JLex

A Simple Lex Implementation

09021227 Jin Qiao

Contents

1 Motivation & Aim

lex is one of the most used lexical analyzer today and it is usually used with *yacc* to generate programs for those who want to write compilers or intepreters. For lex, you only need to write regular expressions and lex will generate corresponding lexical analyzer program. One of the most used implementation of lex is flex and one of the most used implementation of yacc is bison. Both of them are open-sourced and based on BSD license.

According to the description of the experiment, we are going to implement a simple lexical analyzer, which is like flex. It reads regular expressions defined by user and generate corresponding lexical analyzer program. We also need to define our `.l` file, where stores the regular expressions.

Therefore, introducing **JLex**, which is short for **J**inbridge-**Lex**ical-Analyzer. (Actually, JinBridge is my nickname). It mainly has following features:

- Nearly the same interface with *lex*. The `.l` file used by **JLex** is very similar to that used by *lex*, with only little change.
- Support escape characters, *e.g.* `\t`, `\n`.
- Created with C++11.

Anyway, I shall not spend too much time here, let's just get into the main topic.

2 Ideas & Methods

For the main idea and method, I'll just use algorithm discussed on class to keep this project as simple as possible. The program includes two part: the *JLex* itself and the `source.cpp` file.

Actually, I didn't read the source code of *lex* and I used a very dummy implementation, which is to replace text content of a prepared C++ program. The prepared C++ program is `source.cpp` file and *JLex* will replace the content of `source.cpp` to generate corresponding lexical analyzer. Here I draw a simple diagram in Fig. 1 to explain how it works.

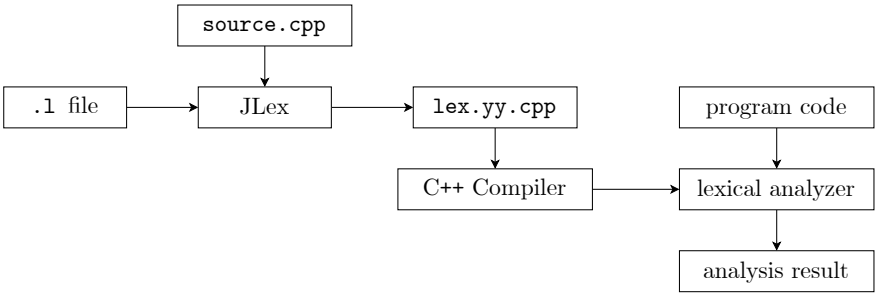


Figure 1: The flow chart of JLex. Almost the same with lex. You just give the `.l` file to *JLex* and it will generate corresponding `lex.yy.cpp` file. Compile this source file and you will get the lexical analyzer program. It reads the source code and outputs the analysis result.

The `source.cpp` file is prepared and all *JLex* do is to replace the content in it. So I will not spend too much time on replacing algorithm in *JLex* and instead, I'll introduce the content in `source.cpp`.

3 Content Description

A simple *lex* implementation written in C++. Source code available on here.

4 Assumptions

4.1 Regular Expression Format

To keep the project simple, I'll not implement all operators in *lex* regular expression format. Instead, the regular expression in *JLex* including following operators: `(`, `)`, `*`, `|`, `@`, which is listed in Table 1.

The first three operators are the same with corresponding symbol in textbook. Note that the fourth symbol, which is `@`. It connects two factors and usually we just ignore it. For example, we always write $a \times b$

as *ab* and we just skip times symbol. When we write regular expression, we also ignore the connect operator and write `a@b` as `ab`.

Operator	Name	Priority	Example
`()`	Parentheses	0	
`*`	Kleene star	1	`ab* = {a, ab, abb, ...}`
`|`	Or	2	`a|b = {a, b}`
`@`	Connect	3	`a@b = {ab}`

Table 1: Operators used in *JLex*.

Based on this 4 operators, we could express almost all the regular expressions. However, it's still not convenient. For example, if we want to express from A to Z, we must write `A|B|C|...|Z`. To make it easier to write regular expression, I defined following marcos: `A-Z`, `a-z`, `0-9`. The usage is the same with in *lex*. You could just write `(a-z)` to match all alphabets.

To match special characters such as `\n` and `\t`, it also support escape characters. Also, when you use `\(` in the regular expression, the `(` will be treated as a character. This is also suitable for `*`, `|` and `@`.

4.2 `.l` File Format

The `.l` file for *JLex* is relatively similar to that for *lex*. The `yylex()`, `yytext` and `yyin` is still available in *JLex*. The difference is:

- *JLex* uses `→` to separate regular expression and the action.
- *JLex* does not have definition section.

However, *JLex* still supports inserting code at the beginning of generated C++ code.

Here I give a simple example of `.l` file for *JLex*. It simply matches the integers in the text.

```
1    %{
2        // any code here will be inserted to the beginning of lex.yy.cpp
3        #include <iostream>
4    %}
5
6    %%
7
8    (0-9)* →  { std::cout << "integer:␣" + yytext << std::endl; }
9
10   %%
11
12   // any code here will be inserted to the end of lex.yy.cpp
13   int main(int argc, char** argv) {
14       yyin = argv[1];
15       yylex();
16       return 0;
17   }
```

An example of matching C language tokens will be given in *Use Cases on Running*.

5 Related FA Descriptions

Here I define two kinds of FA: static FA and runnable FA. Static FA is used to calculate and runnable FA is used to match tokens. We first calculate a static FA and then load it in a runnable FA to run.

5.1 Static FA

Here I'll show how I define static finite-state automaton (FA) in code.

Actually, the FA stores a graph, where nodes are states and edges are transition. I use adjacency list to store the graph because the FA is a sparse graph. I'll introduce adjacency list in *Description of Important*

Data Structures. For each edge in graph, I store three attributes: `_to`, `_next`, `_ch`. The first two attributes are used for adjacency list and the third attribute `_ch` is used to store the character for transition. Note that I use `@` to express ε.

Besides the graph, some other attributes also needed to be stored in FA:

- total states of FA.
- the set of all symbols of FA. This is used for simplify NFA to DFA and minimize DFA.
- the acceptable states of FA and the corresponding token.

Also, I add a simple debug function to print the content in FA. It will print every edge of the FA and outputs the acceptable states in FA.

Below is the `fa` struct. Here I misused the *status* and *state*. But actually, they are the same meanings in my code.

```
1    // actually, it is a graph
2    // use '@' for epsilon
3    struct fa {
4        fa(int total_status) : _total_status(total_status), _head(std::
             vector<int>    (total_status, -1)) {
5            for (int i = 0; i < total_status; ++i) {
6                _accept_status[i] = -1;
7            }
8        }
9
10       int _total_status = 0;
11       // start status is default 0
12       // for nfa, accept status is total status - 1
13       // for dfa, accept status is _accept_status
14
15       struct edge {
16           int  _to, _next;
17           char _ch;
18       };
19       std::vector<int>  _head;
20       std::vector<edge> _edge;
21
22       std::set<char> _symbols;
23       // std::set<int>      _accept_status;
24       // key: status, value: token enum in int
25       std::map<int, int> _accept_status;
26
27       void add_edge(int from, int to, char c) {
28           _edge.push_back(edge());
29           _edge[_edge.size() - 1]._to   = to;
30           _edge[_edge.size() - 1]._ch   = c;
31           _edge[_edge.size() - 1]._next = _head[from];
32           _head[from]                   = _edge.size() - 1;
33       }
34
35       void print() {
36           for (int i = 0; i < _total_status; ++i) {
37               if (_head[i] == -1)
38                   continue;
39               std::cout << "Status␣" + std::to_string(i) << std::endl;
40               for (int idx = _head[i]; idx != -1; idx = _edge[idx]._next
                     )
41                   std::cout << "␣␣␣␣--" << _edge[idx]._ch << "-->␣" <<
                        std::to_string(_edge    [idx]._to) << std::endl;
42           }
43           std::cout << "Accept␣states:" << std::endl;
```

```
44              for (auto i : _accept_status)
45                  std::cout << "␣␣␣␣" << i.first << ":␣" << i.second << std
                        ::endl;
46          }
47      };
```

5.2 Runnable FA

After we calculate a static FA, we need to load it into a runnable FA. Compared to static FA, the runnable FA stores current state, and it supports input characters. Also, it supports you to reset it.

```
1     struct runnable_fa {
2         runnable_fa(fa f) : _fa(f), _current_state(0) {}
3         fa  _fa;
4         int _current_state;
5         // return value:
6         // false: failed, true: find
7         bool input(char ch) {
8             for (int i = _fa._head[_current_state]; i != -1; i = _fa._edge
                  [i]._next) {
9                 if (_fa._edge[i]._ch == ch) {
10                    _current_state = _fa._edge[i]._to;
11                    return true;
12                }
13            }
14            return false;
15        }
16
17        void reset() {
18            _current_state = 0;
19        }
20    };
```

6 Description of Important Data Structures

6.1 Adjacency List

Adjacency list is used to store a graph, especially a sparse graph. It includes following parts:

- `head` array, which stores the index of first edge from the node. For example, `head[i]` is the index of first edge from node `i`.
- `edge` array, which stores the information of edges.
- the number of total edges.

For each edge, it has three attributes:

- `next`, stores the index of next edge which starts from the same node.
- `to`, stores the destination of the edge.

I'll not explain the algorithms used to add edge and iterate edges because they are not important, and you could easily get more detailed explanation about them on Google.

6.2 Containers of STL

To simplify the project, I used some containers in STL:

- `std::vector`, it's time to throw traditional array away.
- `std::stack`, a STL implementation of stack.
- `std::set`, stores a set where there are no two same elements.
- `std::string`, stores the string.
- `std::function`, stores the function when match a specific regular expression. I used it together with lambda expression of C++11.

Also, I'll not explain them here because a more detailed explanation could be found in CppReference.

7 Description of Core Algorithm

The core algorithms are the same with those on textbook. I'll introduce them according to the sequence of I used them.

7.1 Convert Regular Expression to NFA

For this part, the main task is to convert the given regular expression to NFA. Note that before we actually convert it, we need to expand the macros like `a-z`, and read the escaped characters like `\(`. Also, we need to complete all connect operators we ignored before.

After we applied these three steps on given regular expression, we then convert the regular expression from infix expression to postfix expression so that we could process it with a stack.

Here's the algorithm to convert infix expression to postfix expression, which is Algorithm 1.

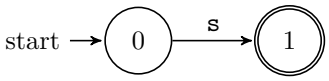
Algorithm 1: Convert infix regular expression to postfix regular expression

Input: Infix regular expression R
Output: Postfix regular expression R'

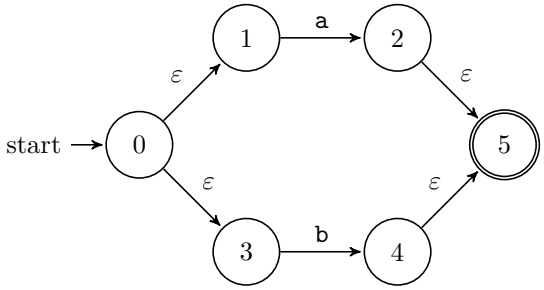
```
Let O be the operator stack.
Let I be the intermediate stack.
for i ← 0 to R's length do
    if R[i] is not operator then
        Push R[i] into I.
        continue
    end
    if R[i] is operator then
        if O is empty or the top of O is ( then
            Push R[i] into O.
            continue
        end
        while priority of the top of O is higher than R[i] do
            Push the top of O into I.
            Pop the top of O.
        end
        if priority of the top of O is lower than R[i] then
            Push R[i] into O.
            continue
        end
    end
    if R[i] is ( then
        Push R[i] into O.
        continue
    end
    if R[i] is ) then
        while the top of O is not ( do
            Push the top of O into I.
            Pop the top of O.
        end
        Pop the top of O.
        continue
    end
end
Push elements in O into I one by one.
Let R' be the reverse of elements in I.
return R'
```

After we convert the infix regular expression to postfix regular expression, we could calculate the NFA of it just like the way we calculate value of postfix expression. We will use a stack to calculate the NFA of it.

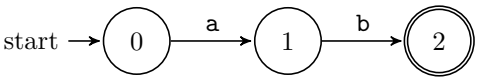
Note that for each of character, we will generate a NFA for it. For example, for character `s` we will generate following NFA for it:



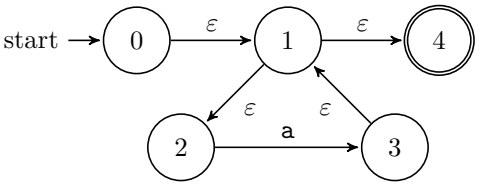
When we combine two NFAs with operator `|`, for example a NFA for `a|b`, we will generate following NFA:



When we combine two NFAs with operator `@`, for example, a NFA for `a@b`, we will generate following NFA:



When we use operator `*`, for example, a NFA for `a*`, we will generate following NFA:



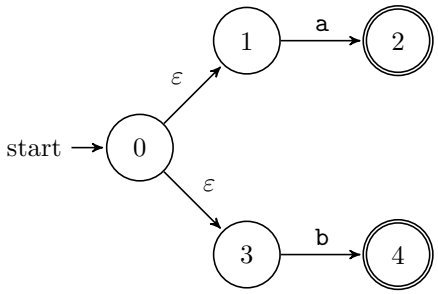
After we define actions for these operators, we are able to introduce the algorithm to generate the NFA, which is Algorithm 2.

Algorithm 2: Convert postfix regular expression to NFA

Input: Postfix regular expression R
Output: Corresponding NFA N

```
Let S be a stack.
for i ← 0 to R's length do
    if R[i] is an operator then
        Get the top two elements in S and remove them in S.
        Perform corresponding operation on these two elements.
        Push the result into S.
    else
        Generate corresponding NFA and push it into S.
    end
end
return R'
```

Therefore, we could convert each regular expression to corresponding NFA. The next problem is how to combine these NFAs to become a big NFA. The idea is very simple, just add a common start and let it point to the start of those NFAs. For example, if we have NFA for `a` and `b`, we could combine them in this way:



The algorithm for this action is shown in Algorithm 3.

Algorithm 3: Combine NFAs to become a big NFA

Input: NFAs $N_1, N_2, ..., N_m$
Output: Combined NFA N
Let N be an empty FA.
Add dummy state 0 to N.
foreach N_i *in NFAs* **do**
    Copy N_i to N.
    Add an edge from 0 to the start of N_i.
end
return N

7.2 Convert NFA to DFA

The next step is to convert the combined NFA to DFA. Here I just use the algorithm in textbook, which is shown in Algorithm 4.

Algorithm 4: Convert NFA to DFA

Input: NFA N
Output: Corresponding DFA D
Initially, ε-*closure*(s_0) is the only state in D, and it is unmarked.
while *there is an unmarked state* T *in* D **do**
    Mark T.
    foreach *input symbol* a **do**
        Let U be ε-*closure*$(move(T, a))$.
        if U *is not in* D **then**
            Add U as an unmarked state to D.
        end
        Add an edge from T to U with char a.
    end
end
return D

7.3 Minimize DFA

In order to make the program run faster, we need to minimize the states in DFA. Here I also used the algorithm in textbook, which is shown in Algorithm 5.

7.4 Run DFA on Given Text

The final part is to run DFA on given text. Again, I used the algorithm in textbook, which is shown in Algorithm 6.

`regexpr.hpp`, `nfa.hpp`, `dfa.hpp`, `minimize_dfa.hpp`, `runner.hpp`, `lex.hpp` including all algorithm I mentioned above. All these files combined to be `source.cpp`. What *JLex* does is to replace the content in `source.cpp`.

Algorithm 5: Minimize DFA

Input: DFA D
Output: Corresponding minimized DFA D'

Construct an initial partition Π with two groups F and $S - F$.
Let Π_{new} be empty.
while Π_{new} *not equals to* Π **do**
    Let Π_{new} equals to Π.
    foreach *group* G *in* Π **do**
        Partition G into subgroups such that two states s and t are in the same subgroup if and only if for all input symbol a, state s and t have transitions on a to states in the same group of Π.
        Replace G in Π_{new} by the set of all subgroups formed.
    end
end
Choose one state in each group of Π_{final} as the *representative* for that group and Construct the rest of DFA by these representatives.
return D'

Algorithm 6: Run DFA on Given Text

Input: DFA D, Given Text S
Output: A boolean value to show if the progress has error

foreach *character* c *in* S **do**
    if c *is not in the alphabets of* S **then**
        return False
    end
    Let s be the current state of D.
    if s *has an edge to s' with character* c **then**
        Let the current state of D be s'.
    else
        if s *is acceptable* **then**
            Perform corresponding action.
            Reset the current state of D to start.
            Retry character c.
        else
            return False
        end
    end
end
return True

8 Use Cases on Running

Here's a simple example to recognize the tokens in C language.

8.1 Input

8.1.1 `example.l`

Here's the content in `example.l` file, and it will simply output what it recognized.

```
%{
    #include <iostream>
    int test = 0;
%}

%%

 ( )* →        {}
\n →           {}
; →            { std::cout << "MATCH: ;" << std::endl; }
\\( →          { std::cout << "MATCH: (" << std::endl; }
\\) →          { std::cout << "MATCH: )" << std::endl; }
{ →            { std::cout << "MATCH: {" << std::endl; }
} →            { std::cout << "MATCH: }" << std::endl; }
[ →            { std::cout << "MATCH: [" << std::endl; }
] →            { std::cout << "MATCH: ]" << std::endl; }

main →         { std::cout << "MATCH: main" << std::endl; }

auto →         { std::cout << "MATCH: auto" << std::endl; }
case →         { std::cout << "MATCH: case" << std::endl; }
const →        { std::cout << "MATCH: const" << std::endl; }
default →      { std::cout << "MATCH: default" << std::endl; }
double →       { std::cout << "MATCH: double" << std::endl; }
enum →         { std::cout << "MATCH: enum" << std::endl; }
float →        { std::cout << "MATCH: float" << std::endl; }
goto →         { std::cout << "MATCH: goto" << std::endl; }
int →          { std::cout << "MATCH: int" << std::endl; }
register →     { std::cout << "MATCH: register" << std::endl; }
short →        { std::cout << "MATCH: short" << std::endl; }
sizeof →       { std::cout << "MATCH: sizeof" << std::endl; }
struct →       { std::cout << "MATCH: struct" << std::endl; }
typedef →      { std::cout << "MATCH: typedef" << std::endl; }
unsigned →     { std::cout << "MATCH: unsigned" << std::endl; }
volatile →     { std::cout << "MATCH: volatile" << std::endl; }
break →        { std::cout << "MATCH: break" << std::endl; }
char →         { std::cout << "MATCH: char" << std::endl; }
continue →     { std::cout << "MATCH: continue" << std::endl; }
do →           { std::cout << "MATCH: do" << std::endl; }
else →         { std::cout << "MATCH: else" << std::endl; }
extern →       { std::cout << "MATCH: extern" << std::endl; }
for →          { std::cout << "MATCH: for" << std::endl; }
if →           { std::cout << "MATCH: if" << std::endl; }
long →         { std::cout << "MATCH: long" << std::endl; }
return →       { std::cout << "MATCH: return" << std::endl; }
signed →       { std::cout << "MATCH: signed" << std::endl; }
while →        { std::cout << "MATCH: while" << std::endl; }
switch →       { std::cout << "MATCH: switch" << std::endl; }
union →        { std::cout << "MATCH: union" << std::endl; }
```

```
50    void →          { std::cout << "MATCH:␣void" << std::endl; }
51    static →        { std::cout << "MATCH:␣static" << std::endl; }
52
53    + →             { std::cout << "MATCH:␣+" << std::endl; }
54    - →             { std::cout << "MATCH:␣-" << std::endl; }
55    \\* →           { std::cout << "MATCH:␣*" << std::endl; }
56    / →             { std::cout << "MATCH:␣/" << std::endl; }
57    % →             { std::cout << "MATCH:␣%" << std::endl; }
58    == →            { std::cout << "MATCH:␣==" << std::endl; }
59    != →            { std::cout << "MATCH:␣!=" << std::endl; }
60    > →             { std::cout << "MATCH:␣>" << std::endl; }
61    < →             { std::cout << "MATCH:␣<" << std::endl; }
62    >= →            { std::cout << "MATCH:␣>=" << std::endl; }
63    <= →            { std::cout << "MATCH:␣<=" << std::endl; }
64    && →            { std::cout << "MATCH:␣&&" << std::endl; }
65    \\|\\| →        { std::cout << "MATCH:␣||" << std::endl; }
66    ! →             { std::cout << "MATCH:␣!" << std::endl; }
67    & →             { std::cout << "MATCH:␣&" << std::endl; }
68    \\| →           { std::cout << "MATCH:␣|" << std::endl; }
69    ^ →             { std::cout << "MATCH:␣^" << std::endl; }
70    ~ →             { std::cout << "MATCH:␣~" << std::endl; }
71    << →            { std::cout << "MATCH:␣<<" << std::endl; }
72    >> →            { std::cout << "MATCH:␣>>" << std::endl; }
73    = →             { std::cout << "MATCH:␣=" << std::endl; }
74    += →            { std::cout << "MATCH:␣+=" << std::endl; }
75    -= →            { std::cout << "MATCH:␣-=" << std::endl; }
76    \\*= →          { std::cout << "MATCH:␣*=" << std::endl; }
77    /= →            { std::cout << "MATCH:␣/=" << std::endl; }
78    %= →            { std::cout << "MATCH:␣%=" << std::endl; }
79    <<= →           { std::cout << "MATCH:␣<<=" << std::endl; }
80    >>= →           { std::cout << "MATCH:␣>>=" << std::endl; }
81    &= →            { std::cout << "MATCH:␣&=" << std::endl; }
82    ^= →            { std::cout << "MATCH:␣^=" << std::endl; }
83    \\|= →          { std::cout << "MATCH:␣|=" << std::endl; }
84    ++ →            { std::cout << "MATCH:␣++" << std::endl; }
85    -- →            { std::cout << "MATCH:␣--" << std::endl; }
86
87    (0-9)* →                          { std::cout << "MATCH:␣integer:␣" +
          yytext << std::endl; }
88    (0-9)*.(0-9)* →                   { std::cout << "MATCH:␣float:␣" +
          yytext << std::endl; }
89    (A-Z|a-z|_)(A-Z|a-z|0-9|_)* →     { std::cout << "MATCH:␣identifier:␣" +
           yytext << std::endl; }
90
91    %%
92
93    int main(int argc, char** argv) {
94        yyin = argv[1];
95        yylex();
96        return 0;
97    }
```

8.1.2 `example.c`

Here's the C language file to be recognized, which is `example.c`:

```
1    int main() {
2        float p    = 1.1;
```

```
3         int   i    = 2;
4         int   loop = 0;
5         int   a[10][10];
6         int   x;
7         while (loop == 0 && i <= 10) {
8             int j = 1;
9             while (loop == 0 && j < i)
10                if (a[i][j] == x)
11                    loop = 1;
12                else
13                    j = j + 1;
14            if (loop == 0)
15                i = i + 1;
16        }
17    }
```

8.2 Usage

Here is the method to run it:

```
1    ./build/jlex ./example/example.l
2    cc -o test ./lex.yy.cpp
3    ./test ./example/example.c
```

8.3 Output

Here's the output of the program:

```
1    MATCH: int
2    MATCH: main
3    MATCH: (
4    MATCH: )
5    MATCH: {
6    MATCH: float
7    MATCH: identifier: p
8    MATCH: =
9    MATCH: float: 1.1
10   MATCH: ;
11   MATCH: int
12   MATCH: identifier: i
13   MATCH: =
14   MATCH: integer: 2
15   MATCH: ;
16   MATCH: int
17   MATCH: identifier: loop
18   MATCH: =
19   MATCH: integer: 0
20   MATCH: ;
21   MATCH: int
22   MATCH: identifier: a
23   MATCH: [
24   MATCH: integer: 10
25   MATCH: ]
26   MATCH: [
27   MATCH: integer: 10
28   MATCH: ]
29   MATCH: ;
```

```
30    MATCH: int
31    MATCH: identifier: x
32    MATCH: ;
33    MATCH: while
34    MATCH: (
35    MATCH: identifier: loop
36    MATCH: ==
37    MATCH: integer: 0
38    MATCH: &&
39    MATCH: identifier: i
40    MATCH: <=
41    MATCH: integer: 10
42    MATCH: )
43    MATCH: {
44    MATCH: int
45    MATCH: identifier: j
46    MATCH: =
47    MATCH: integer: 1
48    MATCH: ;
49    MATCH: while
50    MATCH: (
51    MATCH: identifier: loop
52    MATCH: ==
53    MATCH: integer: 0
54    MATCH: &&
55    MATCH: identifier: j
56    MATCH: <
57    MATCH: identifier: i
58    MATCH: (
59    MATCH: identifier: loop
60    MATCH: ==
61    MATCH: integer: 0
62    MATCH: )
63    MATCH: identifier: i
64    MATCH: =
65    MATCH: identifier: i
66    MATCH: +
67    MATCH: integer: 1
68    MATCH: ;
69    MATCH: }
70    MATCH: }
```

9 Problems Occurred and Related Solutions

Actually, no problems occurred.

10 Feelings and Comments

Pretty interesting. Made me realize more about lexical analysis.