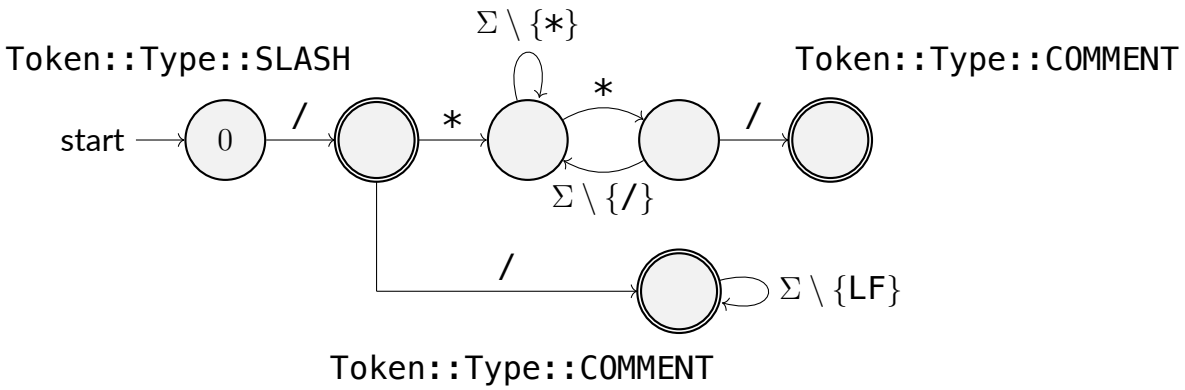

Figure 7: States & transitions for recognising slashes and comments

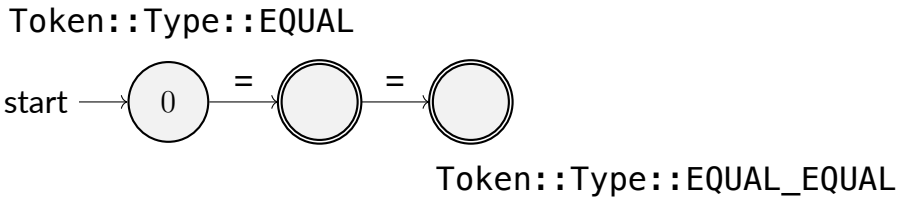

Figure 8: States & transitions for assign and is equal to

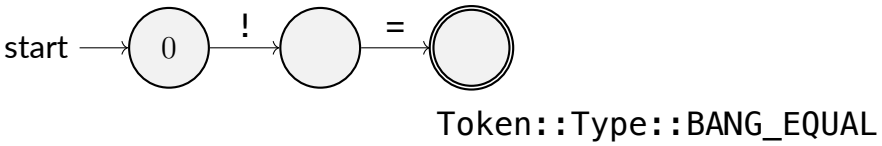

Figure 9: States & transitions for not equal to

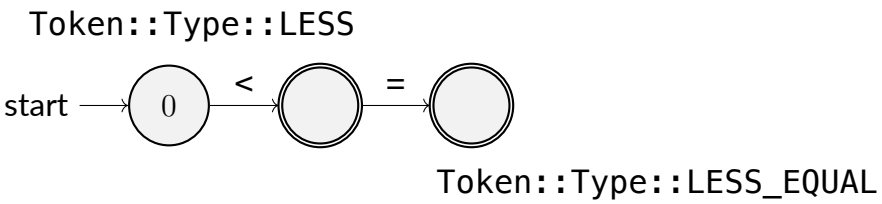

Figure 10: States & transitions for less than and less than or equal to

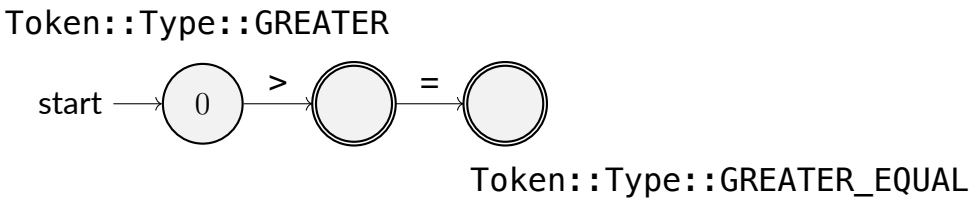

Figure 11: States & transitions for greater than and greater than or equal to

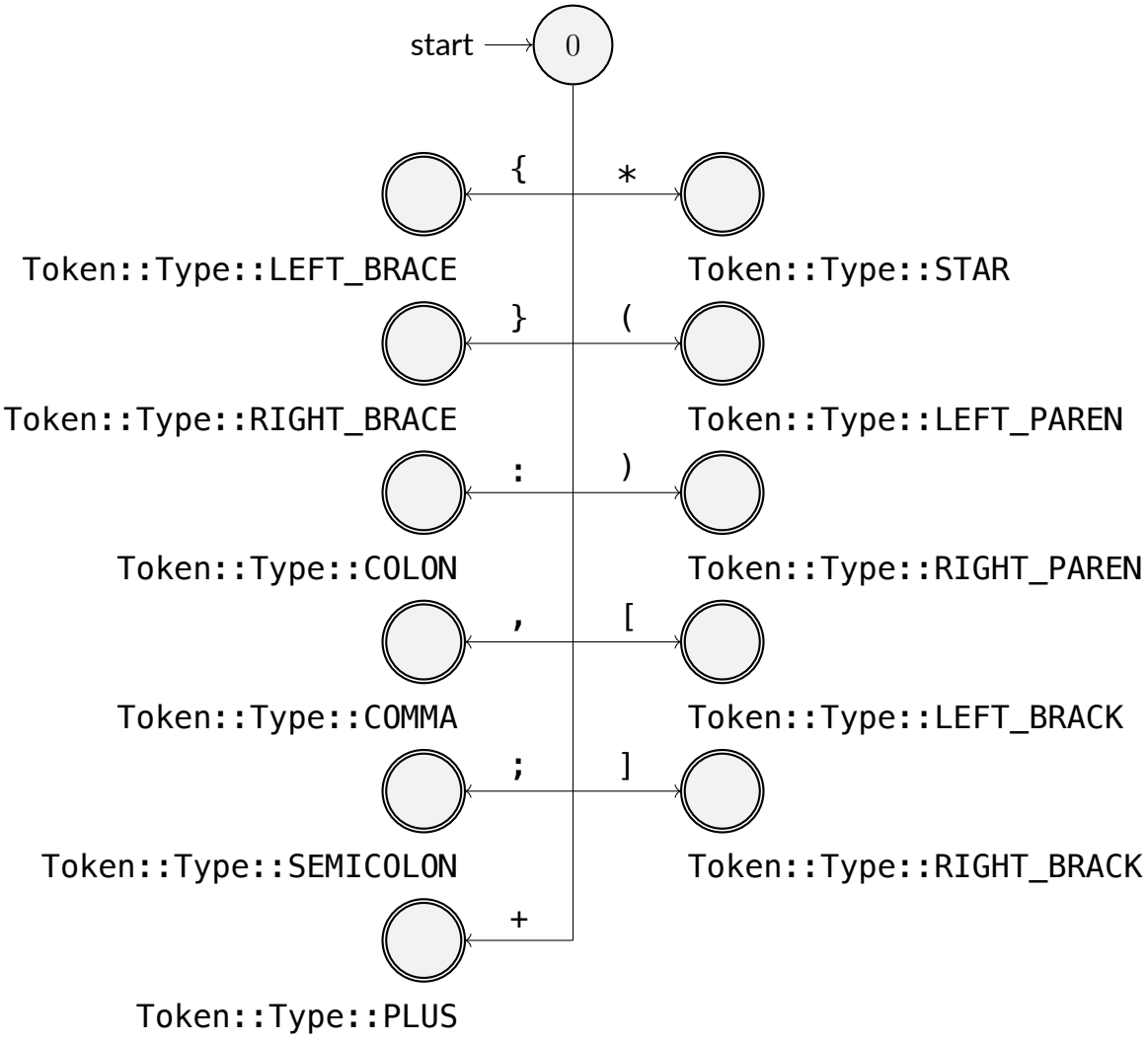

Figure 12: States & transitions for single letter tokens

The Builder & Director

Each sequence of states present is directly represented in code within the `LexerDirector` using methods provided by the `LexerBuilder`.

```
297         .addTransition(34, STAR, 36)
298         .addComplementaryTransition(36, STAR, 36)
299         .addTransition(36, STAR, 37)
300         .addComplementaryTransition(37, SLASH, 36)
301         .addTransition(37, SLASH, 38)
302         .setStateAsFinal(38, Token::Type::COMMENT);
303
304     // builtin
305     builder.addTransition(0, UNDERSCORE, 39)
306         .addTransition(39, UNDERSCORE, 40)
307         .addTransition(40, LETTER, 41)
308         .addTransition(41, {LETTER, DIGIT, UNDERSCORE}, 41)
309         .setStateAsFinal(41, Token::Type::BUILTIN);
```

Listing 2: Code specification of the comments in the `LexerDirector` (lexer/LexerDirector.cpp)

The `LexerBuilder` keeps track of these transitions using less efficient data structures such as hash maps (`std::unordered_map`) and sets (`std::unordered_set`).

Then the `build()` method processes the user defined transitions and normalises everything into a single transition table for use in a DFSA. Additionally, it also produces two other artefacts. The first is called `categoryIndexToChecker`. It is a hash map from the index of a category to a lambda function which takes a character as input and returns true or false.

The lambdas and the category indices are also registered by the user. See Listing 3 for a registration example. Additionally, the category indices although they are integers for readability they are defined as an enumeration.

```
51          .addCategory(
52              HEX,
53              [](char c) -> bool {
54                  return ('0' <= c && c <= '9') ||
55                         ('A' <= c && c <= 'F') ||
56                         ('a' <= c && c <= 'f');
57              }
```

```
58          )
```

Listing 3: Registration of the hexadecimal category checker (lexer/LexerDirector.cpp)

The second artefact produced by the builder is also a hash map from final states to their associated token type.

The transition table is then passed onto the DFSA. And the DFSA, and the two artefacts are passed onto the Lexer class.

```
210     // create dfsa
211     Dfsa dfsa(
212         noOfStates,
213         noOfCategories,
214         transitionTable,
215         initialStateIndex,
216         finalStateIndices
217     );
218
219     // create lexer
220     Lexer lexer(
221         std::move(dfsa),
222         std::move(categoryIndexToChecker),
223         std::move(finalStateIndexToTokenType)
224     );
```

Listing 4: Constructions of the Lexer (lexer/LexerBuilder.cpp)

The Actual Lexer

The lexer's core is as was described during the lectures and the core/main method is `simulateDFSA()`.

It also has a number of very important auxiliary methods and behavioural changes. Specifically, the `updateLocationState()`, see Listing 5, is critical for providing adequate error messages both during the current stage and for later stages. This function is called every time a lexeme is consumed allowing the lexer to keep track of where in the file it is, in terms of lines and columns.

```
110 void Lexer::updateLocationState(std::string const& lexeme) {
```

```
111     for (char ch : lexeme) {
112         mCursor++;
113 
114         if (ch == '\n') {
115             mLine++;
116 
117             mColumn = 1;
118         } else {
119             mColumn++;
120         }
121     }
122 }
```

Listing 5: The `updateLocationState()` lexer method (lexer/Lexer.cpp)

Additionally, if an invalid / non-accepting state is reached the invalid lexeme is consumed and the user is warned, see Listing 6. After this the lexer, is left in a still operational state. Hence, `nextToken()` can be used again.

This is critical to provide users of the PArL compiler with a list of as many errors as possible, since it would be a bad experience to have to constantly run the PArL compiler to see the next error.

```
56     if (state == INVALID_STATE) {
57         mHasError = true;
58 
59         fmt::println(
60             stderr,
61             "lexical error at {}:{}:: unexpected "
62             "lexeme '{}'",
63             mLine,
64             mColumn,
65             lexeme
66         );
67     } else {
68         try {
69             token = createToken(
70                 lexeme,
71                 mFinalStateToTokenType.at(state)
72             );
73         } catch (UndefinedBuiltin& error) {
74             mHasError = true;
```

```
75
76              fmt::println(
77                  stderr,
78                  "lexical error at {}:{}:: {}",
79                  mLine,
80                  mColumn,
81                  error.what()
82              );
83          }
84      }
```

Listing 6: Error handling mechanism in the `nextToken()` lexer method (lexer/Lexer.cpp)

Hooking up the Lexer to the Runner

The `Runner` class is the basic structure which connects all the stages of the compiler together together.

In this case the Runner passes in a reference to the lexer into the parser, this allows the parser to request tokens and they are computed on demand improving overall performance. Additionally, this has the benefit of allowing the parsing of larger and multiple files since, the parser is no longer limited by the amount of usable memory, since it does not need to load the whole file.

However, in this case no such optimisation is present.

```
22 Runner::Runner(bool dfsaDbg, bool lexerDbg, bool parserDbg)
23     : mDfsaDbg(dfsaDbg),
24       mLexerDbg(lexerDbg),
25       mParserDbg(parserDbg),
26       mLexer(LexerDirector::buildLexer()),
27       mParser(Parser(mLexer)) {
28 }
```

Listing 7: The Runner constructor passes `mLexer` into the Parser constructor (runner/Runner.cpp)

2 | The AST & Parsing

2.1 | Modifications to the EBNF

3 | Attributions

- Sandro Spina for the brilliant description of table-driven lexers
- Robert Nystrom and his great book Crafting Intepreters for a great outline for parsing and error recovery/management for languages which support exceptions