# Lexical analyzer

xpojez00, xbudin05, xhribi00, xchupa03

October 5, 2021

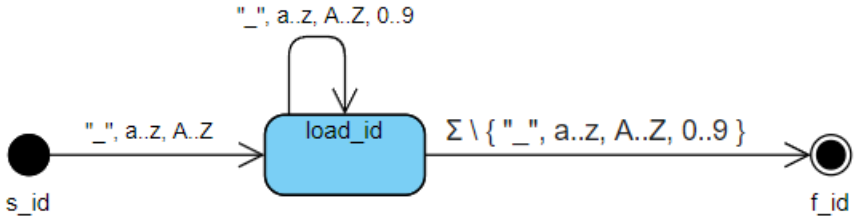
## 1 Introduction

Our approach was creating state machine (also referred as SM) for each category separately according to the project brief (in Czech). As seen in images beneath, we were not considering hexadecimal literals as valid values for integer. Our State Machines are affected by our approach of easier and cleaner implementations instead of more mathematical one. Nonetheless they should be considered equally correct. We have decided on distinguishing keywords in lexical analyzer and tokenizing them to off-load syntactic analyzer. Lexical analyzer should also find corresponding structure of the identifier in the symbol table[1] and send it as an attribute of the token[2].

## 2 SMs for each case separately

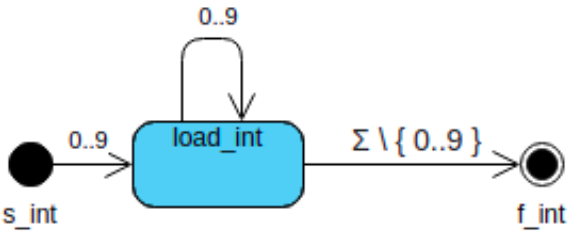
### 2.1 Identifiers or keywords

As stated above, we have decided for more programmatic approach and we consider identifiers and keywords the same type of lexeme. Distinguishing keywords is discussed later.




### 2.2 Integer

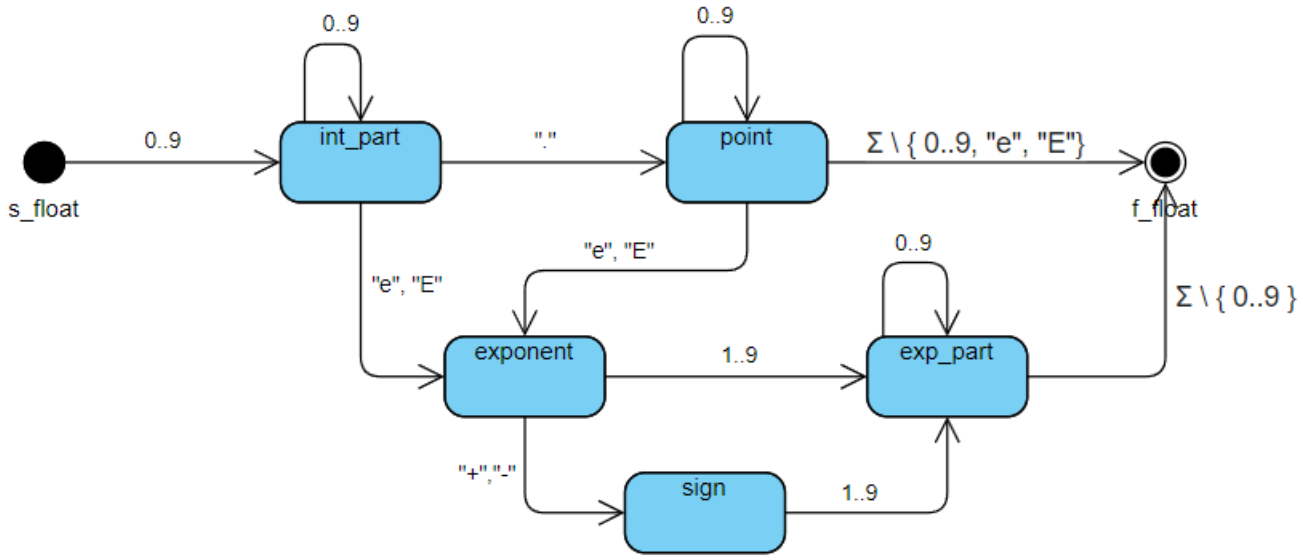
In our current state we do not consider hexadecimal literals to be possibly assigned to integers. It is not clear to us if it is or is not possible.



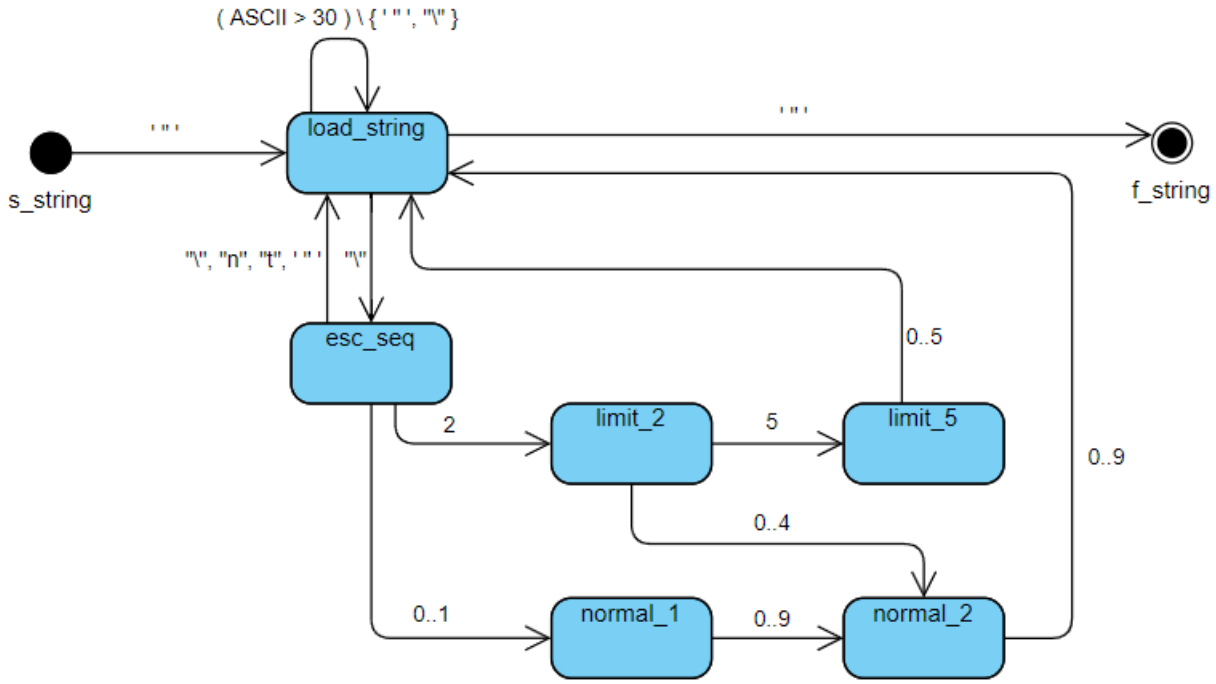

[1] Symbol table is implemented with hash table where every element is double linked list. Double linked list was selected for being able to add and remove elements in constant time while needing only the pointer to the current element for this operation

[2] Token is a struct containing enum value **type** to determinate what type it is and then **attribute**, which might be set depending on the value of **type**

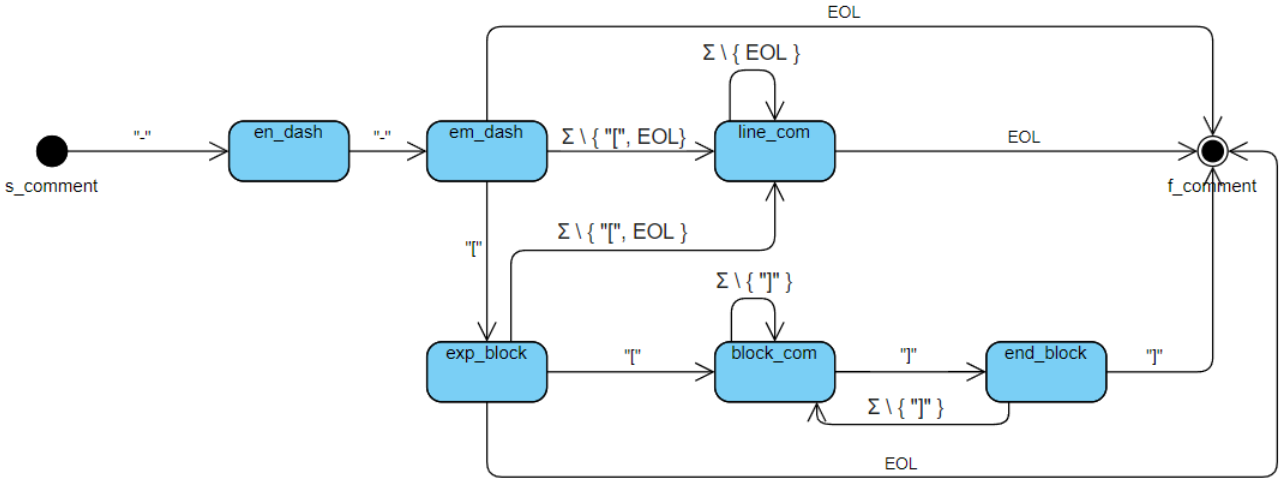
## 2.3 Floating point number




## 2.4 String

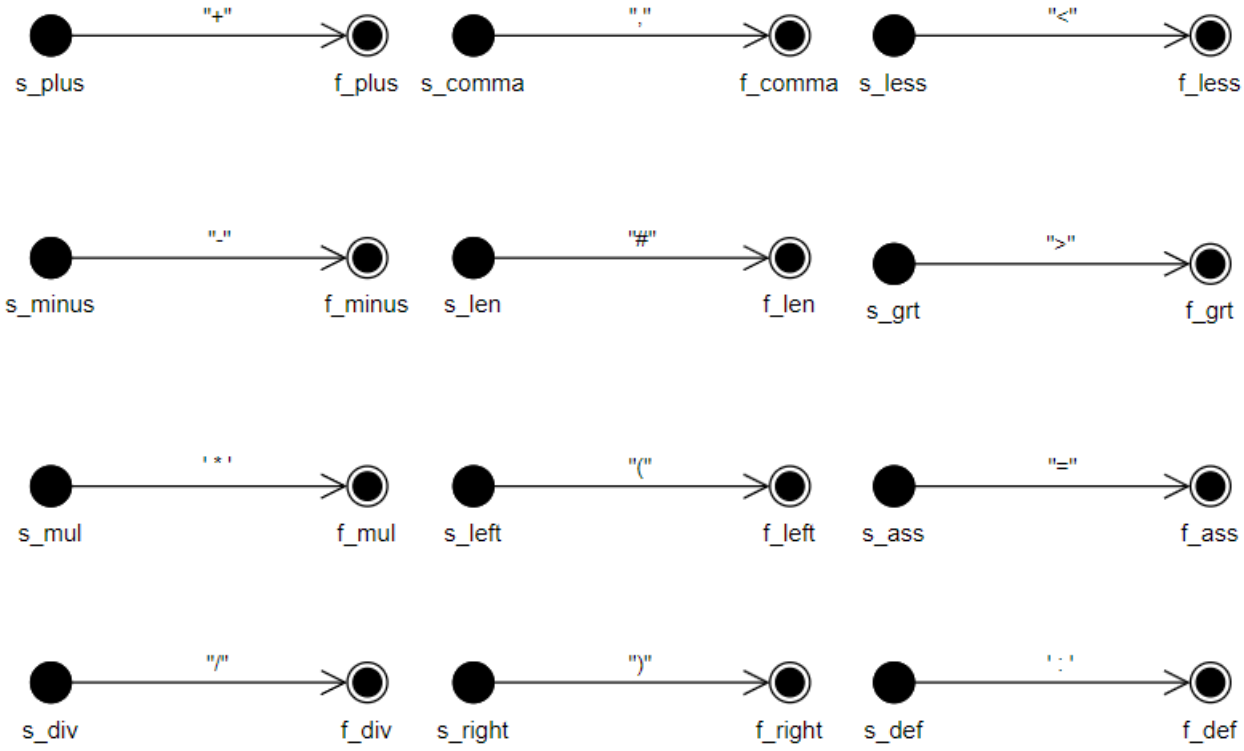


## 2.5 Line and block comment



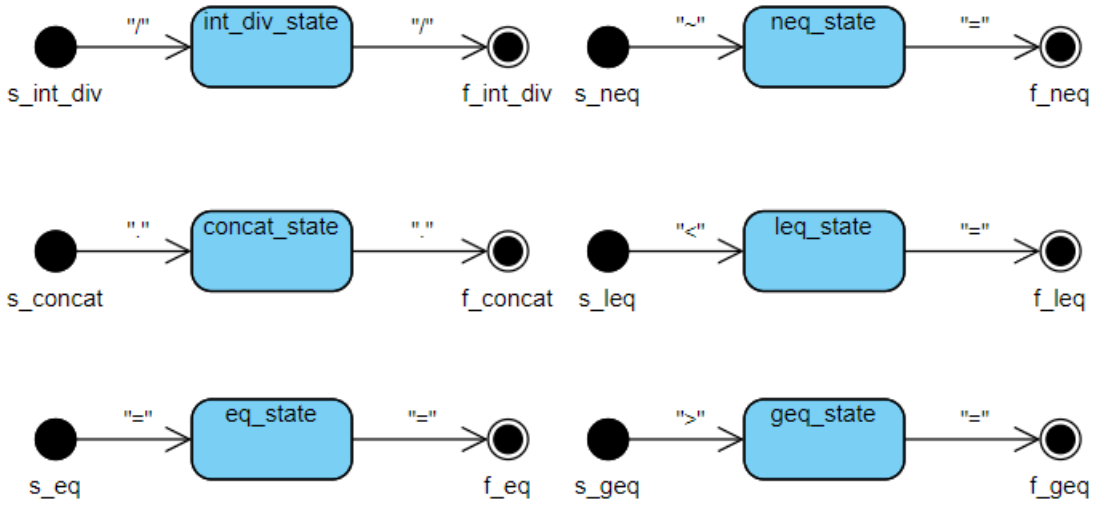

## 2.6 SMs with one transition

All SMs for one character lexemes are shown beneath



### 2.7 SMs with one state

All SMs with one state not considering starting and finishing state.




## 3 Creating single finite state machine

We have created one starting state and made transition to every starting state of separate SMs via $\epsilon$. We have not made a single finishing state as we did not consider it important for our case.

### 3.1 Deleting $\epsilon$ transitions

We deleted every starting state to which we were connected via $\epsilon$ and preserved the character leading to another non-$\epsilon$ transition. We were left with SM without $\epsilon$ transitions.

### 3.2 Removing non-determinism

If we found a unique transition we preserved it with it's state unchanged. If there were any overlapping transitions we merged overlapping characters into on transition leading to new state, which was named after conjunction of states. For every disjunctive character we would create new transition to new state called after the original state with description it was derived from the original. All overlapping transitions were deleted with their corresponding states and their transitions were attached to the new state. To derived states were attached all leaving transitions. We continued this process until we checked every state in the SM.

### 3.3 Removing non-starting states

We would remove all non-starting or non-finishing states as non-starting do not affect the Finite state machine (also referred as FSM).

### 3.4 Removing non-finishing states and creating finite state machine

We have deleted all non-finishing states as their existence only makes the not finite and we created new finishing state to which we lead transition from each state, which could had gotten stuck. This was achieved by transition over $\Sigma$ without characters leading to other states from the current state, to maintain determinism.

The final version is saved in Lexical_analyzer_FSM.pdf, you should open in it two-page mode.

### 3.5 Implementing details

We have implemented it with case conditions for state with variable **state** holding the enum value of the current state. Inside case we have if conditions for deciding the next state. We transition through the FSM as seen in the picture above. We store all read characters into another variable **lexeme**. After the FSM is iterated we know the final state and the lexeme.

## 4 Distinguishing keywords

Because of small amount of reserved keywords we have decided to create an array with all keywords. If the current identifier from FSM is in this array we know whether it is a keyword or not. This can be implemented with simple iteration through this array and comparing the lexeme with the values in the array.

## 5 Identifier

We find the element corresponding to the hashed lexeme and iterate through the double linked list to find the last occurrence of identifier. Then we attach this as an attribute to the token send it to the syntactic analyzer.