COMP 442 – Compiler Design
Assignment 1 – **Lexical Analyzer**

Jafar Abbas (26346650)
January 30, 2017

Introduction

This report documents the design and implementation of a scanner for a programming language whose lexical specifications are given below.

Requirements

1. The scanner identifies and outputs tokens (valid words and punctuation) in the source program (text file).

2. When called, the lexical analyzer should extract the next token from the source program.

3. The tokens should be later usable to verify the program is syntactically valid (saved in a data structure containing the token type, value (lexeme) and its location in the source code.

4. Even if the input does not form a correct program, the lexical analyzer should be able to output a token.

5. The lexical analyzer should pass a wide variety of valid and invalid test cases.

6. The token stream is able to be saved to an output file.

7. A driver should repeatedly call the lexical analysis function and print the type of each token.

8. An output file should contain representative error messages each time an error is encountered with its location (line number) in the source program.

9. The lexical analyzer will be blind to upper/lower case characters.

Analysis and Design

Below are the lexical elements as given by the instructor. In this section I will identify the tokens from the following regular expressions and correct any ambiguities.

Atomic lexical elements of the language:

```
id          ::=   letter alphanum*
alphanum    ::=   letter | digit | _
num         ::=   integer | float
integer     ::=   nonzero digit* | 0
float       ::=   integer fraction
fraction    ::=   .digit* nonzero | .0
letter      ::=   a..z |A..Z
digit       ::=   0..9
nonzero     ::=   1..9
```

Operators, punctuation and reserved words:

```
 ==     +      (      if
 <>     -      )      then
 <      *      {      else
 >      /      }      for
 <=     =      [      class
 >=     and    ]      int
 ;      not    /*     float
 ,      or     */     get
 .             //     put
                      return
                      program
```

Ambiguities

Upon reading the above, some questions come to mind that require clarification:

1) What about negative numbers?
2) What constitutes delimiters / whitespace?
3) “nonzero” is a subset of “digit”
4) “num” is redundant and not atomic
5) “alphanum” intersects with “id”, is redundant and not atomic
6) Reserved word “float” same as lexical element “float”
7) “.” used in “fraction” and as punctuation
8) “fraction” is only relevant in the context of floating points and is not significant on its own.

Addressing the ambiguities

1) <u>What about negative numbers?</u>

   It will be left up to the syntax analyzer to translate "minus" before a number as a negative number.

2) <u>What constitutes delimiters / whitespace?</u>

   ```
   add:
       whitespace := (blank | tab | newline)+
   ```

   The whitespace can sometimes be an important element of syntax in the language and used as delimiters.

3) <u>"nonzero" is a subset of "digit"</u>

   ```
   rename: "nonzero" to "digit"
   add:
       zero:= 0
   modify:
       integer := digit (digit | zero)* | zero
       fraction := .(digit | zero)* digit | .zero
   ```

   Since there are common elements between the digit and nonzero terminals, this would lead to an arbitrary choice by the compiler. Instead I have clearly separated them, and in the end they will be turned into integers or floating points.

4) <u>"num" is redundant and not atomic</u>

   ```
   remove: "num"
   ```

   Since this element is not adding anything to the language, it can simply be removed.

5) <u>"alphanum" intersects with "id", is redundant and not atomic</u>

   ```
   remove: "alphanum"
   modify:
       id := letter (letter | digit | _)*
   ```

   Removing it does not diminish the expressive power of our language, we just use more atomic elements instead.

6) Reserved word “float” same as lexical element “float”

modify:

floating_point := integer fraction

This change is just a simple renaming to avoid confusion.

7& 8) “.” used in “fraction” and as punctuation, “fraction” is only relevant in the context of floating points and is not significant on its own.

We can look ahead at next character then backtrack to prioritize “fraction” over “period”

Remove: “fraction” – This solves the problem, reduces complexity and does not reduce our language. Instead we merge fraction into floating point:

floating_point := integer (.(digit | zero)* digit) | .zero

Methods used to apply changes

To determine where the ambiguities and redundancies existed in the specifications, I started by asking myself “what if…?” and “what about…?” questions as I read them.

If it was a question of redundancy, I isolated the elements responsible which don’t seem atomic and tried systematically to remove one rule then generate the language with the rest of the rules. If the same language can be generated by removing a specific rule, then it can be removed because it is in fact redundant.

In the case where one element was a subset of another, I separated the terminals into mutually exclusive elements. This is because the compiler should not make arbitrary selections.

These changes do not reduce the expressive power of our language while in fact reducing complexity/size and eliminating ambiguities.

Final lexical specifications

After the previous modifications to correct for ambiguities and redundancy, this is the final list of lexical elements:

```
id               ::=   letter (letter | digit | _ )*
integer          ::=   digit (digit | zero)* | zero
floating_point   ::=   integer (.(digit | zero)* digit)|.zero
letter           ::=   a..z | A..Z
digit            ::=   1..9
zero             ::=   0
whitespace       ::=   (   | \t | \n )+
```

The operators, punctuation and reserved words are left as they are:

```
==     +      (      if
<>     -      )      then
<      *      {      else
>      /      }      for
<=     =      [      class
>=     and    ]      int
;      not    /*     float
,      or     */     get
.            //     put
                    return
                    program
```

Tokens

In this section, I translate the final specifications of lexical elements in our language into tokens recognized by the lexical analyzer.

Lexeme	Token	Attribute
An identifier	id	value
An integer number	integer	value
A floating point number	floating_point	value
Whitespace (space, tab, newline)	ws	-
=	assign	-
==	relop	EQ
<>	relop	NE
<	relop	LT
>	relop	GT
<=	relop	LE
>=	relop	GE
;	semicolon	-
_	underscore	-
‘	apostrophe	-

.	period	-
+	arop	ADD
-	arop	SUB
*	arop	MUL
/	arop	DIV
and	bop	AND
not	bop	NOT
or	bop	OR
)	close_paren	-
(	open_paren	-
}	close_brace	-
{	open_brace	-
]	close_sq_bracket	-
[	open_sq_bracket	-
/*	open_comment	-
*/	close_comment	-
//	line_comment	-
if	reserved	IF
then	reserved	THEN
else	reserved	ELSE
for	reserved	FOR
class	type_decl	CLASS
int	type_decl	INT
float	type_decl	FLOAT
get	reserved	GET
put	reserved	PUT
return	reserved	RETURN
program	reserved	PROGRAM
Unexpected symbol	unexpected	-

Comments:

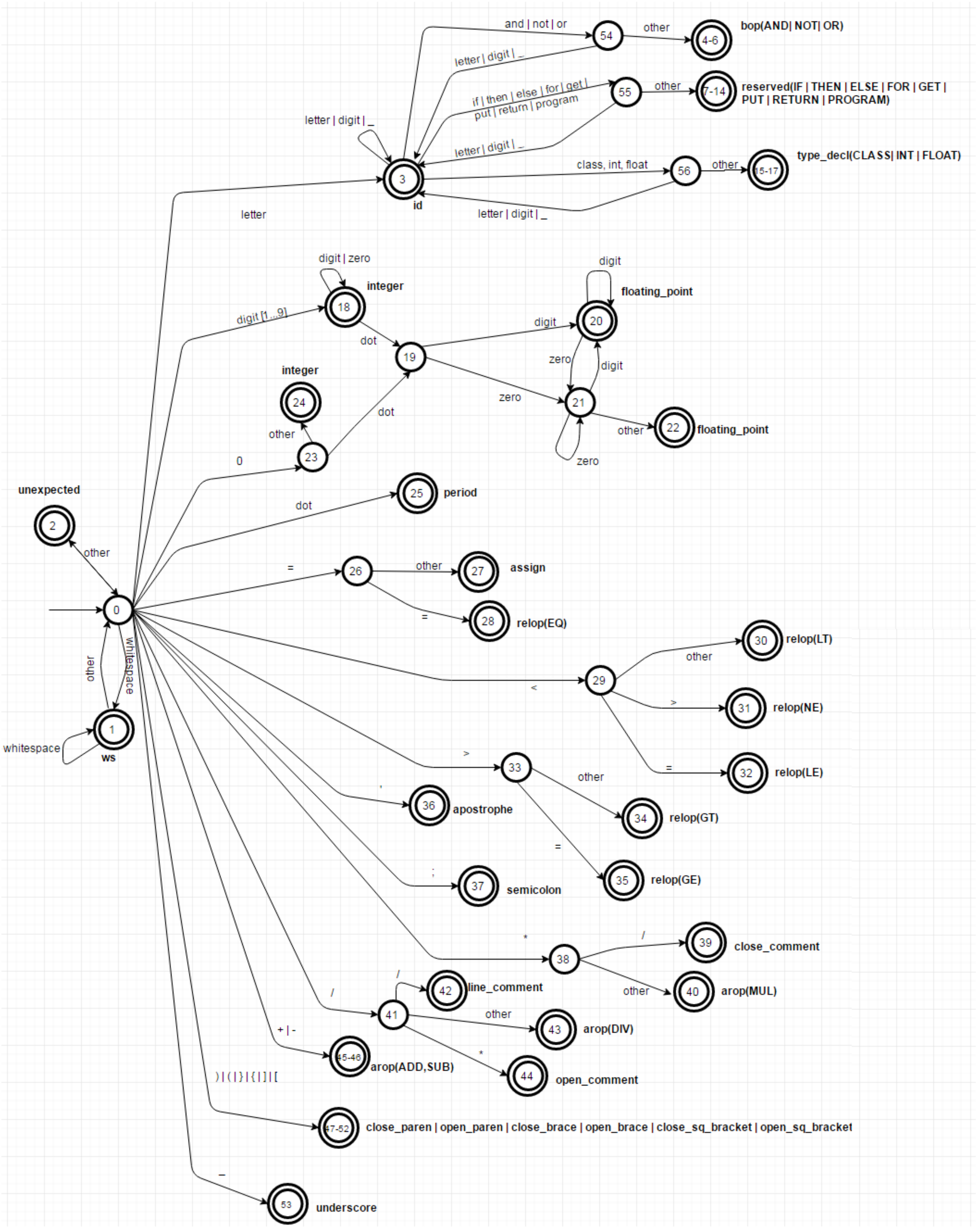
- The attribute column specifies a unique feature needed to distinguish the token.
- Digit refers to [1…9].
- The elements "zero", "letter", "digit", "fraction" are typically constituents of other combinational tokens so they were not given their own token.
- "Whitespace" token will likely be discarded in the end, but that will depend on the syntax later defined so it's kept as a final state for now.
- In the DFA, due to the large size of tokens, some transitions were compressed; the corresponding state is in the left-to-right order in the bolded text.

Deterministic Finite Automata

To generate the DFA seen on the next page, I did not follow the generic steps of deriving DFA from regular expressions (i.e. Thompson's construction, Rabin-Scott power-set construction). I did not go through the process of generating NDFAs, then translating that into a DFA.

Instead, I went over each element from the lexical specifications and selected the important tokens and directly translated the symbols into a DFA using my human brain.

Through iteration, by looking from the perspective of one character at a time and considering the "look-ahead" symbol, I've identified the terminal states and intermediate non-terminal states that lead to correctly and uniquely identifying each token.



and | not | or
other
bop(AND| NOT| OR)
letter | digit | _
if | then | else | for | get | put | return | program
reserved(IF | THEN | ELSE | FOR | GET | PUT | RETURN | PROGRAM)
class, int, float
type_decl(CLASS| INT | FLOAT)
id
letter
digit [1...9]
digit | zero
integer
dot
digit
floating_point
zero
0
other
period
unexpected
=
assign
relop(EQ)
<
>
relop(LT)
relop(NE)
relop(LE)
relop(GT)
relop(GE)
whitespace
ws
apostrophe
;
semicolon
*
/
close_comment
arop(MUL)
line_comment
arop(DIV)
open_comment
+ | -
arop(ADD,SUB)
) | (| } | { |] | [
close_paren | open_paren | close_brace | open_brace | close_sq_bracket | open_sq_bracket
_
underscore

Implementation

I have selected C++ as the programming language to implement my compiler because it allows powerful memory management and low-level control which is needed for compilers. My knowledge is also strongest in C++ and I'm always trying to improve my skills.

To implement my solution, I have devised the following data structures and classes:

Compiler

This class is rather empty for now, it only has the Tokenizer module. More modules will be added in the future. Its role will be to manage the flow of compilation in the future.

```
class Compiler {
public:
	Compiler();
	~Compiler();

private:
	Tokenizer tokenizer;
	std::string sourceFile;
};
```

Token

The token structure holds the data regarding the tokens, i.e. its name, type and value when applicable. The role of this structure is just a dummy data holder.

The token types have been defined as the following enum used by Toke n and Tokenizer.

```
enum class Types { integer, string, floatingpoint, reserved, type_decl, bop, relop, arop,
none };


struct Token {

	Token(Tokenizer::States _state,
		Types _type = Types::none,
		std::string _value = "");

	Tokenizer::States state;
	std::string name;
	Types type;
	std::string value;
};
```

** Note: minor detail forgotten, including in the token struct the location in the original program, it will be very easy to add.

Tokenizer

This is the grand-daddy class that handles tokenizing. Its functions include:

- Reading the source file
- Managing the Finite State Machine (representation of my DFA discussed in the next section)
- Outputting the token stream
- Outputting clear lexical error messages

```
class Tokenizer {
	friend class Compiler;

public:
	Tokenizer();
	~Tokenizer();
	const char* GetCurrentState();
	void GenerateTokenStream(std::string fileName);
	void OutputTokenStream(std::string fileName = "tokenstream.txt");
	void OutputErrorMessages(std::string fileName = "errors.txt");

protected:
	enum class States {… };
	enum class Triggers {…};
	FSM::Fsm<States, States::s0 /*Starting state*/, Triggers> fsm;

	std::string curr_line;
	int line_pos;
	int line_number;

	bool successful_transition;
	bool final_state;
	bool keyword_hit;

	std::vector<std::string> errorMessages;
	std::vector<Token> tokenStream;
	Token * curr_token;
	States latestCorrectState;

	void initializeFSM();
	static const char* stringifyState(States state);
	Triggers convertCharToTrigger(char ch);
	void isKeywordHit();
	void pushError(std::string comment);
	void pushToken();
	void newToken(States state, bool terminal, Types type, std::string value);
	void updateLastToken(States state, bool terminal, Types type, std::string value);
	void handleReservedWords();
	void resetFSM();

	void addTransition(States from_state,     // from state
	                   States to_state,       // to state
	                   Triggers trigger,      // trigger
	                   std::function<bool(void)> guard_func, // condition to meet
before transitioning
	                   std::function<void(void)> action_func); // function to call
when transitioning from A to B)
};
```

The Tokenizer class is fairly straightforward and the role of each field and method is obvious from their names.
What isn't obvious from the above is the behind-the-scenes behavior:

- It initializes the predefined states in the DFA
- It initializes the predefined transitions (trigger) in the DFA
- Reads the source file line by line, character by character.
- Each character is mapped to a trigger using a simple switch function (and regex used for letters and digits)
- When the trigger executes, the state of the FSM is updated and a pre-defined function attached to each transition is executed.

Implementation Design and Tools

I've touched on the finite state machine (FSM) in the previous section, in this section I will elaborate on how it was achieved.

The FSM is the core of the tokenizer. Its correctness is vital to obtaining correct output. Getting it right on paper through a DFA is tricky in itself however mapping/encoding it into a program is another level of complexity. To simply managing all the states and the even more numerous transitions between these states I had to research for a suitable C++ library. My solution: **fsmcpp**.

This open-source header-only, light-weight library allows for a declarative way to map all the valid states and their transitions (triggers). It also allows me to associate functions to execute when specific triggers lead to transitions and add conditional statements to check before going forward with an execution.

As you saw in the Tokenizer, I've defined the enum of States and Triggers.
Here's an example:

```
enum class States{ s0, s1, …, s99};
enum class Triggers{ whitespace, letter, close_brace, return, ….};
```

The next step is to manually declare all the valid transitions, for example:

```
addTransition(States::s0, States::s1, Triggers::whitespace, [&]{return true; },
[&]{newToken(States::s1, true); });
```

Finally, to piece everything together, I used a function to map characters to the corresponding triggers, for example:

```
switch (ch) {
	case '\t':	return Triggers::whitespace;
	case '\n':	return Triggers::whitespace;
	case ' ':	return Triggers::whitespace;
```

In the case of letters:

```
        std::regex letter("([a-zA-Z])");
        std::smatch result;
        bool matched = false;

        matched = std::regex_match(temp, result, letter);
```

```
        if (matched) {
                return Triggers::letter;
        }
```

For reserved words, I just used a function to analyze the contents of id:

```
void Tokenizer::handleReservedWords() {
        std::string id = curr_token->value;
        if (id == "and") { fsm.execute(Triggers::and); return; }
```

All in all, I'm pretty confident with my solution. It greatly reduces complexity of managing all the states and it also allows to easily hack away at edge cases. Debugging is also very straightforward because I can easily follow along the transitions and triggers detected. The main labor was initially typing out all the transitions (triggers), states and mapping characters to triggers and states to their string equivalent.

Error Messages

In terms of error handling, I've used the idea of terminating at non-final states to generate errors. The lexical errors are pretty limited to invalid symbols and unresolved tokens.

From the state diagram you can guess what when wrong during those intermediate states, and those states were mapped to error messages, for example:

```
switch (fsm.state()) {
        case States::s2:
                error_msg += "Unexpected symbol";
```

I also log some relevant information:

```
        error_msg += " at line: " + std::to_string(line_number);
        error_msg += " loc: " + std::to_string(line_pos) + ".";
        error_msg += " Last safe state: ";
        error_msg += stringifyState(latestCorrectState);
```

Test Cases

Because of the large scope of valid and invalid inputs, I've devised some cases to verify the correctness of my tokenizer. On top of these test cases, a lot of testing was done manually with random input I came up with while debugging. Due to time constraints, I was unable to code a testing suite.

Lexically valid inputs

Input	Output
float; flOAt int123 program +	type_decl(float) semicolon type_decl(float) id reserved(program) arop(ADD)
soup = 50.0;	id assign floating_point semicolon
Int a + 1 = 213;	type_decl(int) id arop(ADD) integer assign integer semicolon
Abc123_ return;	id reserved(return) semicolon
Integer = 123.0;	id assign floating_point semicolon
0.0000005	floating_point
/* meow */	open_comment id close_comment
int abc_d = 0;	type_decl(int) id assign integer semicolon
- 000213	arop(SUB) integer integer integer integer
_dsadas_21321	underscore id

if ((a+b) == 6) {
   return 0;
}
	arop(SUB) integer integer integer integer

void integer(int a = 1; int b);	underscore id

for (int i = 0; i < 100.0; i++) {
   /* for loop // */
  i ====== 0;
};
	reserved(if) open_paren open_paren id arop(ADD) id close_paren relop(EQ) integer close_paren open_brace

a and b or c and d + k	reserved(return) integer semicolon
(1 <> 1)	close_brace
A <= b >= 3;	id id open_paren type_decl(int) id assign integer semicolon type_decl(int) id close_paren semicolon
char ch = 'c';	reserved(for) open_paren type_decl(int) id assign integer semicolon id relop(LT) floating_point semicolon id arop(ADD) arop(ADD) close_paren open_brace
1 / 0;	open_comment reserved(for) id line_comment close_comment

Lexically invalid inputs

Input	**Output**	**Errors**
$$$	unexpected	Invalid token: unexpected at line: 1 loc: 0. Last safe state: unexpected. Tokenizer stuck. Unexpected symbol.
ò☼╛ ‖	unexpected	Invalid token: unexpected at line: 2 loc: 0. Last safe state: unexpected. Tokenizer stuck. Unexpected symbol.
0.000.0	floating_point period integer	Invalid token: state23 at line: 3 loc: 7. Last safe state: period. Non-final state. '0' floating.
?	unexpected	
\\/\/\/\/\/\/\/\/	unexpected	Invalid token: unexpected at line: 5 loc: 0. Last safe state: unexpected. Tokenizer stuck. Unexpected symbol.
int != 4^2	type_decl(int) unexpected assign integer unexpected integer	
/	arop(DIV)	Invalid token: state41 at line: 7 loc: 1. Last safe state: state0. Non-final state. Cannot resolve forward slash.
123.	state19	Invalid token: state19 at line: 8 loc: 4. Last safe state: integer. Non-final state. Integer followed by dot but no other numbers.
0.100000	floating_point	Invalid token: state21 at line: 9 loc: 8. Last safe state: floating_point. Non-final state. Float ending with zeros.
and*	bop(and) arop(MUL)	Invalid token: state38 at line: 10 loc: 4. Last safe state: bop. Non-final state. WARNING.

Mix of lexically valid and invalid inputs

Input	**Output**	**Errors**
Fdgmkgfmsd	id	
232423#@#@1	integer unexpected	Invalid token: unexpected at line: 2 loc: 6. Last safe state: unexpected. Tokenizer stuck. Unexpected symbol.
$@sasd in rfew int rpeogram;	unexpected	Invalid token: unexpected at line: 3 loc: 0. Last safe state: unexpected. Tokenizer stuck. Unexpected symbol.
_	underscore ws	
DW!!	id unexpected	Invalid token: unexpected at line: 5 loc: 2. Last safe state: unexpected. Tokenizer stuck. Unexpected symbol.
46654. 0	state19 integer	
()(///*	open_paren close_paren open_paren line_comment open_comment	
'_'*	apostrophe underscore apostrophe arop(MUL)	
A * b = ‘;	id arop(MUL) id assign unexpected semicolon	
1 \ 0	integer unexpected integer	
1 /	integer arop(DIV)	Invalid token: state41 at line: 11 loc: 3. Last safe state: integer. Non-final state. Cannot resolve forward slash.
intint and	id bop(and)	
10001 or 0101	integer bop(or) integer integer	

Note: ' is valid, ‘ is invalid (these are different apostrophes)

Conclusion

My implementation of the tokenizer is fairly successful from my perspective, I look forward for feedback to improve it. One major thing I want to continue improving is the error messages. I would also like to implement a larger sample of test cases.

The design is flexible and it’s very easy to expand, adjust and debug. Fixing the DFA was the most important part regarding my design, since the tokenizer relies heavily on the logic encoded in the states and the transitions. Despite the dependency on the FSM, my solution allows me to tweak edge cases very easily.

To verify the above test cases, please see the attached README.txt file, copy the corresponding test case into the source.txt file and run the compiler.