COMP 442 / 6421 Compiler Design

Tutorial 2
Lexical Analyser

Instructor: Dr. Joey Paquet paquet@cse.concordia.ca
TAs: Haotao Lai h_lai@encs.concordia.ca

Lab Instructor

Section: lab hours NNK M------ 20:30-22:20 H819

Name: Haotao Lai (Eric)

Office: EV 8.241

Email: h_lai@encs.concordia

Website: http://laihaotao.me/ta

Lexical Analyser

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens .

Input: source code file

Output: tokens

What is token?

Token

- A syntactic category
  - In English:
    - Noun, verb, adjective
  - In a programming language:
    - Identifiers, Integer, Floating point number, Keywords, …

Token

In our assignment:

TOKEN is a data structure with the following components:

- Type
- Lexeme
- Position

Lexical Analyser

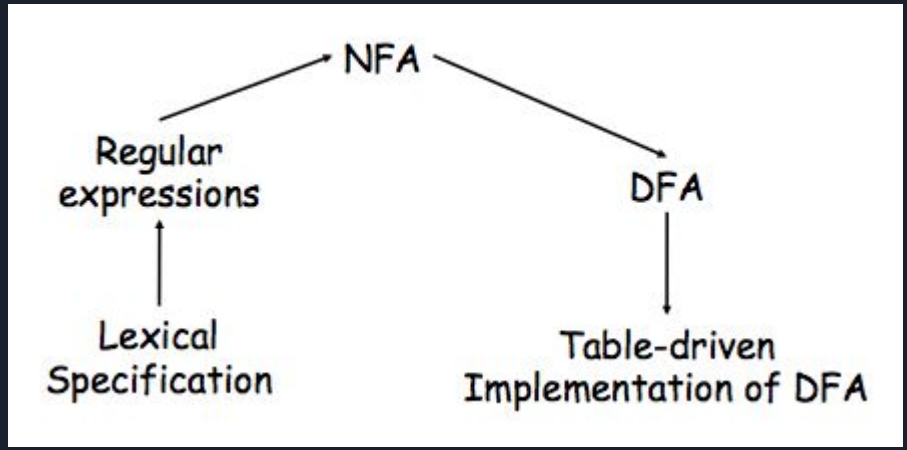
How to design a lexical analyser? (That's what the lexical specification does)

1. Define a finite set of tokens
2. Describe which strings belong to each token

How to implement a lexical analyser? (That's your assignment)

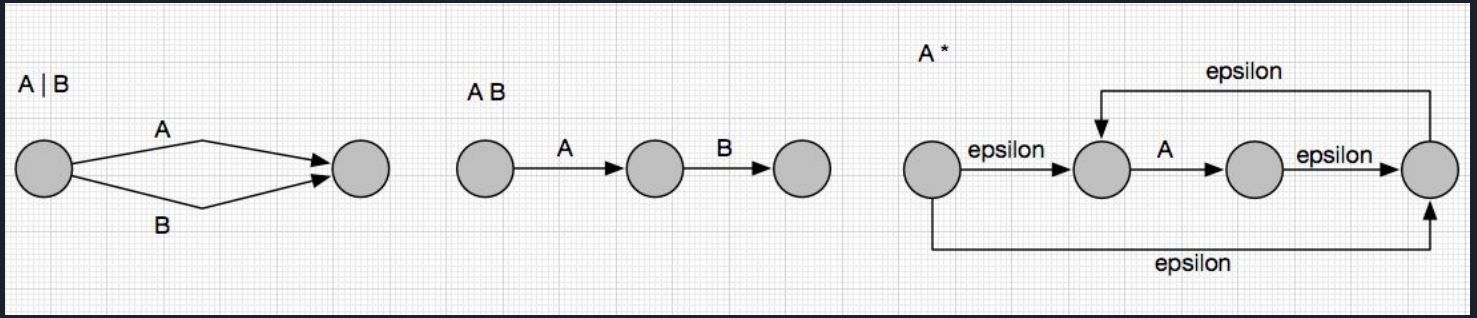
1. Recognize substrings corresponding to tokens
2. Return the value or lexeme of the token

Implementation



Regx to NFA

There are three basic operations:



Regx to NFA

Example1

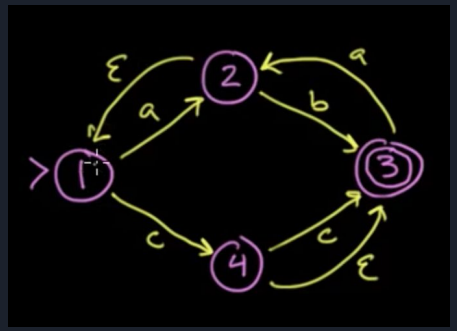
You are given the regular expression: (a | b)*abb, please draw the NFA.

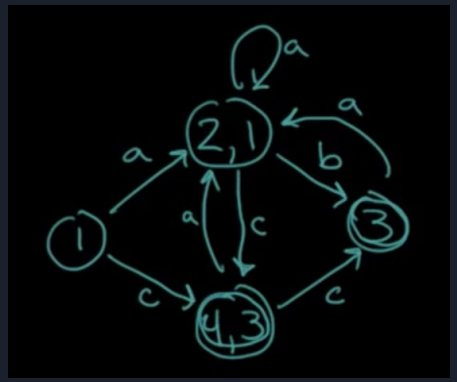
Regx to NFA

Example2

You are given the regular expression: letter alphanum*, please draw the NFA.

NFA to DFA





	a	b	c	ε^*
>1	2	—	4	1
2	—	3	—	2,1
③	2	—	—	3
4	—	—	3	4,3

	$a\varepsilon^*$	$b\varepsilon^*$	$c\varepsilon^*$
> 1	2,1	—	4,3
2,1	2,1	3	4,3
4,3	2,1	—	3
③	2,1	—	—

Regx to DFA

Regx: **letter (letter | digit)***

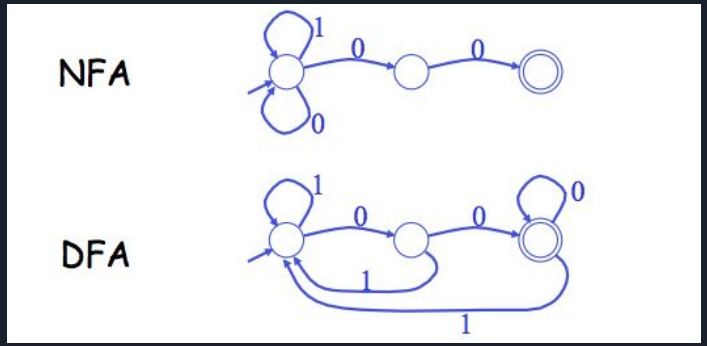
NFA vs. DFA

Nondeterministic Finite Automata

- Can have multiple transitions for one input in a given state
- Can have ε-move

Deterministic Finite Automata

- One transition per input per state
- No ε-move

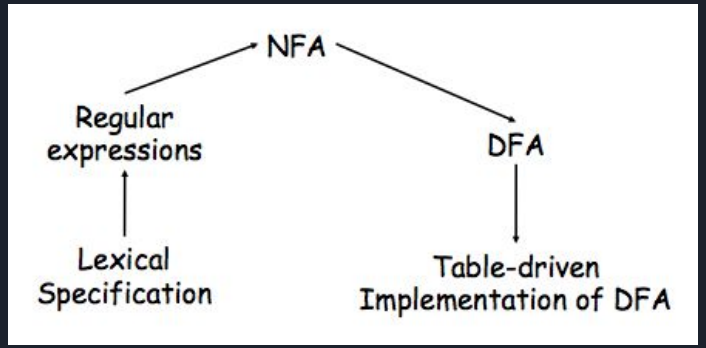


NFAs and DFAs recognize the same set of languages, but DFAs are faster to execute.

Lexical Analyser

Once you have the transition table, you should be able to implement the state machine which is the most important part in lexical analysis.

- nextToken( )
- nextChar( )
- backupChar( )
- isFinalState(state)
- table(currentState, comingChar)
- createToken(state)



An useful tool --- AtoCC

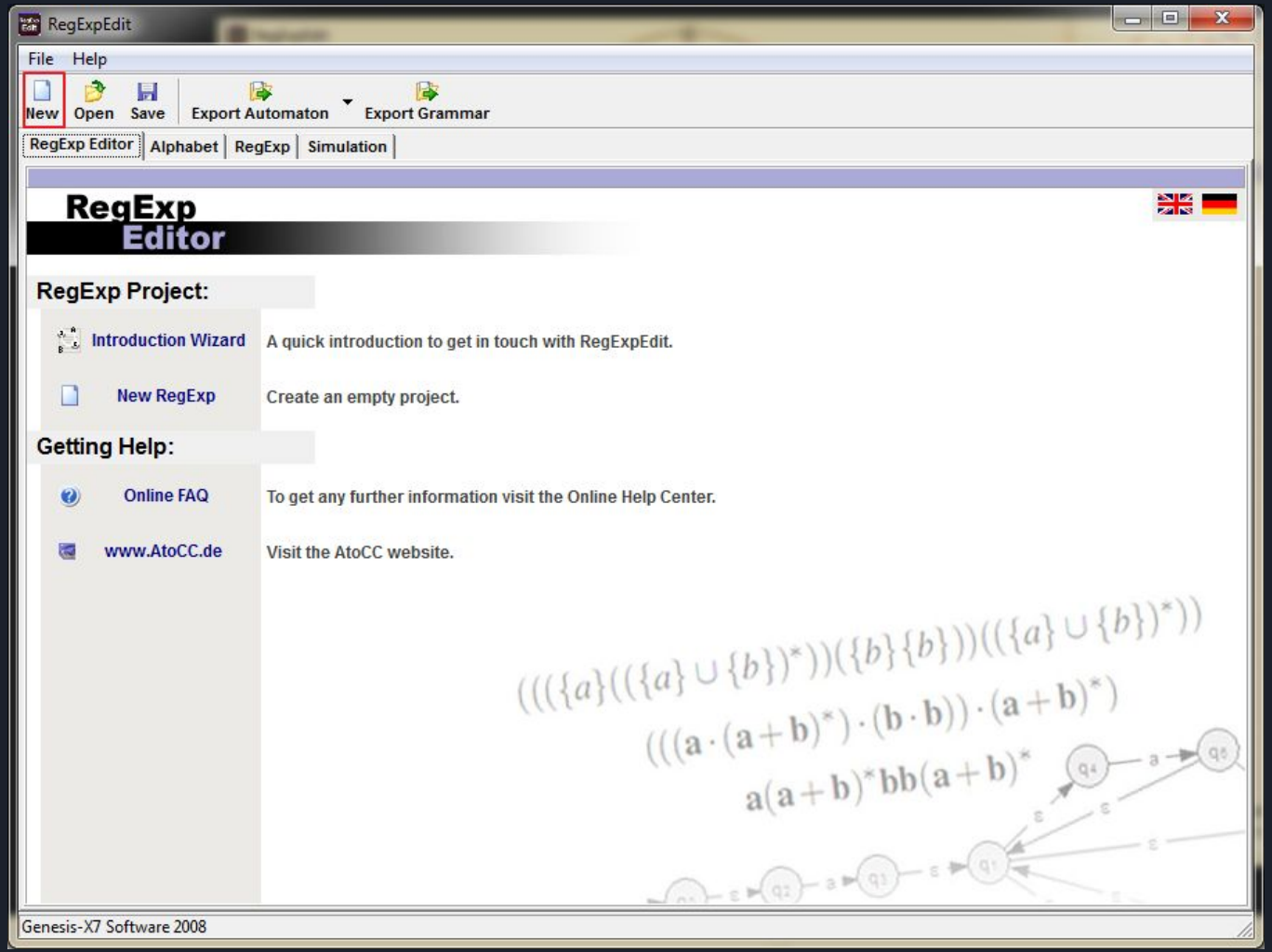
The learning environment can be of use in teaching abstract automata, formal languages, and some of its applications in compiler construction. From a teacher's perspective AtoCC aims to address a broad range of different learning activities forcing the students to actively interact with the subjects being taught.

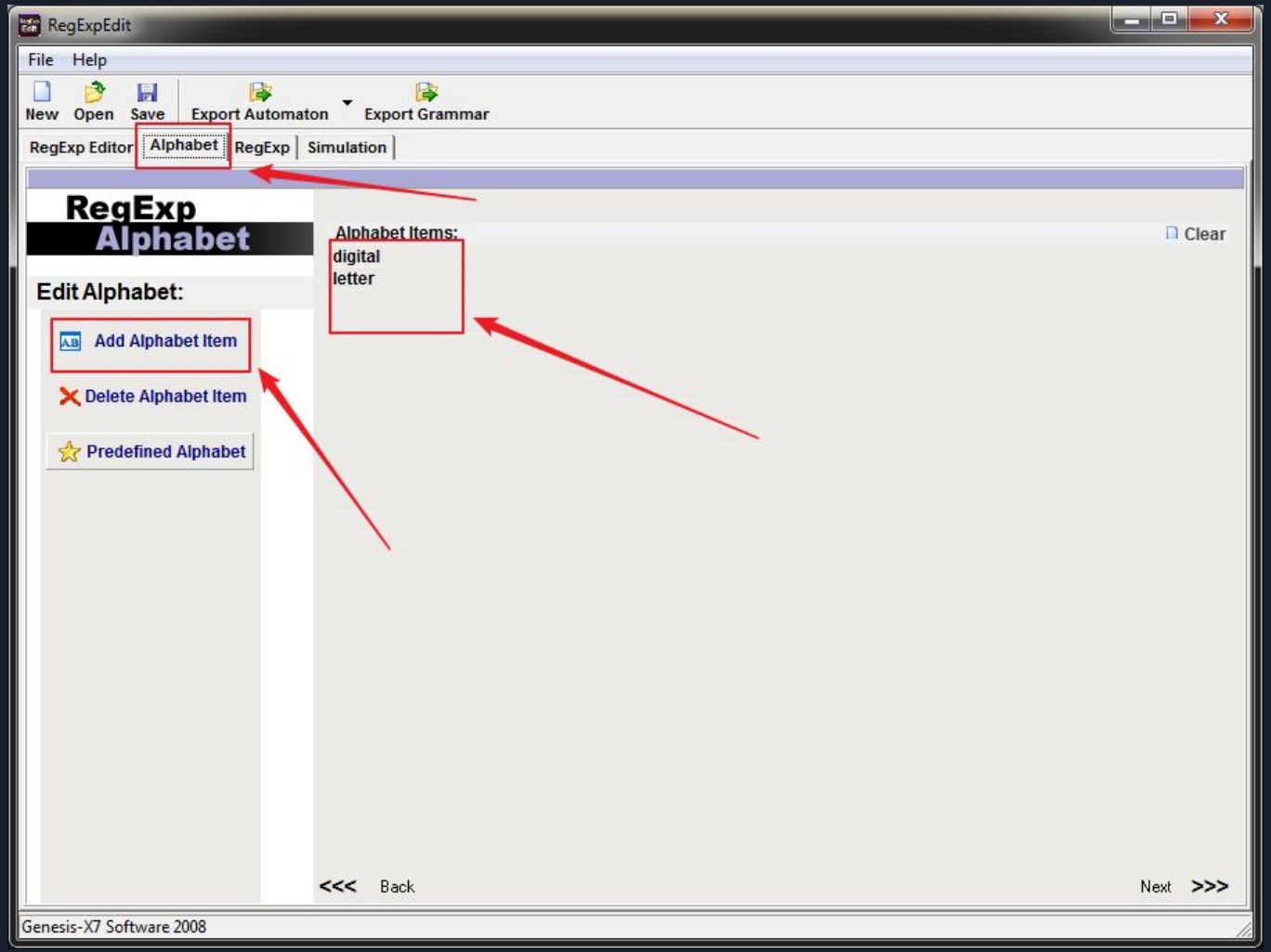
Note: We will need to use it for assignment 2 for grammar verification (will explain into detail when you receive assignment 2)

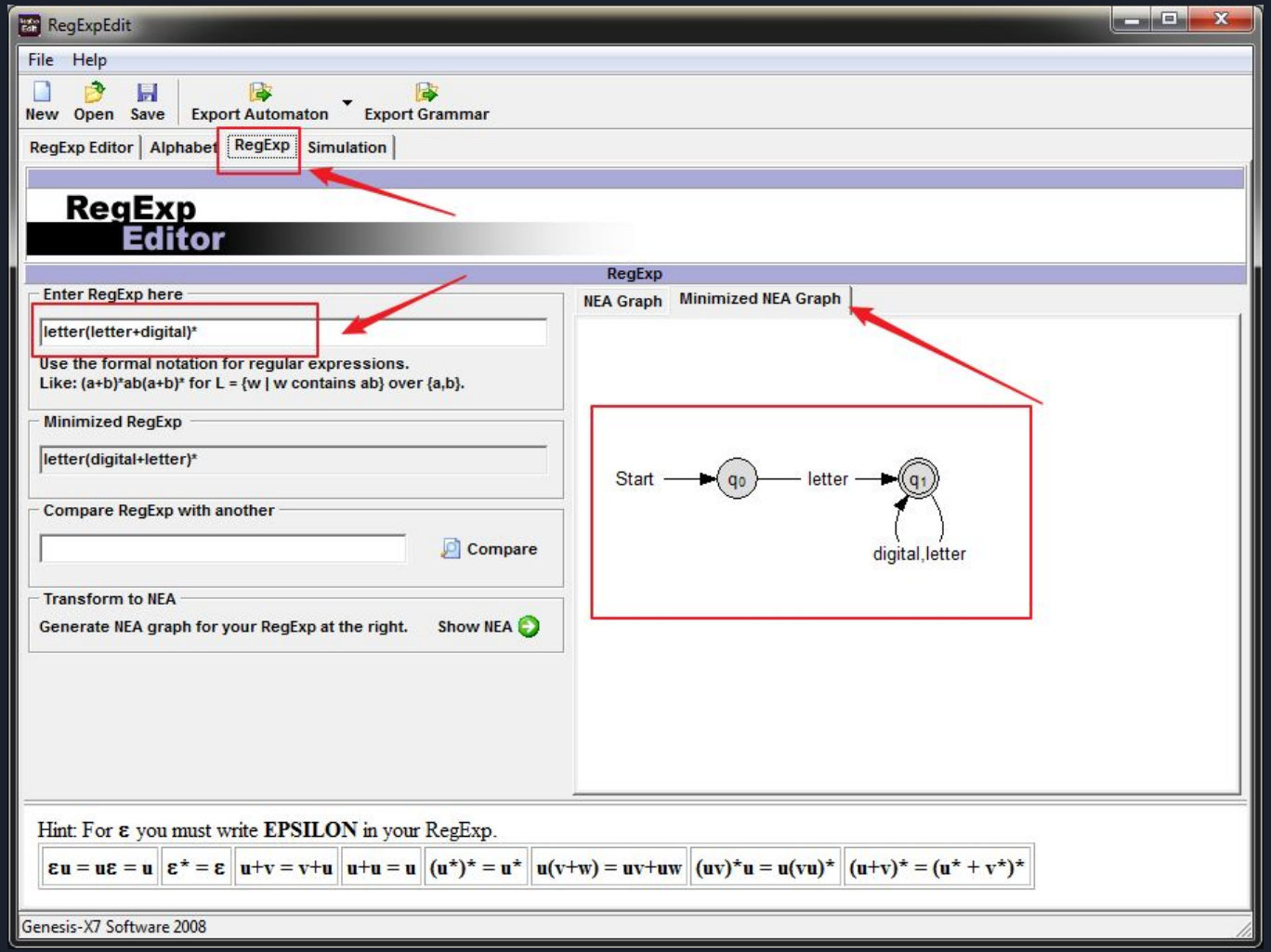
http://www.atocc.de/cgi-bin/atocc/site.cgi?lang=en&site=main

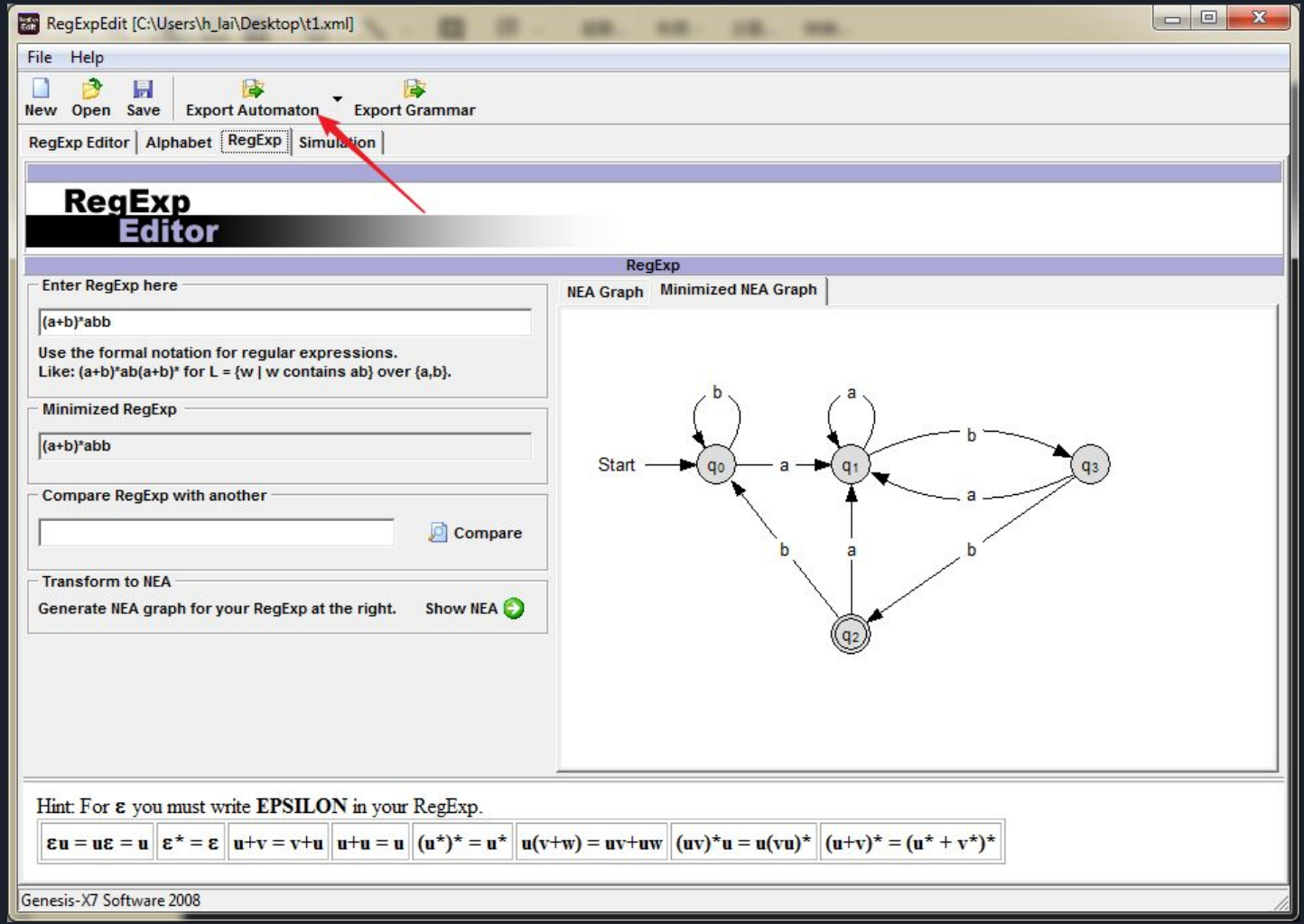
AtoCC --- RegExp Edit

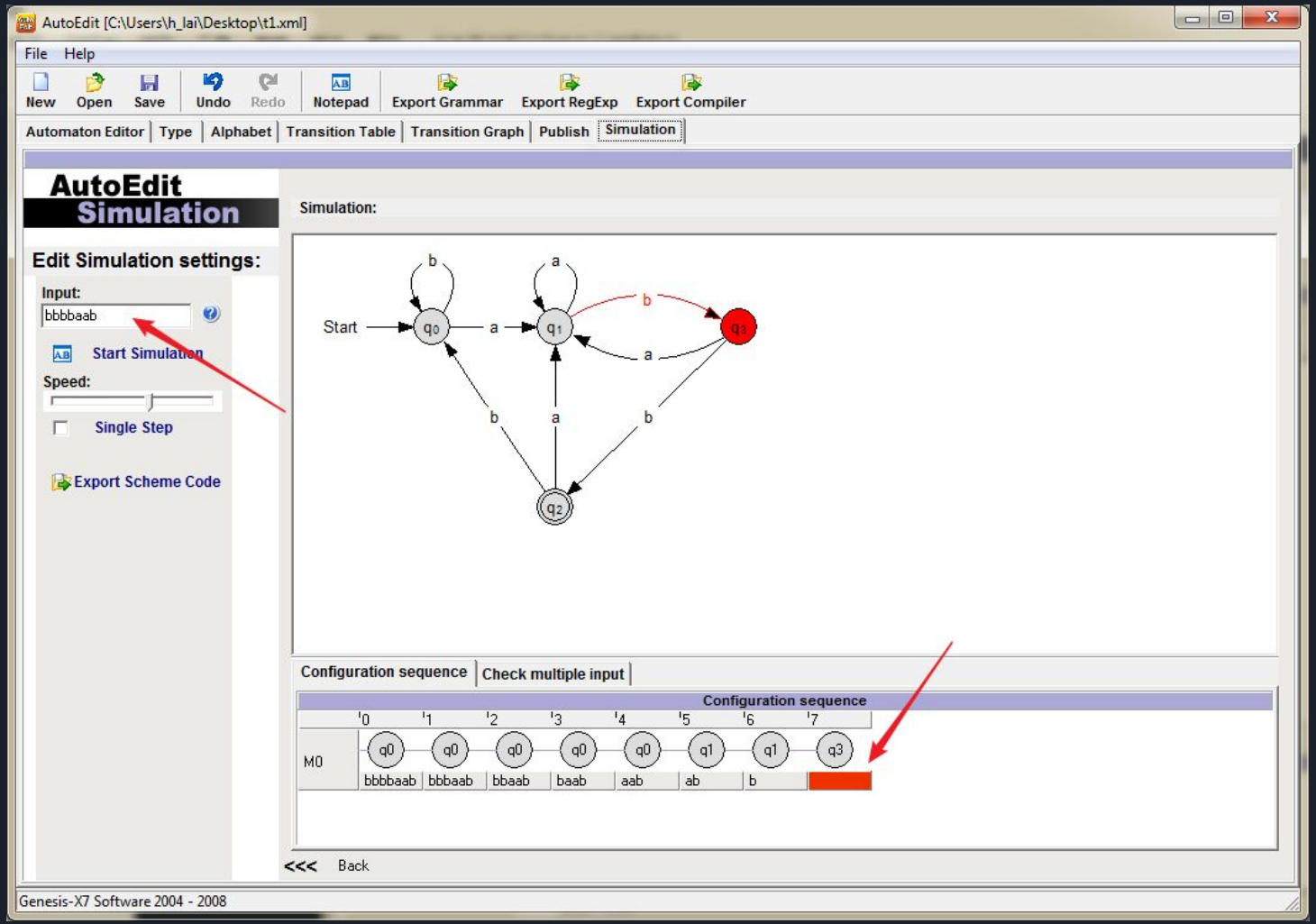
It is a powerful tool that we can use to generate DFA from regular expression and validate your work. In the following slides you will find screenshots on how to use this tool in order to create a DFA from a regular expression that should conform to the lexical specification of the language.

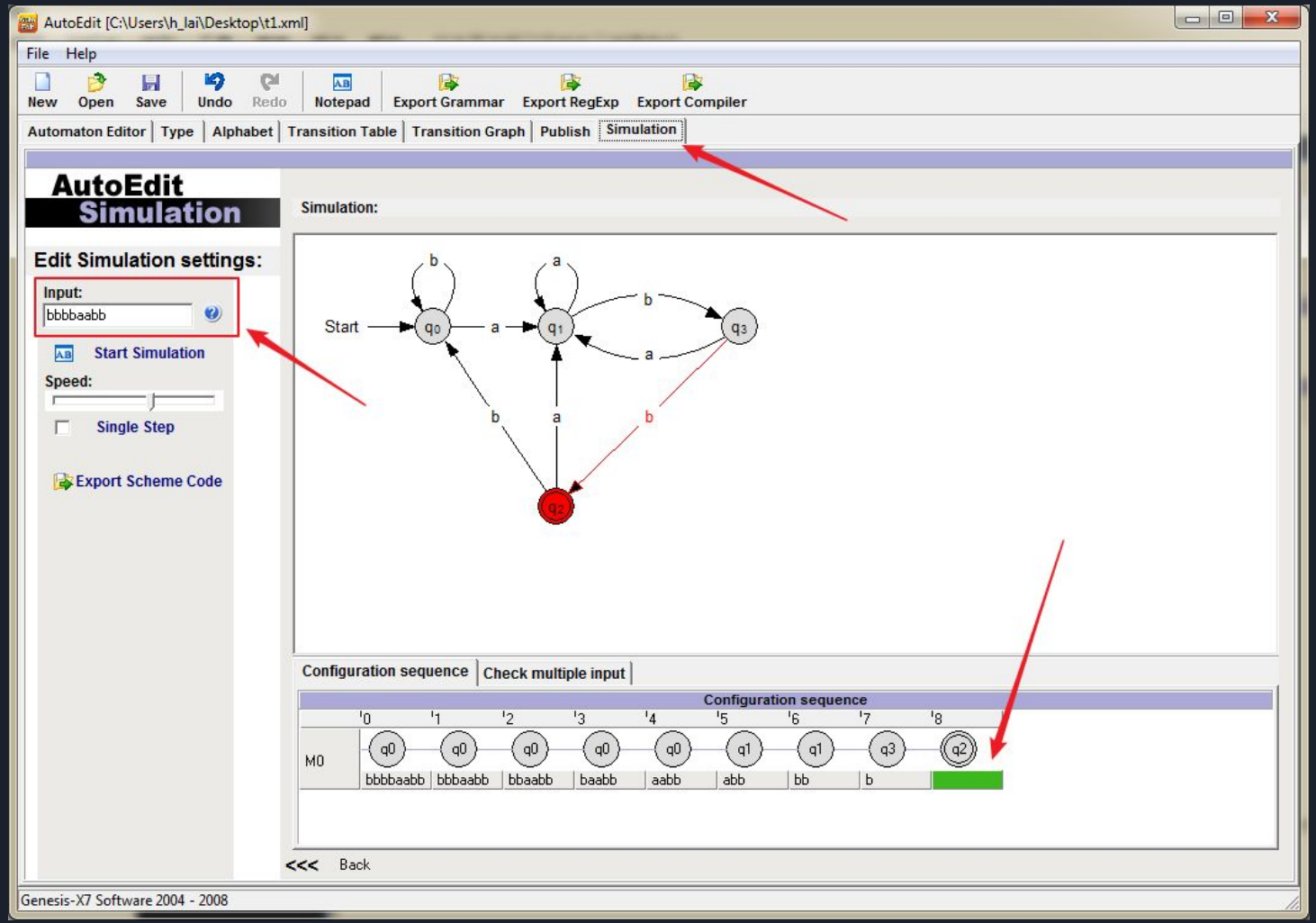

RegExpEdit
File
Help
New
Open
Save
Export Automaton
Export Grammar
RegExp Editor
Alphabet
RegExp
Simulation
RegExp
Editor
RegExp Project:
Introduction Wizard
A quick introduction to get in touch with RegExpEdit.
New RegExp
Create an empty project.
Getting Help:
Online FAQ
To get any further information visit the Online Help Center.
www.AtoCC.de
Visit the AtoCC website.
Genesis-X7 Software 2008


RegExpEdit
File
Help
New
Open
Save
Export Automaton
Export Grammar
RegExp Editor
Alphabet
RegExp
Simulation
RegExp
Alphabet
Edit Alphabet:
Add Alphabet Item
Delete Alphabet Item
Predefined Alphabet
Alphabet Items:
digital
letter
Clear
<<< Back
Next >>>
Genesis-X7 Software 2008


RegExpEdit
File Help
New Open Save Export Automaton Export Grammar
RegExp Editor Alphabet RegExp Simulation
RegExp Editor
RegExp
Enter RegExp here
letter(letter+digital)*
Use the formal notation for regular expressions.
Like: (a+b)*ab(a+b)* for L = {w | w contains ab} over {a,b}.
Minimized RegExp
letter(digital+letter)*
Compare RegExp with another
Compare
Transform to NEA
Generate NEA graph for your RegExp at the right.
Show NEA
NEA Graph
Minimized NEA Graph
Start
q0
letter
q1
digital,letter
Hint: For ε you must write EPSILON in your RegExp.
εu = uε = u
ε* = ε
u+v = v+u
u+u = u
(u*)* = u*
u(v+w) = uv+uw
(uv)*u = u(vu)*
(u+v)* = (u* + v*)*
Genesis-X7 Software 2008


RegExpEdit [C:\Users\h_lai\Desktop\t1.xml]
File Help
New Open Save Export Automaton Export Grammar
RegExp Editor Alphabet RegExp Simulation
RegExp Editor
RegExp
Enter RegExp here
(a+b)*abb
Use the formal notation for regular expressions.
Like: (a+b)*ab(a+b)* for L = {w | w contains ab} over {a,b}.
Minimized RegExp
(a+b)*abb
Compare RegExp with another
Compare
Transform to NEA
Generate NEA graph for your RegExp at the right.
Show NEA
NEA Graph
Minimized NEA Graph
Start
q0
q1
q2
q3
a
b
Hint: For ε you must write EPSILON in your RegExp.
εu = uε = u
ε* = ε
u+v = v+u
u+u = u
(u*)* = u*
u(v+w) = uv+uw
(uv)*u = u(vu)*
(u+v)* = (u* + v*)*
Genesis-X7 Software 2008


AutoEdit [C:\Users\h_lai\Desktop\t1.xml]
File Help
New Open Save Undo Redo Notepad Export Grammar Export RegExp Export Compiler
Automaton Editor Type Alphabet Transition Table Transition Graph Publish Simulation
AutoEdit
Simulation
Edit Simulation settings:
Input:
bbbbaab
Start Simulation
Speed:
Single Step
Export Scheme Code
Simulation:
Start
q0
q1
q2
q3
Configuration sequence
Check multiple input
Configuration sequence
M0
bbbbaab
bbbaab
bbaab
baab
aab
ab
b
<<< Back
Genesis-X7 Software 2004 - 2008


AutoEdit [C:\Users\h_lai\Desktop\t1.xml]
File Help
New Open Save Undo Redo Notepad Export Grammar Export RegExp Export Compiler
Automaton Editor Type Alphabet Transition Table Transition Graph Publish Simulation
AutoEdit
Simulation
Edit Simulation settings:
Input:
bbbbaabb
Start Simulation
Speed:
Single Step
Export Scheme Code
Simulation:
Start
q0 q1 q2 q3
Configuration sequence
Check multiple input
Configuration sequence
M0
bbbbaabb bbbaabb bbaabb baabb aabb abb bb b
<<< Back
Genesis-X7 Software 2004 - 2008

Automaton Editor | Type | Alphabet | Transition Table | Transition Graph | Publish | Simulation

AutoEdit Publish

Edit Publish settings:

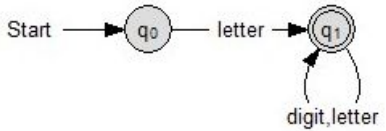
- Change Font
- Export Graph ...
- Export Definition ...
- Export Transitions ...
- Export as HTML

HTML Publish:

Automaton

Type: NEA

Transition Graph:



Definition: $M = (\{q_0,q_1\},\{digit,letter\},\delta,q_0,\{q_1\})$

Transition Table:

δ	digit	letter
q_0	{}	$\{q_1\}$
q_1	$\{q_1\}$	$\{q_1\}$

Grammar: G = (N,T,P,s)

G = ({q0,q1},{digit,letter},P,q0)

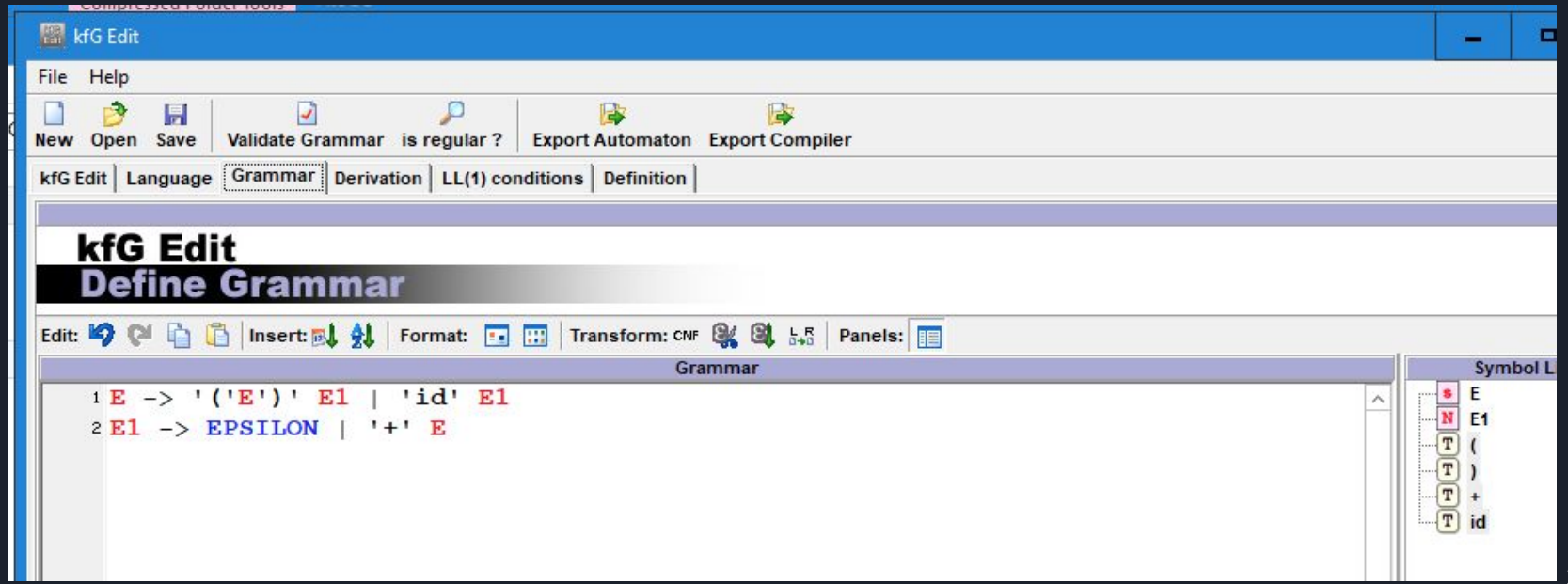
P = {
  q0 -> letter q1 | letter
  q1 -> digit q1 | digit | letter q1 | letter
}

RegExp: letter(digit+letter)*

<<< Back Next >>>

AtoCC Format

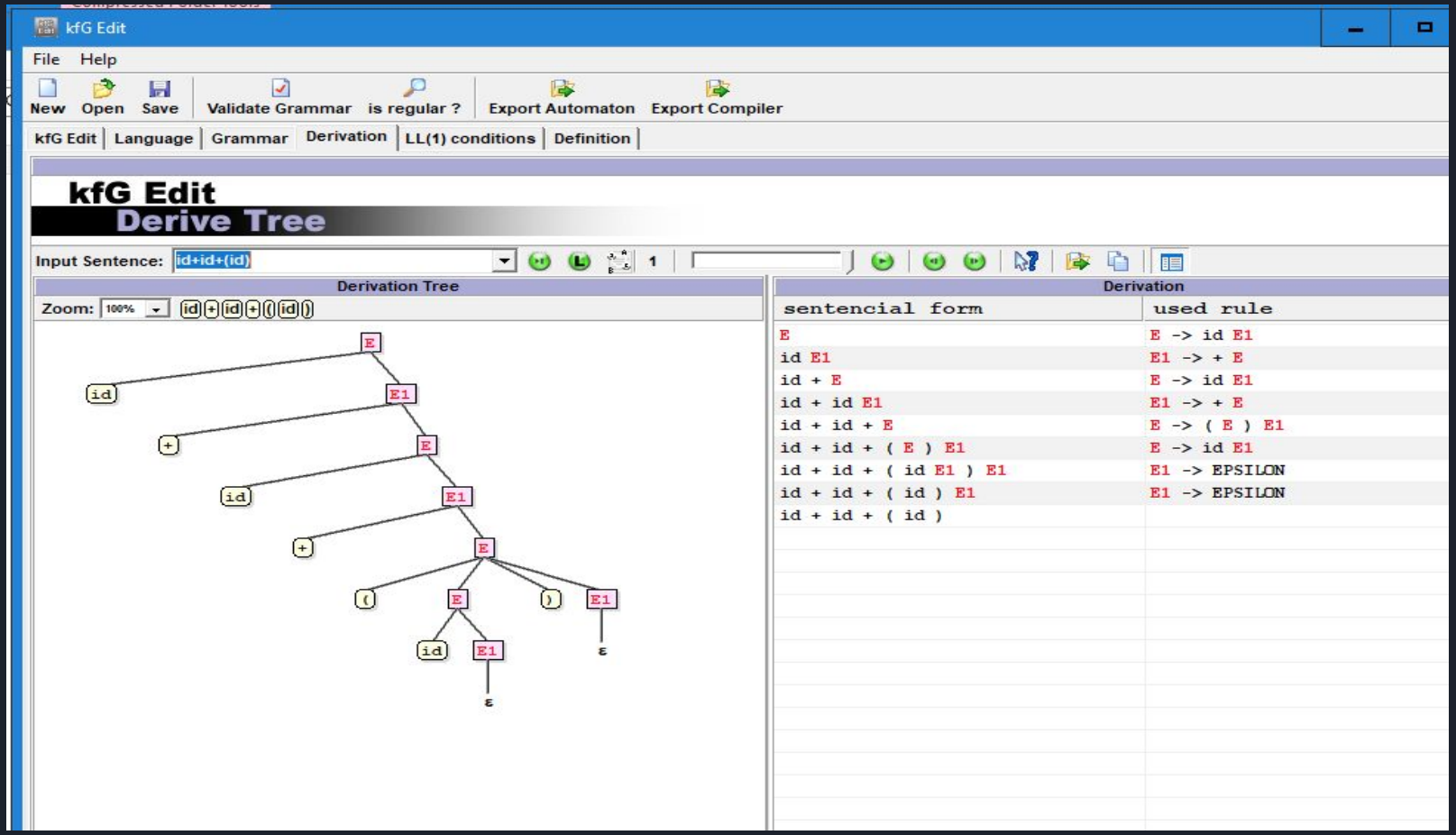
In assignment 2, you will have to use AtoCC to verify your grammar. For example, you will enter your grammar in the kfgEdit tool like this (simple grammar shown):



Continued --- AtoCC Format

Then, one very convenient thing that this tool allows is to verify a string against the grammar, by inputting a string in the "input sentence" field in the Derivation tab window (see the image in the next page)

This allows the tool to verify if this string is parsable or not, and if it doesn’t generate a tree and a derivation for it. What I want the lexical analyzer to output is a string that you can copy in the"input sentence" box. This way, you can verify if your grammar is correct by using your lexical analyzer to output a string representing the token stream, and the kfgEdit tool to verify that you grammar can parse it.


kfG Edit
File Help
New Open Save Validate Grammar is regular ? Export Automaton Export Compiler
kfG Edit | Language | Grammar | Derivation | LL(1) conditions | Definition
kfG Edit
Derive Tree
Input Sentence: id+id+(id)
Derivation Tree
Zoom: 100%
Derivation
sentencial form | used rule
E | E -> id E1
id E1 | E1 -> + E
id + E | E -> id E1
id + id E1 | E1 -> + E
id + id + E | E -> (E) E1
id + id + (E) E1 | E -> id E1
id + id + (id E1) E1 | E1 -> EPSILON
id + id + (id) E1 | E1 -> EPSILON
id + id + (id)

Implementation of lexical analyzer

Two ways to implement the lexical analyzer:

1. Table driven (but constructing a transition table by hand is not an easy job)
2. Handwritten (it require you to be very careful considering all the possible situations)

Note: It is your choice to pick one of the methods to implement and your choice will not affect the prospective assignments. The output of the Scanner is the stream of tokens which can be accessed when the nextToken() method being called.

Reference

https://users.encs.concordia.ca/~paquet/wiki/images/1/19/COMP442-6421.2.lexical.ppt
http://web.stanford.edu/class/cs143/lectures/lecture03.pdf
http://web.stanford.edu/class/cs143/lectures/lecture04.pdf
https://www.youtube.com/watch?v=dIH2pIndNrU
https://www.youtube.com/watch?v=taClnxU-nao
https://www.youtube.com/watch?v=RYNN-tb9WxI
https://github.com/laihaotao/compiler_design.git