Compiler Design Project
Phase#1

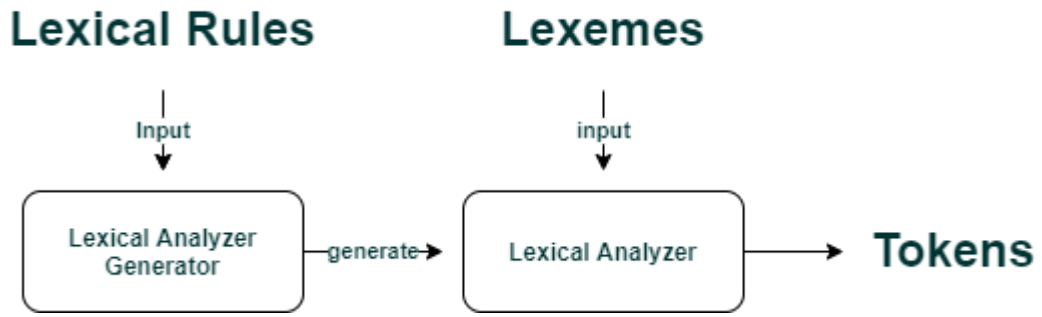
Lexical Analyzer Generator

Abobakr Abdelaziz ID : 2
Ahmed Ali Elsayed ID : 8
Elsayed Akram Elsayed ID : 16
Fares Medhat El-Saadawy ID: 47

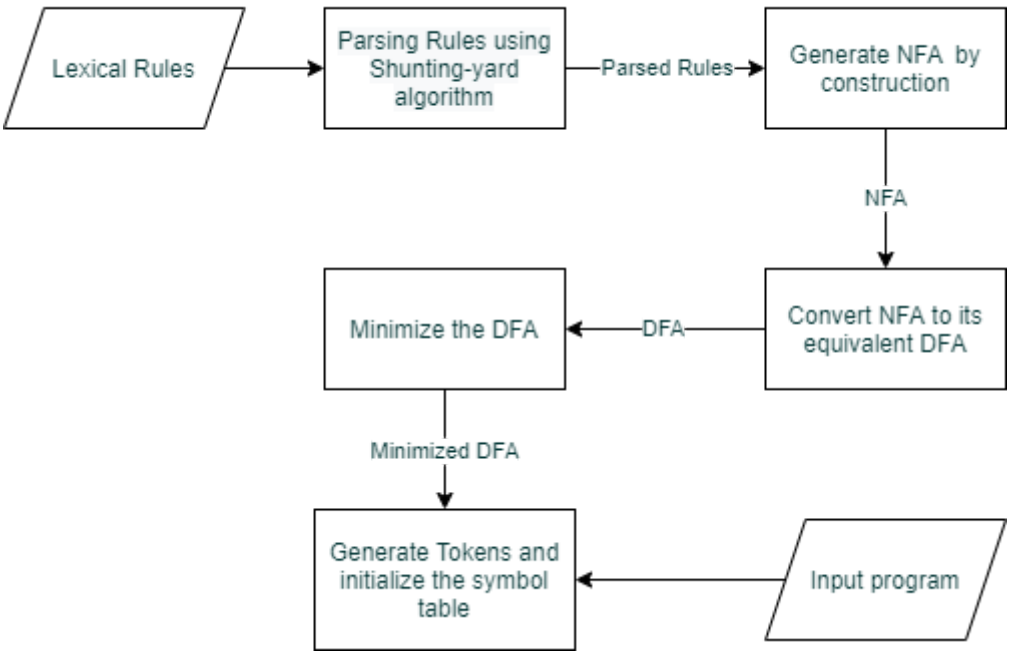
PROF. Nagia Ghanem

Overview

What is lexical analysis? lexical analysis or tokenization is the process of converting a sequence of characters (lexemes)- such as in a computer program - into a sequence of tokens - strings with an assigned and thus identified meaning -
What is the lexical analyzer generator? A lexical analyzer generator is a program designed to generate lexical analyzers, which recognize lexical patterns in text. The lexical analyzer generator is required to automatically construct a lexical analyzer from a regular expression description (rules) of a set of tokens.

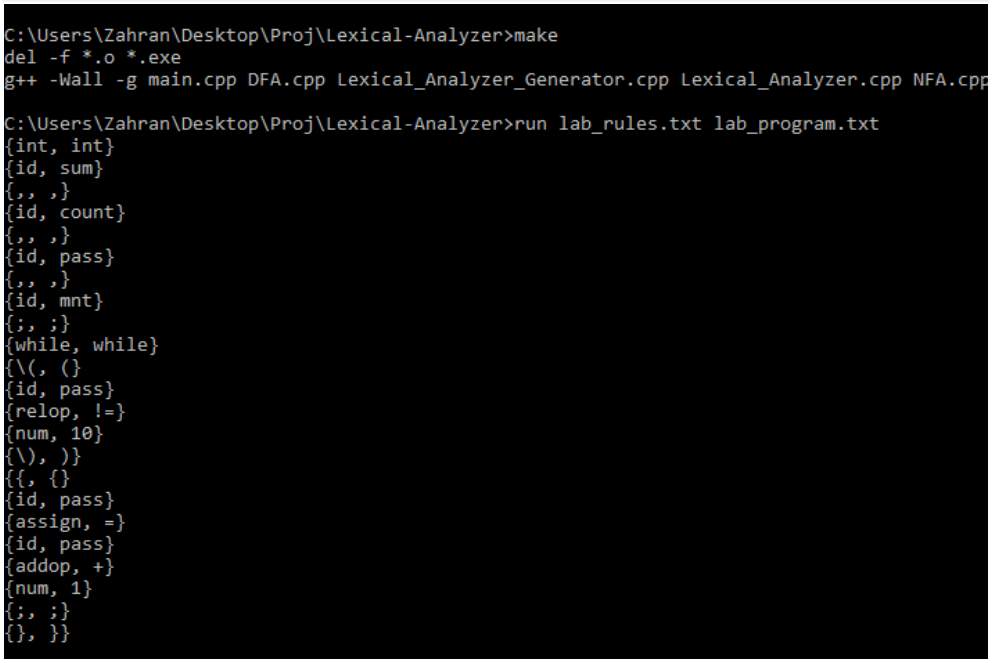


Program Flow



How to run

```
Run this command in terminal
    -   make
    -   run [rules-file-path] [program-file-path]
EX:
    -   make
    -   run lab_rules.txt lab_program.txt
```



Algorithms

Shunting-yard algorithm:

Shunting-yard algorithm converts regular definitions and regular expressions into the postfix representation to be easily calculated for constructing the NFA.
Ex : `l ( l | d ) * --> l l d | *`

NFA Builder:

The NFA builder implements a `stack<NFA*>` to keep track of the latest build NFA to perform Thompson construction algorithm on it.
For the previous example stack is `[ l ] -> [ l , l ] -> [ l , l , d ] -> [l , (l|d) ] -> [l , (l|d)* ] ->[ l(l|d)* ]`

Subset Construction

The algorithm constructs a transition table for the DFA. Each state of the DFA is a set of NFA states. The algorithm needs functions to work properly on the NFA states :

1. **epsClosure(s):** gets all the states that move with EPS transition from current state by putting the input states in a stack then getting all states that go with EPS transition and push them into the stack until all states with EPS transition are reached.
2. **epsClosure(T):** works the same as **epsClosure(s)** but on a set of states T.
3. **move(s, condition):** the function gets the set of states that s moves to on a specific condition from the transition table.

Minimization of DFA

1. Maps each set of states by its accepting token into a `unorderd_map<TokenKey, set<State*>>` and the non accepting ones are mapped into a temporary token "NonAccepting".
2. Then looping over all input symbols and use symbol **x** to try to split each group into smaller subgroups.
3. In `partition(groups, x)` function, we loop over groups, and each group we loop over a states ***s*** and mark this state (this mark means it's been put in a subgroup), then loop over other the rest of states and try the input x on each of them, if a state ***v*** goes to the same group then it's put in the same subgroup with state ***s*** and also marked.

Data Structures

- At class token: Set for keys
- At class utilities : Vector
- At class NFABuilder: Vector of pointers - unordered_map for "regularDefinitionNFAs"
- At class NFA: Set for "states" - map for "transitions From State" which maps state to set of transitions
- At class Lexical_Analyzer: Set for "Dstates' - map for "Transitions" which maps state to set of transitions - vectors.
- At class DFA : Set of strings - set of pointers of tokens - map from set to pointer of state - map from pointer of state to set.

Minimal DFA Transition Table

Excel Sheet.

Resultant Tokens of test program

EX 1 :

For the following rules and test program

```
letter = a-z | A-Z
digit = 0 - 9
```

```
{boolean int float}
{while if else}
id: letter (letter | digit)*
digits = digit+
num: digit+ | digit+ . digits (\L | E digits)
relop: \=\= | !\= | > | >\= | < | <\=
assign: \=
[; , \( \) { }]
addop: \+ | \-
mullop: \* | /
```

```
int sum , count , pass , mnt; while (pass !=
10 )
{
pass = pass + 1;
}
boolean float int intx
```

The output tokens are

```
{int, int} {id, sum} {id, count} {id, pass} {id, mnt} {while,
while} {id, pass} {relop, !=} {num, 10} {id, pass} {assign, =}
{id, pass} {addop, +} {num, 1} {boolean, boolean} {float,
float} {int, int} {id, intx}
```

EX 2 :

For the following rules and test program

```
letter = a-z | A-Z
digit = 0 - 9
{boolean int float while if else class public private}
{static void for main String System out println java io import}
id: letter (letter | digit)*
digits = digit+
num: digit+ | digit+ . digits (\L | E digits)
relop: \=\= | !\= | > | >\= | < | <\=
assign: \=
[; & ^ | , \( \) [ ] { } .]
addop: \+ | \-
mullop: \* | /
inc: \+\+
dec: \-\-
bitwise: >> | <<
```

```
import java.io.*;
public class GFG {
    static int N = 100000;
    static int n;
    static int []tree = new int[2 * N];
    static void build( int []arr) {
        for (int i = 0; i < n; i++)tree[n + i] = arr[i];
        for (int i = n - 1; i > 0; --i)
            tree[i] = tree[i << 1] + tree[i << 1 | 1];
    }
    static void updateTreeNode(int p, int value) {
        tree[p + n] = value;
        p = p + n;
        for (int i = p; i > 1; i >>= 1)
            tree[i >> 1] = tree[i] + tree[i^1];
    }
    static int query(int l, int r) {
        int res = 0;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if ((l & 1) > 0)res += tree[l++];
            if ((r & 1) > 0)res += tree[--r];
        }
        return res;
    }
    static public void main (String[] args) {
        int []a = {1, 2, 3, 4, 5, 6, 7, 8,9, 10, 11, 12};
        n = a.length;
        build(a);
        System.out.println(query(1, 3));
        updateTreeNode(2, 1);
        System.out.println(query(1, 3));
    }
}
```

The output tokens are

```
{import, import} {java, java} {io, io} {mullop, *} {public,
public} {class, class}   {id, GFG} {static, static} {int, int}
{id, N} {assign, =} {num, 100000} {static, static} {int, int}
{id, n} {static, static} {int, int} {id, tree} {assign, =} {id,
new} {int, int} {num, 2} {mullop, *} {id, N} {static, static}
{void, void} {id, build} {int, int} {id, arr} {for, for} {int,
int} {id, i} {assign, =} {num, 0} {id, i} {relop, <} {id, n}
{id, i} {inc, ++} {id, tree} {id, n} {addop, +} {id, i}
{assign, =} {id, arr} {id, i} {for, for} {int, int} {id, i}
```

```
{assign, =} {id, n} {addop, -} {num, 1} {id, i} {relop, >}
{num, 0} {dec, --} {id, i} {id, tree} {id, i} {assign, =} {id,
tree} {id, i} {bitwise, <<} {num, 1} {addop, +} {id, tree} {id,
i} {bitwise, <<} {num, 1} {num, 1} {static, static} {void,
void} {id, updateTreeNode} {int, int} {id, p} {int, int} {id,
value} {id, tree} {id, p} {addop, +} {id, n} {assign, =} {id,
value} {id, p} {assign, =} {id, p} {addop, +} {id, n} {for,
for} {int, int} {id, i} {assign, =} {id, p} {id, i} {relop, >}
{num, 1} {id, i} {bitwise, >>} {assign, =} {num, 1} {id, tree}
{id, i} {bitwise, >>} {num, 1} {assign, =} {id, tree} {id, i}
{addop, +} {id, tree} {id, i} {num, 1} {static, static} {int,
int} {id, query} {int, int} {id, l} {int, int} {id, r} {int,
int} {id, res} {assign, =} {num, 0} {for, for} {id, l} {addop,
+} {assign, =} {id, n} {id, r} {addop, +} {assign, =} {id, n}
{id, l} {relop, <} {id, r} {id, l} {bitwise, >>} {assign, =}
{num, 1} {id, r} {bitwise, >>} {assign, =} {num, 1} {if, if}
{id, l} {num, 1} {relop, >} {num, 0} {id, res} {addop, +}
{assign, =} {id, tree} {id, l} {inc, ++} {if, if} {id, r} {num,
1} {relop, >} {num, 0} {id, res} {addop, +} {assign, =} {id,
tree} {dec, --} {id, r} {id, return} {id, res} {static, static}
{public, public} {void, void} {main, main} {String, String}
{id, args} {int, int} {id, a} {assign, =} {num, 1} {num, 2}
{num, 3} {num, 4} {num, 5} {num, 6} {num, 7} {num, 8} {num, 9}
{num, 10} {num, 11} {num, 12} {id, n} {assign, =} {id, a} {id,
length} {id, build} {id, a} {System, System} {out, out}
{println, println} {id, query} {num, 1} {num, 3} {id,
updateTreeNode} {num, 2} {num, 1} {System, System} {out, out}
{println, println} {id, query} {num, 1} {num, 3}
```

Assumptions & their Justifications

Recovery routine: if an input goes to a dead state because of an unknown symbol like underscore in "_abc", an error is printed and all the word is neglected.