RAMAIAH UNIVERSITY OF APPLIED SCIENCES

Assignment

Course Code	CSC211A
Course Name	Formal Languages and Automata Theory
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Satyajit Ghana
Reg. No.	17ETCS002159
Semester/Year	04/2017
Course Leader(s)	P. Padma Priya Dharishini

Declaration Sheet

Student Name	Satyajit Ghana		
Reg. No	17ETCS002159		
Programme	B.Tech	Semester/Year	04/2017
Course Code	CSC211A		
Course Title	Formal Languages and Automata Theory		
Course Date		to	
Course Leader	P. Padma Priya Dharishini		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Contents

List Of Figures

1 Question 1

Solution to Question No. 1 Part A

1.1 Introduction

Formal grammars, in particular context-free grammars, are the tools most frequently used to describe the structure of programs. They permit a lucid representation of that structure in the form of parse trees, and one can (for the most part mechanically) specify automata that will accept all correctly-structured programs (and only these). The automata are easy to modify so that they output any convenient encoding of the parse tree.

A grammar specifies a process for generating sentences, and thus allows us to give a finite description of an infinite language. The analysis phase of the compiler, however, must recognize the phrase structure of a given sentence: It must parse the sentence. Assuming that the language has been described by a grammar, we are interested in techniques for automatically generating a recognizer from that grammar. There are two reasons for this requirement:

- It provides a guarantee that the language recognized by the compiler is identical to that defined by the grammar.
- It simplifies the task of the compiler writer.

1.2 Discussion on FLAT aids on designing compilers for programming languages

A Compiler is a program that reads a program written in one language- the source language- and translates it into an equivalent program in another language- the target language. Compilation can be divided into two parts: Analysis and Synthesis.

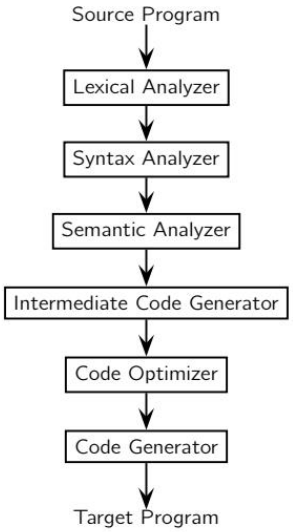


Figure 1-1 Architecture of a Compiler

The Analysis part is what we are interested in which contains Lexical Analysis, Syntax Analysis and Semantic Analysis.

Lexical Analysis

The program is considered as a unique sequence of characters, the analyzer reads the program from left-to-right and sequence of characters are grouped into tokens – lexical units with a collective meaning.

Syntactic Analysis

This is also known as Parsing; the tokens are grouped into grammatical phrases represented by a Parse Tree which gives a hierarchical structure to the source program.

The hierarchical structure is expressed by recursive rules, called Grammar's Productions.

Example: Grammar's Productions for assignment statements are:

```
<assignemnt> -> ID "=" <expr>
<expr>       -> ID | NUM | <expr> <op> <expr> | (<expr>)
<op>         -> + | - | * | /
```

The Parse Tree for the same would be,

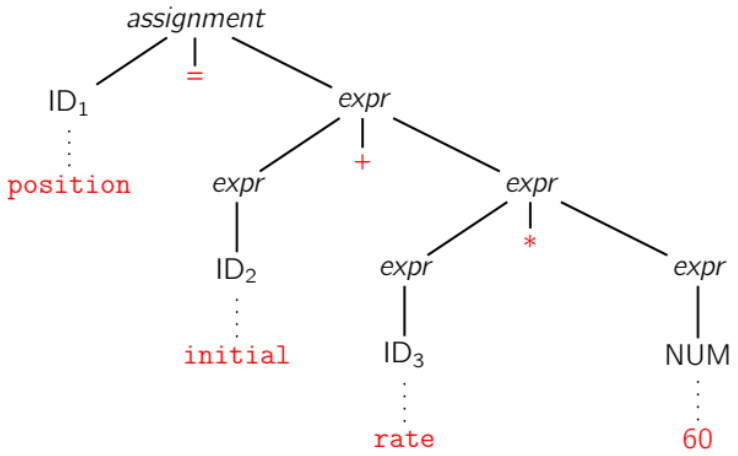


Figure 1-2 Parse Tree

All of these are done using Formal Language Concepts such as Parse Trees, CFG, PDA, NFA, Bottom-Up Analysis, LR(k) Grammars.

(Alessandro Artale, Free University of Bozen-Bolzano)

Here we've used automata as models for the parsing process of the possible input strings given in the programming language, this verifies the syntactical nature of the language thus defined.

BNF notation was first used to describe ALGOL 60 [Naur, 1963]. Many authors have proposed extensions similar to our EBNF, using quoted terminals rather than bracketed non-terminals and having a regular expression capability. EBNF definitions are usually shorter than their BNF equivalents, but the important point is that they are textual representations of syntax charts [Jensen and Wirth, 1974; ANSI, 1978a]. This means that the context-free grammar can actually be developed and described to the user by means of pictures.

Pushdown automata were first examined by SAMELSON and BAUER [1960] and applied to the compilation of a forerunner of ALGOL 60. Theoretical mastery of the concepts and the proofs of equivalence to general context-free grammars followed later. Our introduction of LR(k) grammars via reduction classes follows the work of LANGMAACK [1971]. AHO and ULLMAN [1972] (and many other books dealing with formal languages) cover essentially the same material as this chapter, but in much greater detail. The proofs that are either outlined here or omitted entirely can be found in those texts.

1.3 Conclusion

The basic symbols of a programming language are often described by arbitrary context-free productions, as illustrated by the LAX. This description does not provide a suitable starting point for mechanical construction of a lexical analyzer, and must therefore be recast by hand in terms of a regular set or regular grammar. Our interpretation of finite automata and pushdown automata as special cases of general rewriting systems follow SALOMAA [1973]. By this means we avoid a special definition of concepts such as configurations or transitions of an automaton.

(William M. Waite, Gerhard Goos, Compiler Construction)

2 Question 2

Solution to Question 1 Part B

2.1 Introduction

A Seat-Belt Controller system is to be made with the following constrains,

- Initially SBC is in idle state.
- When a person is seated, not fasten the seat belt within 'x' time units and engine is ON, SBC is responsible for automatically switch off the engine.
- On fastening of seat belt, SBC allow the person to switch ON the engine.
- When a person is seated, not fasten the seat belt within 'x' time units and engine is OFF, SBC is responsible for raising an alarm.
- On fastening of seat belt, SBC has to switch off the alarm
- When a person is not in seat then SBC has to be in idle state

The problem requires us to make either a PDA/NDPDA for the seat belt controller process. Since it's easier to design a ND-PDA (Non-Deterministic Push Down Automata), we would be doing so in this problem.

2.2 Problem Solving Approach

A ND-PDA differs from an NFA in the case of memory, PDA has an infinite memory stack to keep track of previous transitions while NFA does not. We use this feature of the PDA to keep track of the different states of the Seat-Belt Controller.

The different states chosen for the Automata are,

IDLE, SEATED, FASTENED, ENGINE_OFF, ENGINE_ON, WAIT_STATE, ALARM_ON, ALARM_OFF, SBC_OK

To keep track if the stack is empty we use Z as the symbol in the stack, which indicates that the stack is empty.

The approach in the form of an Algorithm:

Step 1: IDLE

Step 2: Read seated and push seated to stack

Step 3: Read fastened and push fastened to stack

Step 4: If fastened and seated in stack and ENGINE_ON then turn on Engine

Step 5: If only Seated in stack and ENGINE_OFF wait for x units of time, if not fastened then raise alarm.

Step 6: When fastened turn off alarm

Step 7: If only Seated in stack and ENGINE _ON wait for x units of time and go to ENGINE_OFF.

Step 8: If the stack is empty at any of the steps above then go to final state SBC_OK.

To keep track if the user is seated, fastened, and for x units of time wait, we use 3 different stack symbols.

When the user is seated, seated is pushed to the stack, now whenever we want to check if the user was seated or not we can check for this symbol in the stack, if the user unseats, then we can pop seated from the stack.

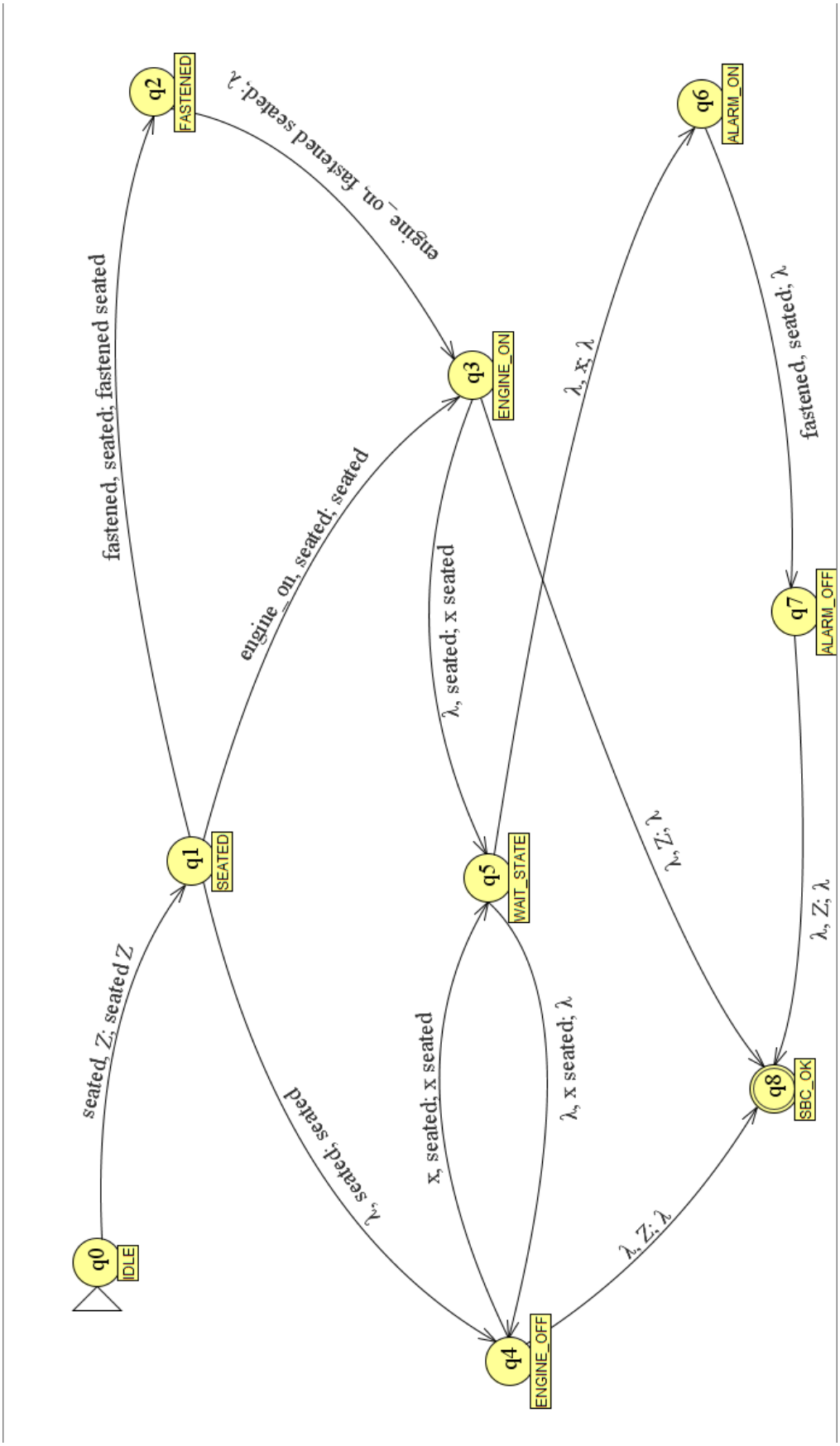
A similar logic applies for fastened. For example, when the Engine is turned ON, we can check in the stack if fastened and seated exists, then the user is allowed to keep the engine on and the final state SBC_OK is reached in this case.

Since this is a normal ND-PDA we cannot keep track of time, in that case we would require a Timed Non-Deterministic Push Down Automata, which is not in the scope of this assignment, instead we push 'x' into the stack to inform that we are waiting for x units of time and go to the WAIT_STATE, here we can check for further inputs, if no inputs are given the element 'x' is popped and we transition to the timed_out state, whichever that may be, in this question it is ALARM_ON.

2.3 Design and Validation

The Design for the above approach was done in JFLAP 8.0 beta

2.3.1 Design


q0
IDLE
q1
SEATED
q2
FASTENED
q3
ENGINE_ON
q4
ENGINE_OFF
q5
WAIT_STATE
q6
ALARM_ON
q7
ALARM_OFF
q8
SBC_OK
seated, Z; seated Z
fastened, seated; fastened seated
engine_on, fastened seated; λ
engine_on, seated; seated
λ, seated; x seated
x, seated; x seated
λ, x seated; λ
λ, x; λ
λ, Z; λ
fastened, seated; λ
λ, Z; λ
λ, Z; λ

The Design is very simple for the reason that we have used PDA's instead of NFA's, the number of states required are way less as we can use the Stack to keep track of different previous user inputs.

$$\Sigma = \{\text{engine\_on}, \text{fastened}, \text{seated}, \text{x}\}$$
$$\Gamma = \{\text{Z}, \text{fastened}, \text{seated}, \text{x}\}$$
$$\text{z} = \text{Z}$$

A Push Down Automata is a 7-tuple structure

$$\text{PDA} = (\text{Q}, \Sigma, \Gamma, \delta, \text{S}, \text{z}, \text{F})$$

The Final State is,

$$\text{F} = \text{q}_8$$

The States are,

$$Q = \{q_i \mid i = 0,1,2..8\}$$

The Start state is,

$$\text{S} = \text{q}_0$$

The transition functions are,

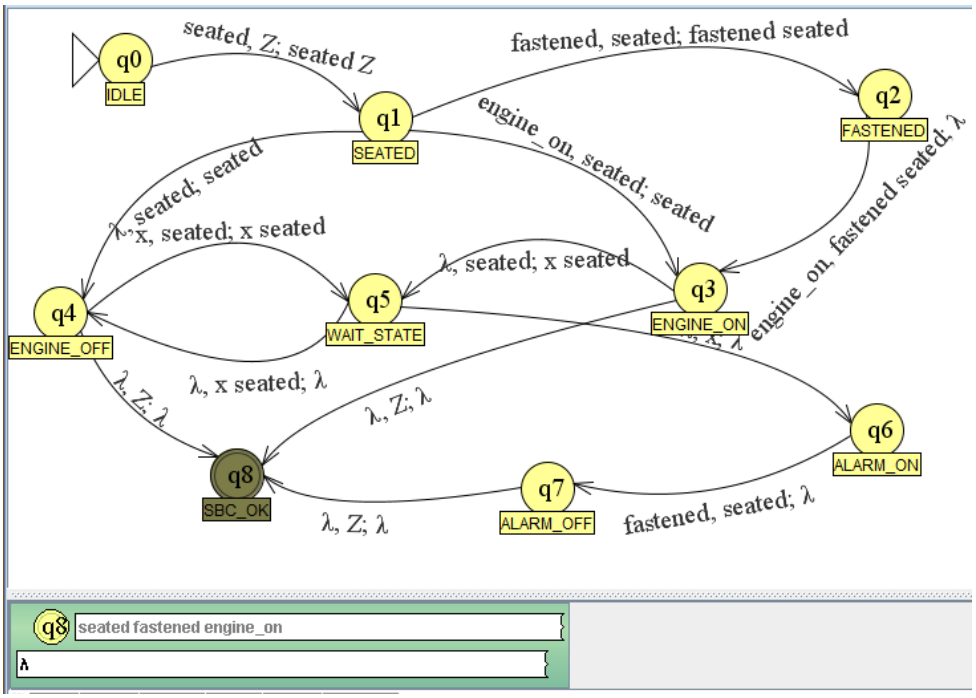
$$\delta\ \text{q}_0, \text{seated}, \text{Z} = \text{q}_1, \text{seated Z}$$
$$\delta\ \text{q}_1, \text{fastened}, \text{seated} = \text{q}_2, \text{fastened seated}$$
$$\delta\ \text{q}_1, \text{engine\_on}, \text{seated} = \text{q}_3, \text{seated}$$
$$\delta\ \text{q}_1, \lambda, \text{seated} = \text{q}_4, \text{seated}$$
$$\delta\ \text{q}_2, \text{engine\_on}, \text{fastened seated} = \text{q}_3, \lambda$$
$$\delta\ \text{q}_3, \lambda, \text{seated} = \text{q}_5, \text{x seated}$$
$$\delta\ \text{q}_3, \lambda, \text{Z} = \text{q}_8, \lambda$$
$$\delta\ \text{q}_4, \text{x}, \text{seated} = \text{q}_5, \text{x seated}$$
$$\delta\ \text{q}_4, \lambda, \text{Z} = \text{q}_8, \lambda$$
$$\delta\ \text{q}_5, \lambda, \text{x seated} = \text{q}_4, \lambda$$
$$\delta\ \text{q}_5, \lambda, \text{x} = \text{q}_6, \lambda$$
$$\delta\ \text{q}_6, \text{fastened}, \text{seated} = \text{q}_7, \lambda$$
$$\delta\ q_7, \lambda, Z = q_8, \lambda$$

The final state here is q_8 or SBC_OK

2.3.2 Validation

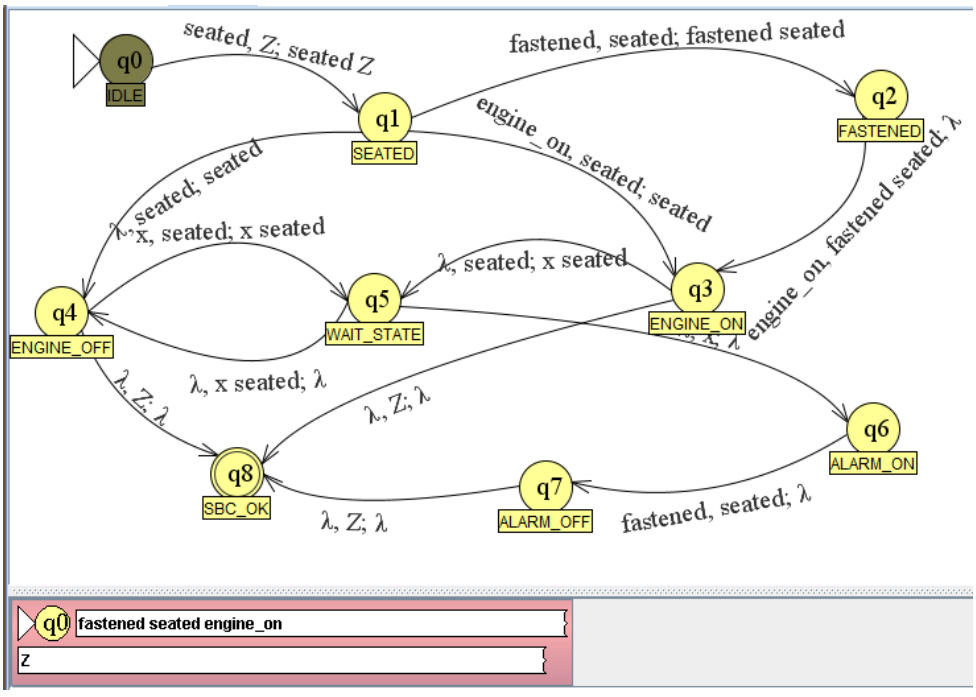
For the Validation we have taken 2 cases, one where the string is accepted and one where it is rejected.

1. w = seated fastened engine_on



Refer to Appendix for Step-Wise simulation for the given input tape.

2. w = fastened seated engine_on



2.4 Concluding Remarks

In conclusion we have created a ND-PDA for the given Seat-Belt Controller system, which was tested for some inputs in JFLAP, and successfully simulated and ended up in the final state SBC_OK.

2.4.1 Limitations

The designed PDA has a few limitations such as,

- Since this is a trivial PDA there is no concept of time here, a PDA can only work with one thing at a time, the problem requires a 'x' units of time delay which was implemented by pushed 'x' into the stack, this can be done formally by using a Timed Push-Down Automata.
- The PDA designed is a one time run thing, i.e. it has to be reset after every usage, this ca be fixed by using more symbols to store the states in the stack, although this would make it more complicated and which is out of scope of this assignment.

2.4.2 Improvements

- The PDA here has very few features, since ND-PDA's are flexible, it can be further extended to work for more inputs, also a Turing Machine can be made for better control, since in PDA's we can only work with stack's top, in Turing machines we can access any arbitrary location data, which makes it easier to handle memory.
- The timing issue can be solved using a TPDA, which is described as,

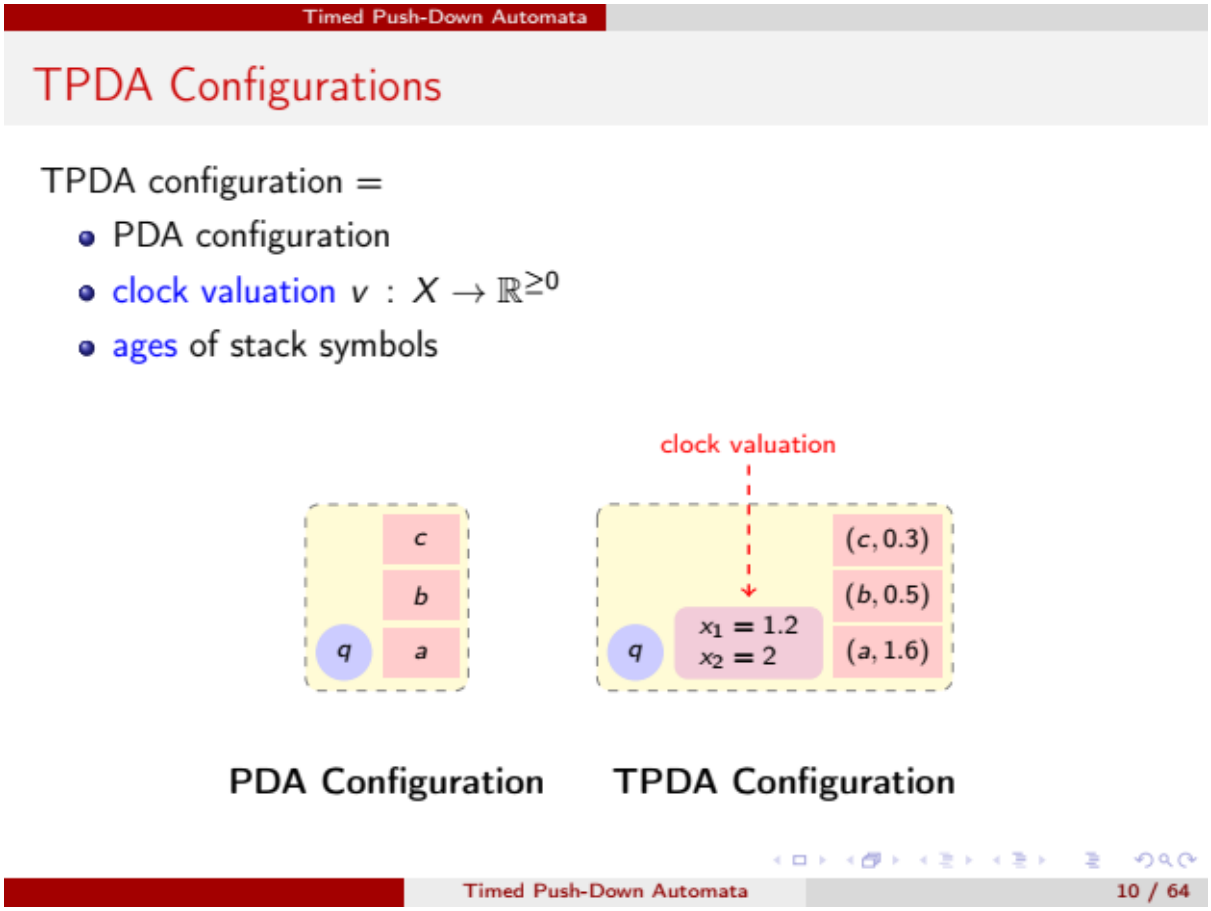


Figure 2-1 TPDA

(Parosh Aziz Abdulla, Department of Information Technology, Uppsala University, Sweden)

3 Question 3

Solution to Question 2 Part B

3.1 Introduction

In formal language theory, a Context Free Language is a language generated by some Context Free Grammar.

The set of all CFL is identical to the set of languages accepted by Push-Down Automata.

Context Free Grammar is defined by 4 tuples as $G = \{V, \Sigma, S, P\}$ where

V = Set of Variables or Non-Terminal Symbols

∑ = Set of Terminal Symbols

S = Start Symbol

P = Production Rule

Context Free Grammar has Production Rules of the form

$$\text{A} \rightarrow \text{A}$$
$$\text{where, a} = \{\text{V U } \Sigma\}^* \text{ and A} \in \text{V}$$

Here we are required to make the Context Free Grammar of the Seat Belt Controller designed previously. The Steps of doing so are described in the latter part of this assignment.

3.2 Problem solving approach

To form the CFG from the PDA, the productions in P are induced by moves of PDA as follows,

1. S productions are given by $S \rightarrow [q_0\ z\ q]$ for every $q \in Q$
2. Each erasing move $\delta\ q, a, z\ = (q', \Lambda)$ induces production $[q\ z\ q'] \rightarrow a$
3. Each non-erasing move $\delta\ q, a, z\ = (q_1, z_1 z_2 \ldots z_m)$ induces many productions of form

   $$[q\ z\ q'] \rightarrow a[q_1\ z_1\ q_2]\ [q_2\ \ z_2\ q_3] \ldots [q_m\ z_m\ q']$$

   where each state $q', q_2, \ldots q_m$ can be state in Q

Another common logic that we have to use is that in CFG,

- If the symbol pushed at the beginning is the symbol popped at the end, the stack is empty only at the beginning and the end of P's computation on x.
- Else the initially pushed symbol must get popped at some point before the end of x, and thus the stack becomes empty at this point.
- For any string x that take P from p and q, starting and ending with an empty stack, P's first move on x must be a push; the last move on x must be a pop.

3.3 Design and Validation

A PDA can be converted into a CFG to generate the Language, i.e. the grammar defines the set of strings that the automata accepts. The PDA made in 2.3.1 has to be converted to a Context Free Grammar, this is done manually since JFLAP exceeds out the time limit, as there are too many states to simplify, a few of the states are taken into consideration with the unwanted states removed for a simpler CFG.

3.3.1 Design

The Design was created using the rules we have defined in 3.2. The unnecessary rules that we generated are removed and the simplified version of the production rules are as below,

P:

1. $S \rightarrow [q_0\ z\ q_3]$
2. $[q_0\ z\ q_3] \rightarrow s\ [q_1\ s\ q_2][q_3\ z\ q_8]$
3. $[q_1\ s\ q_2] \rightarrow f\ [q_2\ fs\ q_3]$
4. $[q_1\ s\ q_4] \rightarrow \lambda$
5. $[q_1\ s\ q_2\ ] \rightarrow\ e\ [q_3\ s\ q_4][q_4\ s\ q_2]$
6. $[q_4\ s\ q_2] \rightarrow x\ [q_5\ xs\ q_4][q_1\ s\ q_2]$
7. $[q_2\ fs\ q_3] \rightarrow e$
8. $[q_3\ z\ q_8] \rightarrow \lambda$
9. $[q_4\ z\ q_8] \rightarrow \lambda$
10. $[q_5\ xs\ q_4] \rightarrow\ \lambda$
11. $[q_5\ x\ q_6] \rightarrow\ \lambda$

12. $[q_6\ s\ q_7] \rightarrow\ f$
13. $[q_7\ z\ q_8] \rightarrow\ \lambda$
14. $[q_3\ s\ q_4] \rightarrow \lambda$

Here "-" represent that any of the states from q0 to q8 can be placed here.

f = fastened

s = seated

e = engine_on

$\Sigma = \{f, s, e, \lambda\}$

$S = S$

$V = \left\{ \begin{matrix} [q_0\ z\ q_3], [q_0\ z\ q_3], [q_1\ s\ q_2], [q_1\ s\ q_4], [q_1\ s\ q_2\ ], [q_4\ s\ q_2], \\ [q_2\ fs\ q_3], [q_3\ z\ q_8], [q_4\ z\ q_8], [q_5\ xs\ q_4], [q_5\ x\ q_6], [q_6\ s\ q_7], [q_7\ z\ q_8], [q_3\ s\ q_4] \end{matrix} \right\}$

When the Stack becomes "empty" the string is accepted by the PDA.

3.3.2 Validation

Since we have already tried to validate our PDA using the input string

`seated fastened engine_on`

which is, if the user is seated, and has fastened seat-belt and turns on engine then the SBC is OK.

We should be able to generate the same using the Production Rules defined above,

$S \rightarrow [q_0\ z\ q_3]\ rule\ 1$
$\rightarrow s\ [q_1\ s\ q_2][q_3\ z\ q_8]\ rule\ 2$
$\rightarrow s\ f\ [q_2\ fs\ q_3]\ [q_3\ z\ q_8]\ rule\ 3$
$\rightarrow s\ f\ e\ [q_3\ z\ q_8]\ rule\ 7$
$\rightarrow s\ f\ e\ rule\ 8$
$\rightarrow seated\ fastened\ engineon$

Hence, we have successfully generated `seated fastened engine_on` as an accepted string by the Context Free Grammer using the Production rules defined.

Another accepted string is,

```
seated engine_on x
```

which is, if the user is seated and turns on engine, as the user has not fastened seatbelt, within x unit of time the engine is turned off and the SBC returns to OK state.

$$
\begin{aligned}
&S \rightarrow [q_0\ z\ q_3]\ rule1 \\
&\rightarrow seated\ [q_1\ s\ q_2][q_3\ z\ q_8]\ rule\ 2 \\
&\rightarrow seated\ engineon\ [q_3\ s\ q_4][q_4\ s\ q_2]\ rule\ 5 \\
&\rightarrow seated\ engineon\ [q_4\ s\ q_2]\ rule\ 14 \\
&\rightarrow seated\ engineon\ x\ [q_5\ xs\ q_4][q_3\ s\ q_4]\ rule\ 6 \\
&\rightarrow seated\ engine\ on\ x\ [q_3\ s\ q_4]\ rule\ 10 \\
&\rightarrow seated\ engineon\ x\ rule\ 14
\end{aligned}
$$

Hence here also our string is accepted, the sequence of rules has been defined above.

3.4 Concluding Remarks

The Context Free Grammar for the PDA made in 3.2 was generated, that can validate if a string can be accepted by the language or not, a few sets of combinations are taken for validating the CFG, and assuming that it would work for most of the cases, now we analyze the limitations and improvements associated with it.

3.4.1 Limitations

Since our aim was to simplify the automata, we've missed some of the production rules in the automata, it works for most of the input strings that could be expected to be worked by the seat belt controller, although not all the transitions are considered here.

3.4.2 Improvements

The CFG was far too complex to take in every possible input string, hence the automata have to be simplified further with fewer states and transition, this can be achieved by finding the Normal form of it, normal forms make it easier to handle because of the simpler structure, so we can find Chomsky Normal Form of the automata and use the CYK-algorithm.

The key advantage of Chomsky Normal Form is that every derivation of a string of n letters has exactly 2n-1 steps, thus one can determine if a string is in the language by exhaustive search of all derivations.

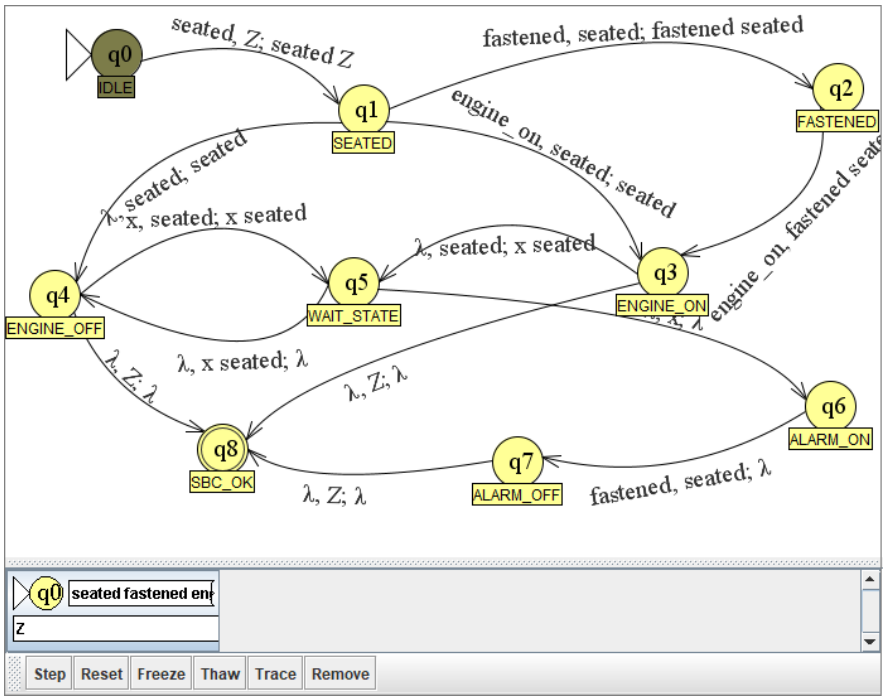
Bibliography

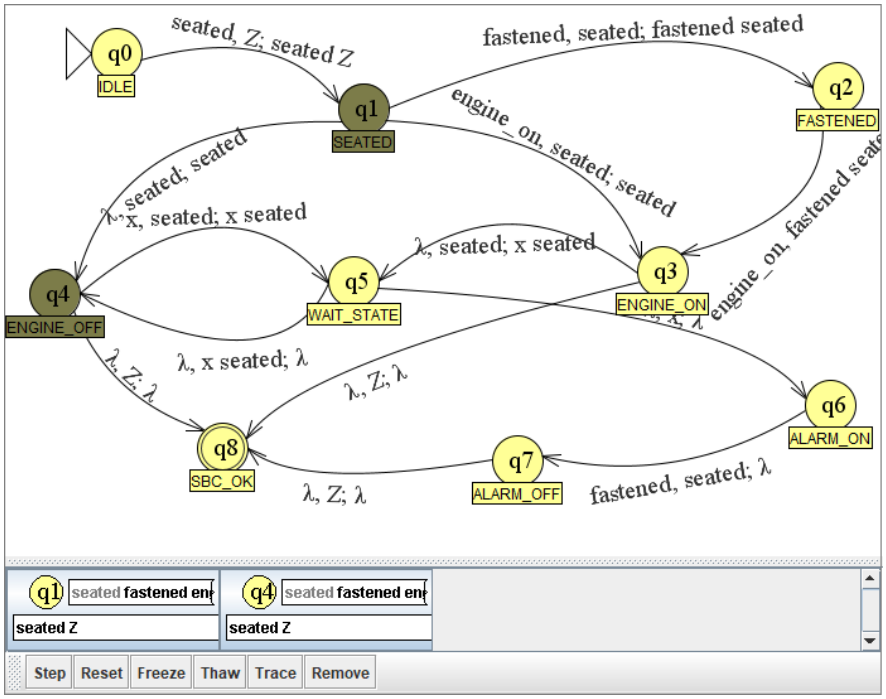
1. Alessandro Artale, Free University of Bozen-Bolzano, Faculty of Computer Science.
2. William M. Waite, Gerhard Goos, Compiler Construction, Karlsruhe, 22nd February 1996.

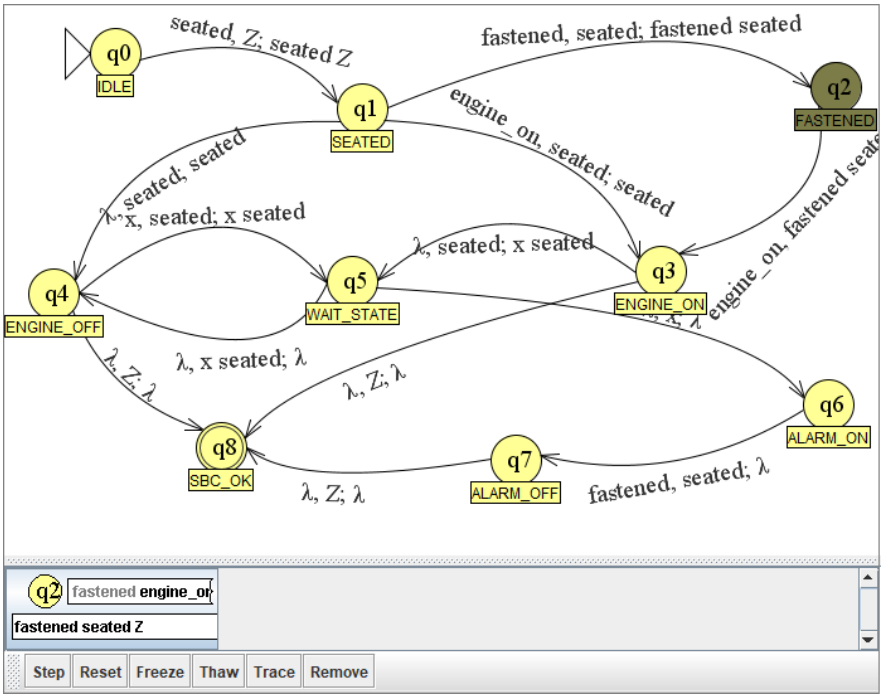
   ftp://i44ftp.info.uni-karlsruhe.de/pub/papers/ggoos/CompilerConstruction.ps.gz

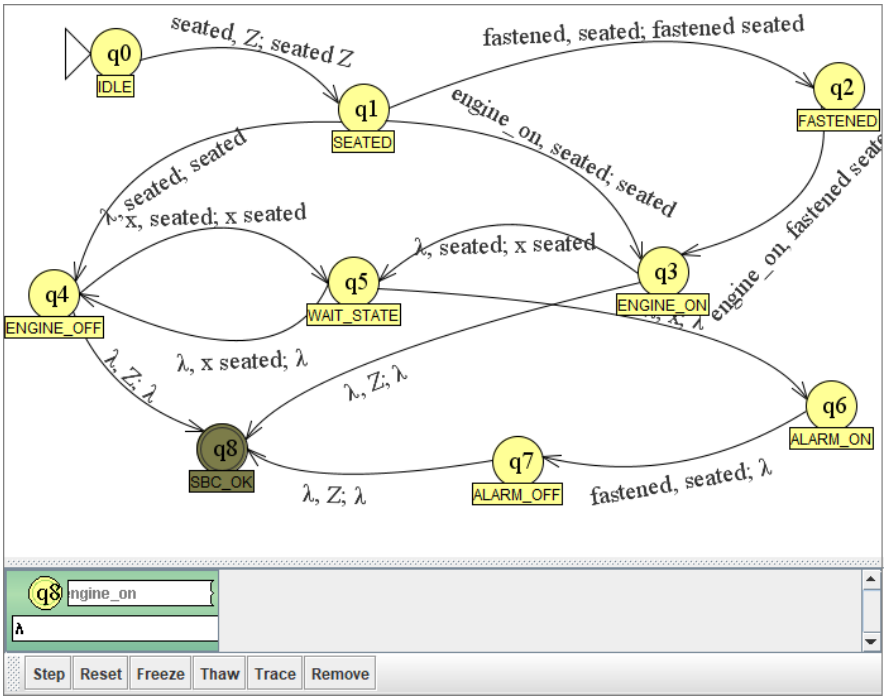
4 Appendix

4.1 PDA Validation






q0
IDLE
seated, Z; seated Z
q1
SEATED
fastened, seated; fastened seated
q2
FASTENED
engine_on, seated; seated
λ, seated; seated
λ, x, seated; x seated
λ, seated; x seated
q4
ENGINE_OFF
q5
WAIT_STATE
q3
ENGINE_ON
engine_on, fastened seate
λ, Z; λ
λ, x seated; λ
λ, Z; λ
q6
ALARM_ON
q8
SBC_OK
q7
ALARM_OFF
λ, Z; λ
fastened, seated; λ
q2
fastened engine_on
fastened seated Z
Step
Reset
Freeze
Thaw
Trace
Remove


q0
IDLE
seated, Z; seated Z
q1
SEATED
fastened, seated; fastened seated
q2
FASTENED
engine_on, seated; seated
λ, seated; seated
λ, x, seated; x seated
λ, seated; x seated
q4
ENGINE_OFF
q5
WAIT_STATE
q3
ENGINE_ON
engine_on, fastened seate
λ, Z; λ
λ, x seated; λ
λ, Z; λ
q6
ALARM_ON
q8
SBC_OK
q7
ALARM_OFF
λ, Z; λ
fastened, seated; λ
q8
engine_on
λ
Step
Reset
Freeze
Thaw
Trace
Remove