Bachelor Thesis

Pushdown Automata Simulator

Felix Erlacher (0616239)
`felix.erlacher@student.uibk.ac.at`

2 November 2009

Supervisor: DI Martin Korp

Abstract

Pushdown automata are widely used to characterize the class of context-free languages. A pushdown automaton is a finite automaton that is equipped with a stack which can record a potentially unbounded amount of information. The aim of this project is to develop a simulator which visualizes the behavior of pushdown automata. The simulator should support the two common definitions of acceptance: by empty stack and by final state. In addition it should be also possible to automatically transform a pushdown automaton that accepts by empty stack into a pushdown automaton that accepts by final state and vice versa.

Contents

1 Introduction

Everyone knows how to form a sentence in his/her native language. We learn these things in school and (almost) intuitively know whether a sentence is grammatically correct or not. For computers that is not so easy. In order to ensure that a computer program can detect grammar mistake it is necessary to describe the set of correct sentences by stating the grammar rules of the corresponding language. This can be done in different ways, for instance by using the so called *Backus-Naur Form*. It has been developed by John Backus and Peter Naur in the late 1950s to define the syntax of programming languages. For example, a simple *if-statement* can be described as follows:

```
<if-stmt>    ::= if <bool-expr> then <stmt> else <stmt>
               | if <bool-expr> then <stmt>
<bool-expr>  ::= <arith-expr> <compare-op> <arith-expr>
<stmt>       ::= <if-stmt> | <assg-stmt>
<arith-expr> ::= (<arith-expr> <arith-op> <arith-expr>)
               | <var>
<compare-op> ::= < | > | <= | >= | == | !
<assg-stmt>  ::= <var> = <arith-expr>
<var>        ::= a | b | ... | y | z
<arith-op>   ::= + | - | * |
```

Every line describes one rule. Objects `<x>` are called nonterminal symbols and generate a set of strings over some finite alphabet. For example the first rule (or production) says that an `<if-stmt>` consists of the word `if` followed by the nonterminal `<bool-expr>`, the word `then`, the nonterminal `<stmt>`, the word `else`, and finally the nonterminal `<stmt>`. The `|` indicates that there is a second, alternative way to generate a if-statement. In this case it means that the `<if-stmt>` can be produced without the else part. All of the nonterminals on the right side of the rule can, as described by the rules further down, be split up into terminals.
Coming back to our initial topic, the question arises whether the Backus-Naur Form can be used to describe natural languages. In order to answer this question we have to know that a grammar in Backus-Naur Form is nothing else than a *context-free grammar*. And indeed, the formalism of context-free grammars has been developed in the mid 1950s by Noam Chomsky based on the grammatical structures linguists used to describe languages. Of course context-free grammars cannot cope with all intricacies of languages. Nevertheless they provide a simple and precise mechanism for describing and analyzing the basic methods by which phrases in natural languages are built.

In computer science *context-free languages* (the set of languages that can be described by context-free grammars) play a central role in the description and design of programming languages and compilers. By that end it is sometimes more convenient to express context-free languages by so called *pushdown automata*—a finite state machine equipped with a stack containing data. Pushdown automata are equivalent to context-free grammars. Nevertheless for humans, pushdown automata are most times easier to understand.

1.1 Requirements

The goal of this project is to create a tool (in the following called simulator) which illustrates the behavior of nondeterministic as well as deterministic pushdown automata (PDAs in the following). In detail, it should fulfill the following requirements:

- *load a PDA*: Existing PDAs should be read from an input file which has a format similar to the common formal definition of PDAs.
- *create a PDA*: The user should be able to create a PDA via the graphical user interface (GUI for short), i.e., without knowing the exact input format.
- *modify a PDA*: The user should be able to modify existing PDAs via the GUI, again, without knowing the exact input format.
- *simulate a PDA on a specific input string*: There should be four buttons available to control the simulation. A forward button which always performs one step, i.e., applies exactly one transition. A backward button which goes one step back, making it possible to repeat that step. A fast forward button which completes the simulation without stopping and finally a fast backward button which goes back to the beginning of the current simulation.
- *change the acceptance mode of a PDA*: Since the simulator should support the two common definitions of acceptance (by final state as well as by empty stack) it should be possible to convert an existing PDA that accepts by empty stack into a PDA that accepts by final state and vice versa.

The program should come equipped with a well understandable and easy to use GUI. This implies that PDAs should be represented in a textual and graphical way. During the simulation the actual configuration including the state, the remaining input string, as well as the stack should be printed. Detailed information, like the previous configuration, the applied transition, and the resulting configuration should be printed in some special window. To make it easier for the user to follow the simulation, the current state in the graphical representation of the PDA should be highlighted. If during the simulation the situation occurs that more than one transition is applicable, the user should be prompted to decide which transition should be applied next. At the end of the simulation it should be indicated whether the input has been accepted or not. If the input has been accepted by the PDA all accepting sequences should be printed.

1.2 Motivation

The simulator and this associated thesis have been created as a bachelor thesis in the summer semester 2009. The simulator will mainly be used to show students that are not familiar with this topic how PDAs work in detail. So the focus of this project has been put on the understandability of the simulator rather than other (popular) issues like time- and memory-performance.

1.3 Structure

The rest of the thesis is organized as follows. In Chapter 2, PDAs are introduced. Beside some basic information, the algorithm used to convert a PDA that accepts by empty stack into a PDA that accepts by final state and vice versa is explained. In Chapter 3 we present some other tools similar to the one developed during this bachelor project. At the end we compare these programs with our simulator and discuss the main differences. Chapter 4 contains a brief overview of the implementation. That means we shortly explain all packages and classes generated during the implementation phase. Finally some interesting aspects of the simulator are discussed in more detail. The last part of the thesis, Chapter 5, is dedicated to the end-user. It describes how the simulator can be installed and used.

2 Pushdown Automata

2.1 Introduction

A PDA is a theoretical construct used to describe a set of strings. Since PDAs belong to the family of abstract machines, their functionality can be described as follows:

> The automaton is in a certain state and gets a string of input symbols. For every single input symbol it changes its state (or stays in the actual state) according to a specific set of rules. When the input has been consumed it is either accepted or not accepted.

What differs a PDA from other abstract machines like finite automata is its additional stack (or pushdown store) which is used to store and read information in a last in first out manner. This gives a PDA additional computational power. While a finite automaton can only represent regular languages, PDAs characterize the class of context-free languages.[1]

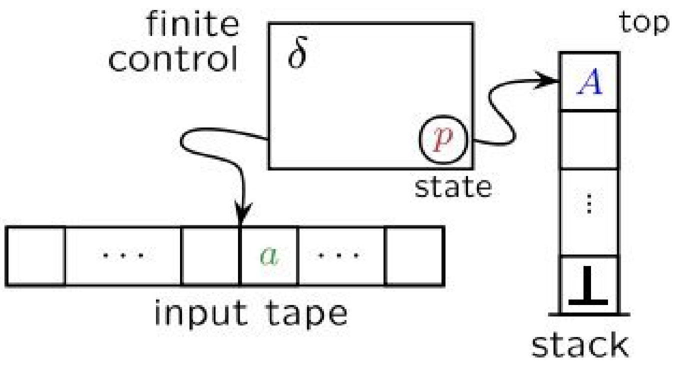


Figure 2.1: Illustration of a PDA

As illustrated in Figure 2.1 a PDA consists of a finite input tape which can be read in only one direction (normally from left to right) and a stack from which the top symbol can be popped of (removed and read) and arbitrary many symbols pushed back onto the stack. Beside that each PDA admits a finite control, the heart of a PDA, which decides based on the current state, the input symbol and the topmost symbol of the stack, in which state to change and what to push back onto the stack. Note that a PDA can also perform so called ϵ-transitions, where the state and stack can be manipulated without reading anything from the input.

[1]With an additional stack (two in total) one would get a PDA as powerful as a Turing machine [8], which is capable of representing all computable functions.

2.2 Formal Definition

In the following section we formally define PDAs. Since our simulator should support deterministic as well as nondeterministic PDAs, and deterministic PDAs are subsumed by nondeterministic PDAs, we only define the latter ones. So from now on we always have a nondeterministic PDA in mind if we write PDA. The presentation follows [5] and [3].
A (nondeterministic) *PDA* is a 7-tuple

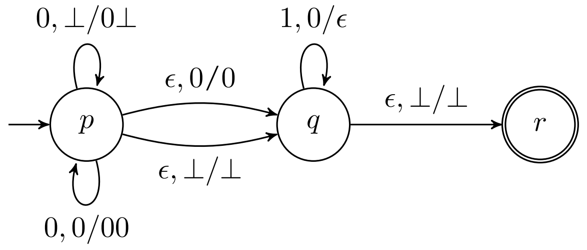
$$M = (Q, \Sigma, \Gamma, \delta, s, \perp, F)$$

where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $\delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$ is a finite set of transitions
- $s \in Q$ is the start state
- $\perp \in \Gamma$ is the initial stack symbol
- $F \subseteq Q$ is the set of final or accepting states

The set of transitions δ is sometimes also called the program of a PDA because it defines its behavior. A transition $((p, a, b), (q, b_1 \cdots b_n))$ means that if the considered PDA is in state p, reads symbol a from the input tape, and pops symbol b from the stack, than it goes to state q and pushes the symbols $b_1 \cdots b_n$ back onto the stack.

Example 2.1. Consider the following PDA M:



We have $Q = \{p, q, r\}$, $\Sigma = \{0, 1\}$, and $\Gamma = \{0, \perp\}$. The start state of the given PDA is p, the inital stack symbol is $\perp$, and $F = \{r\}$. The transitions are $((p, 0, \perp), (p, 0\perp))$, $((p, 0, 0), (p, 00))$, $((p, \epsilon, 0), (q, 0))$, $((p, \epsilon, \perp), (q, \perp))$, $((q, 1, 0), (q, \epsilon))$, and $((q, \epsilon, \perp), (r, \perp))$.

2.2.1 Configuration

Configurations are used to characterize the different states of a PDA $M = (Q, \Sigma, \Gamma, \delta, s, \perp, F)$. Therefore a configuration is a 3-tuple

$$(q, x, y) \subseteq Q \times \Sigma^* \times \Gamma^*$$

where the first argument q describes the current state, x the remaining part of the input, and the last argument y the content of the stack.

Example 2.2. Consider the PDA M of Example 2.1. If M is in the configuration $(p, 0011, \perp)$ then this means that M is in the state p, 0011 is written on the input tape where the first symbol (0) will be read next, and $\perp$ is the only symbol on the stack.

2.2.2 Acceptance

In case of a PDA we differ between two different modes of acceptance:

- acceptance by *final state*
- acceptance by *empty stack*

The first one requires the PDA to be in a defined final-state when the input is finished. The latter one requires that the PDA has an empty stack (no symbols at all) when the input is finished.
Formally let $M = (Q, \Sigma, \Gamma, \delta, s, \perp, F)$ be a PDA. Then M *accepts* a string $x \in \Sigma^*$ *by final state* if there exists a derivation

$$(s, x, \perp) \xrightarrow[M]{*} (q, \epsilon, y)$$

for some $q \in F$ and $y \in \Gamma^*$. In the right configuration, ϵ denotes the null string, which means that the entire input has been read, and y stands for some junk left on the stack. In contrast M *accepts* a string $x \in \Sigma^*$ *by empty stack* if

$$(s, x, \perp) \xrightarrow[M]{*} (q, \epsilon, \epsilon)$$

for some $q \in Q$. Note that the q in the right configuration denotes an arbitrary state. Furthermore, as described before, the ϵ in the second and third position of the right configuration indicates that the entire input has been read and the stack is empty.

Example 2.3. Consider again the PDA M of Example 2.1. To check if M accepts the string 0011 by final state we try to construct a derivation from the initial configuration $(p, 0011, \perp)$ to some final configuration (r, ϵ, y) with $y \in \Gamma^*$. In order to modify the initial configuration we can apply two transitions: $((p, 0, \perp), (p, 0\perp))$ or $((p, \epsilon, \perp), (q, \perp))$. A closer look to the PDA shows that we can consume the symbol 0 only in state p. So to ensure that the whole input is read we apply the first transition an end up with the configuration

$(p, 011, 0\bot)$. After that we apply the same transition again, yielding the configuration $(p, 11, 00\bot)$. The only possible transition that can be applied next is $((p, \epsilon, 0), (q, 0))$ which leads us to the configuration $(q, 11, 00\bot)$. In order to get rid of the 1s in our remaining string we use two times the transition $((q, 1, 0), (q, \epsilon))$. So we end up with the configuration $(q, \epsilon, \bot)$. Last but not least we apply the transition $((q, \epsilon, \bot), (r, \bot))$ which yields the configuration $(r, \epsilon, \bot)$. Since all symbols of the input string have been consumed and r is a final state we can conclude that M accepts 0011 by final state. The found accepting sequence looks as follows:

$$\begin{aligned}(p, 0011, \bot) &\xrightarrow[M]{} (p, 011, 0\bot) \xrightarrow[M]{} (p, 11, 00\bot) \xrightarrow[M]{} (q, 11, 00\bot)\\ &\xrightarrow[M]{} (q, 1, 0\bot) \quad \xrightarrow[M]{} (q, \epsilon, \bot) \quad\;\; \xrightarrow[M]{} (r, \epsilon, \bot)\end{aligned}$$

If we try to check if M accepts the string 0011 by empty stack we can immediately see that this will never happen because the stack symbol $\bot$ cannot be removed, i.e., there is no transition which pops the $\bot$ symbol from the stack and replaces it by ϵ.

2.2.3 Language

Let $M = (Q, \Sigma, \Gamma, \delta, s, \bot, F)$ be a PDA. The *language* $L(M)$ of M is defined as the the set of all strings that are accepted by M. So if M accepts by final state we have $L(M) = \{x \mid \exists q \in F, y \in \Gamma^*\colon (s, x, \bot) \xrightarrow[M]{*} (q, \epsilon, y)\}$ and if M accepts by empty stack then $L(M) = \{x \mid \exists q \in Q\colon (s, x, \bot) \xrightarrow[M]{*} (q, \epsilon, \epsilon)\}$.

Example 2.4. Concerning the PDA M of Example 2.1 we have $L(M) = \{0^n1^n \mid n \geq 0\}$ under the assumption that M accepts by final state (otherwise $L(M) = \varnothing$ as already indicated in Example 2.3). This can be seen as follows: The first transitions that can be used to consume some input symbols are the two looping transitions $((p, 0, \bot), (p, 0\bot))$ and $((p, 0, 0), (p, 00))$. Both of them consume a 0 from the input string and write an additional 0 on the stack. So in state p we write a 0 on the stack for every 0 we read. Since there are no other transitions that consume 0s it is clear that we have to stay in state p as long as our input contains a 0. As soon as there are no 0s left one has to apply either the transition $((p, \epsilon, 0), (q, 0))$ or $((p, \epsilon, \bot), (q, \bot))$. The latter one is needed in the case that the input is empty. In state q we can consume 1s as long as there are 0s on the stack via the transition $((q, 1, 0), (q, \epsilon))$. Because this is the only transition that consumes 1s it follows already that M only accepts strings of the form 0^*1^* because 0s can only be consumed in state p and 1s can only be consumed in state q. It remains to check whether an accepted input must contain equally many 0s and 1s. If the input contains more 0s than 1s we can never reach the final state r because there is at least one 0 above the stack symbol $\bot$. Hence the transition $((q, \epsilon, \bot), (r, \bot))$ cannot be applied. Otherwise, if the input contains more 1s than 0s we can reach the final state r. However there is at least one 1 that cannot be consumed. Therefore the input string is not accepted too.

2.3 Changing Acceptance

As mentioned at the beginning, the simulator has to provide the possibility to transform a PDA that accepts by final state into a PDA that accepts by empty stack and vice versa. In the following we describe the basic algorithm used to perform those transformations. After that we show that the basic algorithm is not optimal, i.e., that the resulting PDAs sometimes consists of unnecessary states and transitions. Finally we discuss some improvements and present a new, improved version of the algorithm.

2.3.1 The Standard Algorithm

The standard algorithm to convert a PDA $M = (Q, \Sigma, \Gamma, \delta, s, \perp, F)$ that accepts by empty stack or by final state into a PDA M' for which acceptance by empty stack and final state coincide works as follows: Let u and t be two new states not in Q and let $\perp\!\!\!\perp$ be a new stack symbol not in Γ. Let

$$G = \begin{cases} Q & \text{if } M \text{ accepts by empty stack} \\ F & \text{if } M \text{ accepts by final state} \end{cases}$$

and

$$\Delta = \begin{cases} \perp\!\!\!\perp & \text{if } M \text{ accepts by empty stack} \\ \Gamma \cup \perp\!\!\!\perp & \text{if } M \text{ accepts by final state} \end{cases}$$

We define M' as

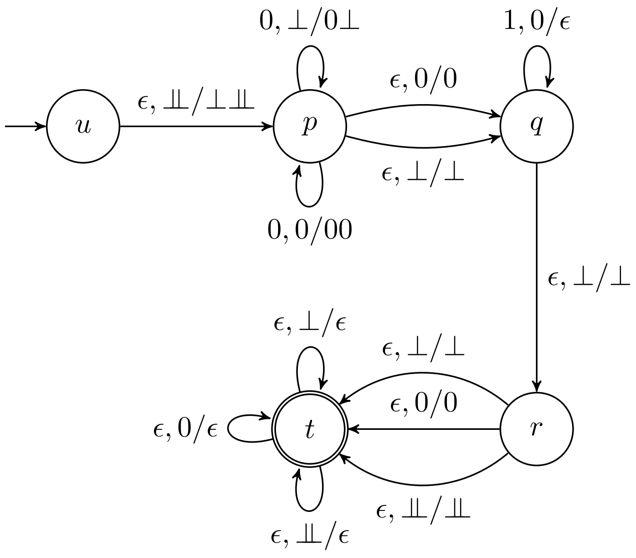
$$M' = (Q \cup \{u, t\}, \Sigma, \Gamma \cup \{\perp\!\!\!\perp\}, \delta', u, \perp\!\!\!\perp, \{t\})$$

where δ' contains all the transitions of δ, as well as the transitions

$$((u, \epsilon, \perp\!\!\!\perp), (s, \perp\perp\!\!\!\perp)) \tag{2.1}$$
$$((q, \epsilon, a), (t, a)) \qquad q \in G, a \in \Delta \tag{2.2}$$
$$((t, \epsilon, a), (t, \epsilon)) \qquad a \in \Gamma \cup \{\perp\!\!\!\perp\} \tag{2.3}$$

Due to the construction the new automaton M' has a new start state u, a new initial stack symbol $\perp\!\!\!\perp$, and a new single final state t. In the first step, according to transition (2.1), M' pushes the old initial stack symbol $\perp$ on top of $\perp\!\!\!\perp$ and enters the old start state s. After that M' simulates M since it contains all the transitions of M. At some point it might enter state t according to (2.2). Once it enters state t, it can dump everything off its stack using transitions (2.3). Note that M' can empty its stack only in state t because it cannot pop $\perp\!\!\!\perp$ from the stack in any other state. Hence it is quite easy to see that M' accepts the same language as M and that acceptance by empty stack and by final state coincides for M' (a detailed proof can be found in [5]).

Example 2.5. If we apply the above algorithm to the PDA of Example 2.1 we obtain the automaton

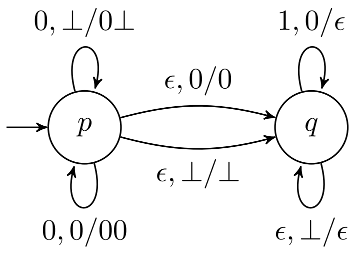


where acceptance by empty stack and by final state coincides.

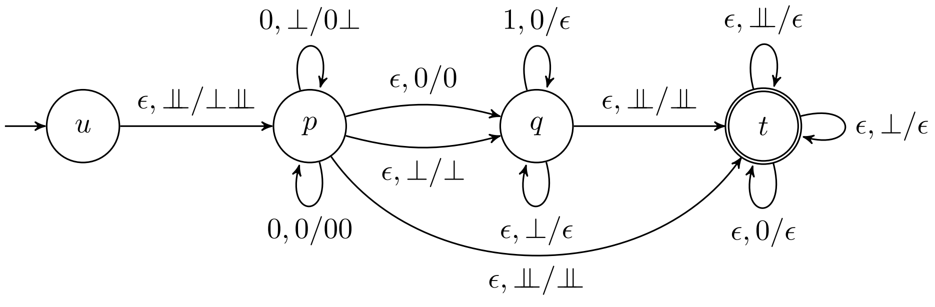
2.3.2 Improvements

It is quite obvious that the previous algorithm is not optimal in the sense that it sometimes introduces unnecessary states and transitions.

Example 2.6. Consider the following PDA M



which accepts the set $\{0^n1^n \mid n \geq 0\}$ by empty stack and assume that we want to convert it into a PDA M' that accepts the same language as M by final state. Applying the previous algorithm yields the following PDA M':



It is easy to see that the transition $((p, \epsilon, ⫫), (t, ⫫))$ can only be applied if all other symbols of the stack have been removed. Since such a situation can only occur in state q it follows that the transition $((p,\epsilon,\perp\!\!\!\perp),(t,\perp\!\!\!\perp))$ is superfluous. Furthermore since we wanted to convert M into a PDA M' that accepts the same language as M by final state, the transitions $((t,\epsilon,a),(t,\epsilon))$ with $a \in \{0,\perp,\perp\!\!\!\perp\}$ are unnecessary too, because if we are in final state t it does not matter what is written on the stack.

In total the following improvements have been incorporated:

1. At first the algorithm checks if it makes sense to convert the given automaton $M = (Q,\Sigma,\Gamma,\delta,s,\perp,F)$. I.e., if M should be converted into a PDA M' that accepts by final state it is checked whether M contains at least one transition of the form $((p,a,b),(q,\epsilon))$ with $p,q \in Q$, $a \in \Sigma \cup \{\epsilon\}$, and $b \in \Gamma$. If that is not the case we know that M accepts by final state or that $L(M) = \varnothing$. So in both cases we can take $M' = M$. If M should be converted into a PDA M' that accepts by empty stack it is checked whether M contains at least one final state. If $F = \varnothing$ then we can conclude that M accepts by empty stack or that $L(M) = \varnothing$. Hence it suffice to take $M' = M$.

2. Assume that we want to convert a PDA $M = (Q,\Sigma,\Gamma,\delta,s,\perp,F)$ that accepts by empty stack to a PDA M' that accepts the same language as M by final state. Then we do not necessarily need a transition $((q,\epsilon,\perp\!\!\!\perp),(t,\perp\!\!\!\perp))$ from every state q to the new final state t. It is enough to consider only those states which have an incoming transition of the form $((p,a,b),(q,\epsilon))$ with $p \in Q$, $a \in \Sigma \cup \{\epsilon\}$, and $b \in \Gamma$.

3. Assume again that we want to convert a PDA M that accepts by empty stack to a PDA M' that accepts by final state. Then it is not necessary to empty the stack in the new final state.

By incorporating this improvements into the standard procedure, the following new algorithm can be derived: Let $M = (Q,\Sigma,\Gamma,\delta,s,\perp,F)$ be a PDA which should be converted into a PDA M' that accepts by empty stack if M accepts by final state and vice versa. Let

$$G = \begin{cases} P & \text{if } M \text{ accepts by empty stack} \\ F & \text{if } M \text{ accepts by final state} \end{cases}$$

where $P = \{q \mid \exists p \in Q, a \in \Sigma \cup \{\epsilon\}, b \in \Gamma\colon ((p,a,b),(q,\epsilon)) \in \delta\}$ and

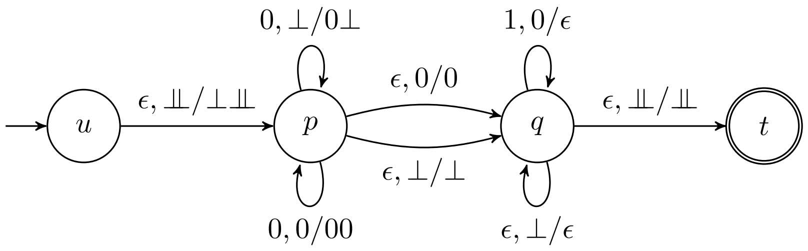
$$\Delta = \begin{cases} \perp\!\!\!\perp & \text{if } M \text{ accepts by empty stack} \\ \Gamma \cup \perp\!\!\!\perp & \text{if } M \text{ accepts by final state} \end{cases}$$

We have $M' = M$ if either M accepts by empty stack and $P = \varnothing$ or M accepts by final state and $F = \varnothing$. Otherwise $M' = (Q \cup \{u,t\}, \Sigma, \Gamma \cup \{\perp\!\!\!\perp\}, \delta', u, \perp\!\!\!\perp, \{t\})$ where

$$\delta' = \begin{cases} \delta'' & \text{if } M \text{ accepts by empty stack} \\ \delta'' \cup \delta''' & \text{if } M \text{ accepts by final state} \end{cases}$$

with $\delta'' = \{((u,\epsilon,\perp\!\!\!\perp),(s,\perp\perp\!\!\!\perp)),((q,\epsilon,a),(t,a)) \mid q \in G, a \in \Delta\}$ as well as $\delta''' = \{((t,\epsilon,a),(t,\epsilon)) \mid a \in \Gamma \cup \{\perp\!\!\!\perp\}\}$.

Example 2.7. If we apply the new algorithm to the PDA of Example 2.6 in order to obtain a PDA that accepts the same language as M by final state we end up with the PDA



3 Related work

3.1 Similar programs

There exist several free ware tools (accessible via the Internet) which can be used to illustrate the behavior of PDAs. In the following section we mention three of them, which seems to be the most suitable ones.

Tool Automaton Simulator

The Automaton Simulator is part of the so called Science Of Computing Suit (SOCS), a collection of programs developed for an introductory survey course in computer science. This simulator allows the user to create and simulate finite automata, Turing machines and deterministic PDAs. It has a very simple graphical user interface which makes it possible to draw an automaton and run it on some input. For further information see [11].

Tool npda

The program npda [12] was developed by Brain Shelbourne, professor at the Wittenberg University in Ohio, USA. It is a graphical simulator that runs under MS-DOS and supports deterministic as well as nondeterministic PDAs. Beside the simulation of a PDA, the program allows the user to load a PDA form a file. During the simulation the program asks the user which transition to take if more than one is applicable.

Tool Another Non-Deterministic Push-Down Automaton

This program is very similar to the one developed during this bachelor project. It is a small and simple Java program that allows the user to run a PDA on different inputs in parallel. All necessary information have to be specified in an input file. PDAs are represented in a graphical way which makes it easier for the user to follow the simulation steps. Further information can be obtained from [10].

3.2 Comparison

In the following section we mention some differences between our simulator and the tools above. As indicated at the beginning, the aim of this project is to create a program that can be used in lectures like Formal Language and Automata Theory or Discrete Mathematics to help students to understand the behavior of a PDA. Thereby, one of the most important requirements is that

the functionalities dedicated to the simulation of a PDA are easy to understand and to use.
Our simulator has an intuitive graphical interface. There are just a few buttons with clear functionality. In detail, to control the simulation the user has the possibility to perform a single forward step or a single backward step. In addition it is also possible to go back to the beginning of the simulation or to switch to an automatic mode which completes the simulation autonomously. To make it easier for the user to follow the simulation detailed textual output is provided. In addition, the actual state of the PDA is highlighted.
A big disadvantage of all mentioned tools is that they provided only insufficient information during the simulation, i.e., they only print the actual configuration. So the user has to guess which transition has been applied to the previous configuration. Furthermore it is not possible to control which acceptance mode should be taken. The tools Automaton Simulator as well as npda support only PDAs that accept by final state, whereas the tool Another Non-Deterministic Push-Down Automaton always checks both modes. This might explain also the fact that none of the tools offer the possibility to transform a PDA that accepts by final state into a PDA that accepts by empty stack or vice versa.
A big disadvantage of the tool Automaton Simulator is that it does not support nondeterministic PDAs. In addition to that it neither allows backward steps nor something like an automatic-mode which completes the simulation without any user interaction. It is also not very intuitive to create a PDA with the provided drag-and-drop environment.
As mentioned at the beginning, the tool npda is written in DOS. So one cannot actually talk about a GUI. Everything is presented in a textual way only. Similar as the tool Automaton Simulator it does not allow backward steps and offers no automatic-mode.
The tool Another Non-Deterministic Push-Down Automaton offers a lot more options than the other two programs. For instance it is possible to perform simple backward steps. With simple we mean here that backward steps are only allowed in the current branch of the computation tree. In case that more than one transition is applicable, the tool automatically chooses the transition that is applied next. So the user cannot really control the simulation of a nondeterministic PDA. Beside that, the tool does not print the accepting sequences.

4 Implementation

The program is written in the programming language Java [13]. The graphical user interface has been created using swing and JGraph [7]. In total the whole simulator consists of about 3500 lines of code.[1] As development environment Eclipse [2] and NetBeans [9] have been used. The used programming guidelines and patterns have been taken from [6] and [14].

4.1 Packages

The code can be divided into four packages:

- def: Contains all classes used to define a PDA.
- main: The package main defines the core of the program. That means it contains the main algorithms of the simulator.
- gui: Provides several classes used to define the GUI of the program.
- helper: This package provides some basic functions used in the other packages.

In the following we explain the most important classes and functions of each package.

4.1.1 Package def

As mentioned at the beginning, this package contains all basic classes used to define a PDA. The class that defines a PDA itself is contained in the package main because it contains all methods that are needed to implement the simulation as well as the conversion algorithm.

Class PDrule

The class PDrule is used to model rules. A rule is stored as a quintuple where the first attribute defines the current state, the second one the symbol read from the input tape, the third the symbol read from the stack, the fourth one the successor state, and the last one the characters that have to be written on the stack. Once a PDrule is created, the attributes cannot be changed anymore. Therefore the only available methods are so called getter-functions to access the attributes.

[1]The lines of code have been counted with the program LOC. LOC counts only *pure* lines of code. So no comments and no lines with less than three characters have been taken into consideration.

Class PDstack

The class PDstack represents the stack used by a PDA. In order to represent the stack in a nice way it is necessary to access all information written on the stack at once without destroying the stack. Since the standard stack implementation of Java does not allow to access all the data written on the stack at once, we implemented our own stack. Nevertheless, apart from the printing functions the used stack behaves exactly like a normal stack where modifications are conducted with the well-known methods push and pop.

Class PDstatus

The class PDstatus is used to represent a configuration. Its attributes are the current state, the actual stack, and last but not least the remaining input. From a method-side of view it includes some basic functions to get the attributes as strings or characters.

Class RunnerList

Another class contained in this package is the so called RunnerList. This class is nothing else than a array-list with some additional methods to add and sort the objects contained in the list. The name RunnerList has been given to this class because the objects stored in the array-list are so called *runners*. Detailed information what runners are and for what they are used are provided in the description of the package main.

4.1.2 Package main

This is the package where the *action* takes place. So, all the methods that are used to implement the execution or conversion of a PDA are located in this package.

Class PDAmain

This class defines a PDA. So it contains on the one hand all the information used to describe a PDA and on the other hand the functionalities needed to create, modify, simulate and convert a PDA. This implies that whenever some action happens it definitely passes this class.

Class Matcher

This class is responsible to collect and select the transitions that can be applied to the given configuration. For each transition that is applicable it creates a runner (see below) and puts it at the beginning of the execution-list (the execution-list defines which transition should be applied next). The method go takes the first runner of the execution-list and executes it. In addition it copies the taken runner to a so called runner-list. This list is used to store the simulation tree where every branch corresponds to one possible derivation. Note that the runner-list is used to implement backward steps.

Class Runner

Objects of this class have the task to apply a given transition to some configuration. Therefore a runner is created via the class Matcher. Each runner consists of a method run which applies a given transition to a some PDstatus object.

Lexer PDA

This file contains all definitions needed for the lexical analysation of a given input file. Note that this file does not contain any Java classes. The corresponding Java class that is used for the lexical analysation of an input is automatically generated via the program JFlex [4].

Parser PDA

This file defines the grammar of the input format. Similar as in the case of the lexer, the Java class used to parse an input file is generated automatically via the program Cup [1].

4.1.3 Package gui

This package contains all classes that are part of the GUI. The GUI has been created via the tool NetBeans. Therefore some parts of the code in this package have been generated automatically.

Class PdaFrame

The class PdaFrame defines the core of the GUI. It is the only class in the whole program that has a main method. So if the file pda.jar is started this method is executed. All the user interaction is handled in this class. Therefore it contains all methods that are used to control button-actions and to fill textpanels and windows with information. Furthermore it contains a method draw that uses JGraph [7] to visualize PDAs.

Class ConvertFrame

This class contains the dialog that opens if the menu button Convert is clicked. It calls the parse method from the class PDAmain to load the given input file. Afterwards it collects all information needed to convert the given PDA into a PDA that accepts by final state or by empty stack.

Class CreateFrame

This class contains the frame that opens if the menu button Create or Modify is clicked. If accessed through the menu button Modify it additionally calls the parse method to load the given PDA.

Class NonDetDialog

This class contains the dialog that opens if during the simulation more than one transition can be applied. As soon as the user has chosen some transition the matcher is addressed to actualize the execution-list.

Class RuleDialog

This class is used by the class CreateFrame to modify and specify the transitions of a PDA.

4.1.4 Package helper

This package provides some basic functions used in the other packages.

Helper

This class contains a lot of useful methods, e.g., dlm2LL which converts a list of type DefaultListModel to a list of type LinkedList.

PDAexception

This class is an extension of the standard Java class Exception. It defines some special exceptions used in the implementation.

4.2 Backstep

An interesting part of the code is the implementation of the backward step algorithm. As mentioned above there are two array-lists called execution-list and runner-list. Runners in the execution-list get removed as soon as they have been executed. If a runner is removed from the execution-list it gets added to the runner-list. Now if the backward step button is clicked the following steps are performed:

1. The runner-list gets ordered so that the last performed runners are at the end of the list.
2. At next the position of the predecessor of the actual runner is computed.
3. After that all runners of the runner-list, starting from the computed position up to the end of the list, are copied into a temporary array-list.
4. Finally, the execution-list is replaced by the temporary array-list.

5 User Guide

5.1 Installation

As this program is available as an executable .jar file, only the Java Runtime Environment version 1.6 [13] has to be installed. It was tested under Windows XP, Ubuntu 9, and Debian 4 but should run on every operating system where Java Runtime Environment 1.6 is installed. To run the program just execute the file pda.jar (proper file association provided), or execute the command

```
java -jar pda.jar
```

in the directory where the file pda.jar is located. After starting the program the main window appears (see Figure 5.1). The main window can be divided

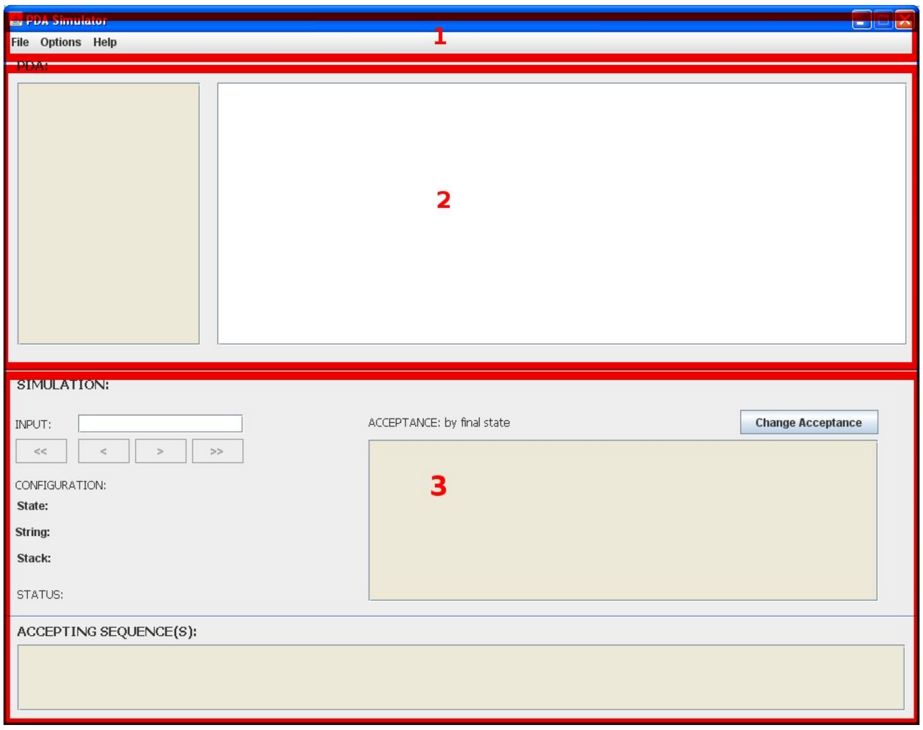


Figure 5.1: The PDA simulator

in three sections: The menu bar (1), the input section (2), and the simulation section (3).
The menu bar consists of the items File, Options, and Help. The entry File contains the most important functionalities. It allows the user to load a PDA

in order to execute it on some input, to create or modify PDAs, and to change the acceptance mode of a PDA.
The input section is used to provide detailed information about the loaded PDA. In the top left corner of the window a text area can be found which represents the loaded PDA in textual form. In order to make it easier for the user to follow the simulation the given PDA is also represented in a graphical form. The graphical illustration of the PDA is shown in the right part of the input section.
As the name already indicates, the simulation section is used to control the simulation and provide the user with detailed information about the simulation. In the top part of this area the control unit is located. It consists of the buttons <<, <, >, >>, and Change Acceptance. The rest of this section is used to print detailed information about the simulation like the applied transitions, the current configuration, the accepting sequences, etc.

5.2 The input format

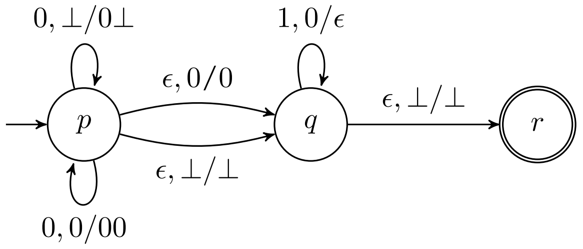
In order to run a PDA on some string, the corresponding PDA has to be specified at first according to the used input format. The developed input format is quite similar to the formal definition of a PDA. Therefore each input file has to admit the following parts (separated by semicolons):

- At first the states of the PDA have to be defined (separated by a comma). Note that each state has to be represented by a single character. This restriction has been made in order to ensure that the graphical presentation of the PDA does not grow to fast.
- Afterwards the input alphabet of the PDA is specified. Similar symbols are separated by commas. Furthermore each symbol has to be represented by a single character.
- After the input alphabet the stack alphabet of the PDA has to be specified. The specification of the stack alphabet follows the same guidelines as the specification of the input alphabet.
- Next the transitions of the PDA have to be defined. Transitions are represented as quintuples where the first argument defines the current state, the second argument the input symbol that is read, the third argument the topmost stack symbol, the fourth one the successor state, and the last argument the symbols that are written on the stack. Each transition has to be specified between parentheses. Arguments as well as transitions are separated by a comma.
- At next the start state has to be defined.
- After the specification of the start state the initial stack symbol has to be specified.

- Next the final states of the PDA have to be defined. Per definition it is not necessary to define final states. In that case the user can leaf this part empty.
- Last but not least the user has to define the symbol that is used as representative for the empty string and empty stack.

Whenever a PDA is loaded, the input file gets checked for errors. Redundancies in the input file, e.g., if a state is defined twice, are corrected automatically. Fatal errors like missing parts lead to an error message.

Example 5.1. Consider the following PDA:

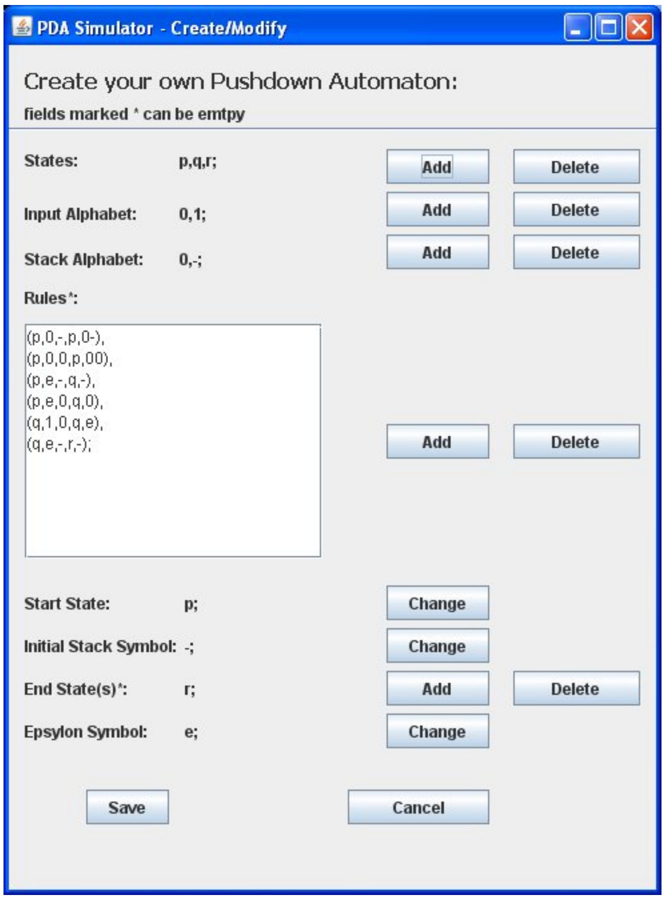


The correct input file for this PDA, where ϵ is replaced by `e` and $\perp$ by `-`, looks as follows:

```
p,q,r;
0,1;
0,-;
(p,0,-,p,0-),
(p,0,0,p,00),
(p,e,-,q,-),
(p,e,0,q,0),
(q,1,0,q,e),
(q,e,-,r,-);
p;
-;
r;
e;
```

5.3 Creating or Modifying a PDA

Via the menu entries Create and Modify the user can create and modify PDAs. Independently from the clicked button the window shown in Figure 5.2 is opened. If the menu entry Modify has been chosen the user is asked first which PDA should be loaded.
In order to create a PDA the states, the input alphabet, the stack alphabet, the transitions, the start state, the initial stack symbol, the final states, and finally the symbol used to represent the empty string and empty stack have to be specified. This can be done using the buttons Add, Delete, and Change. As the


PDA Simulator - Create/Modify

Create your own Pushdown Automaton:

fields marked * can be emtpy

States: p,q,r;

Input Alphabet: 0,1;

Stack Alphabet: 0,-;

Rules*:

```
(p,0,-,p,0-),
(p,0,0,p,00),
(p,e,-,q,-),
(p,e,0,q,0),
(q,1,0,q,e),
(q,e,-,r,-);
```

Start State: p;

Initial Stack Symbol: -;

End State(s)*: r;

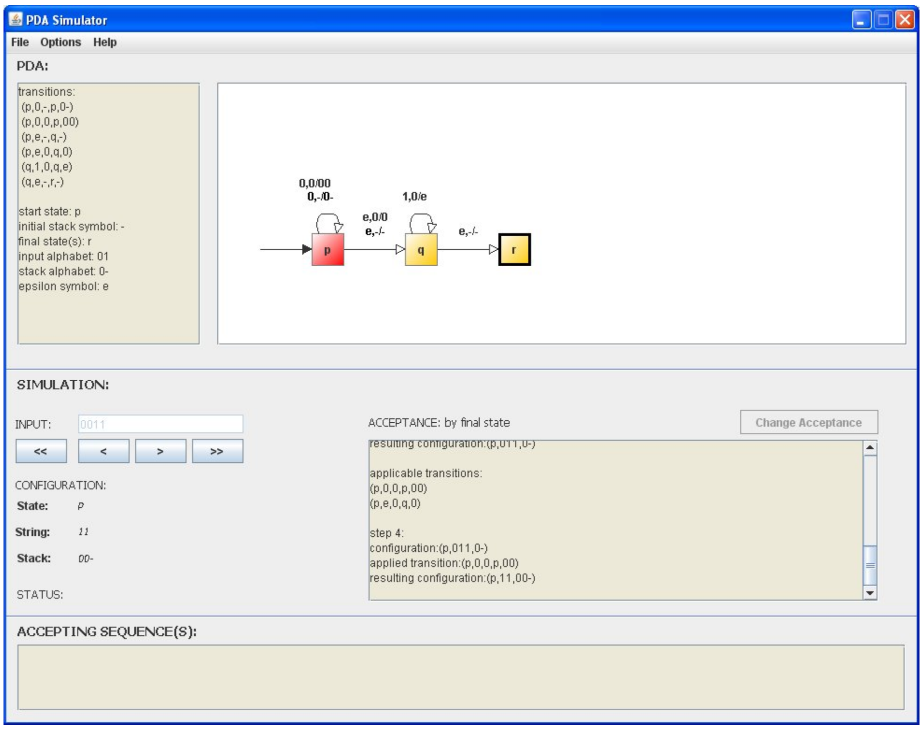
Epsylon Symbol: e;

Figure 5.2: The create and modify window

name already indicates, the Add button is used to add a state (symbol, transition, final state) to the set of states (input or stack alphabet, set of transitions, set of final states respectively). Via the Delete button the last added state (symbol, transition, final state) can be removed from the set of states (input or stack alphabet, set of transitions, set of final states). Finally with the Change buttons the user can specify or change the start state and the initial stack symbol. As soon as all components have been specified the created PDA can be stored by pressing the button Save.
The modification of a PDA works similar as the creation of a new PDA. The only difference is that all fields are prefiled with the data of the chosen PDA.

5.4 Loading and Simulating a PDA

In order to simulate a PDA on some string, at first the corresponding PDA has to be loaded. This can be done by choosing the menu entry Open from the menu item File. As soon as some PDA has been selected the input file is checked for errors. If everything is fine the main window is filled with the information of the chosen PDA (see the top part of Figure 5.3).



```
transitions:
 (p,0,-,p,0-)
 (p,0,0,p,00)
 (p,e,-,q,-)
 (p,e,0,q,0)
 (q,1,0,q,e)
 (q,e,-,r,-)

start state: p
initial stack symbol: -
final state(s): r
input alphabet: 01
stack alphabet: 0-
epsilon symbol: e
```

```
INPUT: 0011

CONFIGURATION:
State:   p
String:  11
Stack:   00-

STATUS:
```

```
ACCEPTANCE: by final state

resulting configuration:(p,011,0-)

applicable transitions:
(p,0,0,p,00)
(p,e,0,q,0)

step 4:
configuration:(p,011,0-)
applied transition:(p,0,0,p,00)
resulting configuration:(p,11,00-)
```

Figure 5.3: The PDA simulator while executing an input

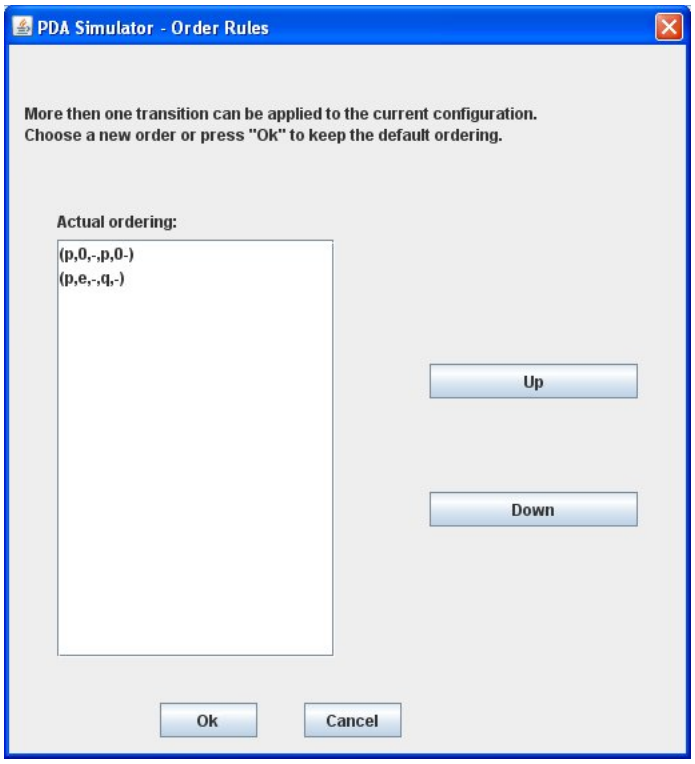
At next the string as well as the acceptance mode of the loaded PDA have to be specified. Per default it is assumed that the PDA accepts by final state. If that is not correct, the acceptance mode can be changed by pressing the button Change Acceptance. The label left of the button shows the selected acceptance mode. The input string has to be specified in the corresponding text field on the left side of the window. If nothing is entered, the simulator will check if the empty string is accepted by the given PDA. After that the simulation can be started by pressing the button $>$ or $>>$. If the user wants to perform a step wise simulation the first button has to be pressed. So by clicking on the button $>$ one transition is applied to the current configuration. If the user presses the button $>>$ the program automatically completes the simulation. In addition to the buttons $>$ and $>>$, which are used to perform forward steps, the two buttons $<$ and $<<$ exist. The first one is used to perform one backward step where the second is used to go back to the beginning of the simulation. If the button $<<$ is pressed twice it will even delete the input. Note that independently from the chosen mode (stepwise computation or automatic computation), all possible derivations are enumerated. In order to make it easier for the user to follow the simulation several information are printed:

- In the left part of the simulation area, the current configuration is shown.
- In the right part of the simulation area, detailed information about each

step, like the previous configuration, the applied transitions, and the resulting (current) configuration, are printed.

- In the graphical representation of the PDA, the actual state is highlighted.
- As soon as an accepting sequence has been found, it is printed in the textfield located at the bottom of the simulation area.
- After the simulation the result, that means, whether the string is accepted or not, is printed.

During a simulation it can happen that more than one transition is applicable. In such a situation the user is asked which transition should be applied first, i.e., the user has to order the applicable transitions. This can be done via the window shown in Figure 5.4. If the user does not want to choose the ordering


PDA Simulator - Order Rules

More then one transition can be applied to the current configuration.
Choose a new order or press "Ok" to keep the default ordering.

Actual ordering:

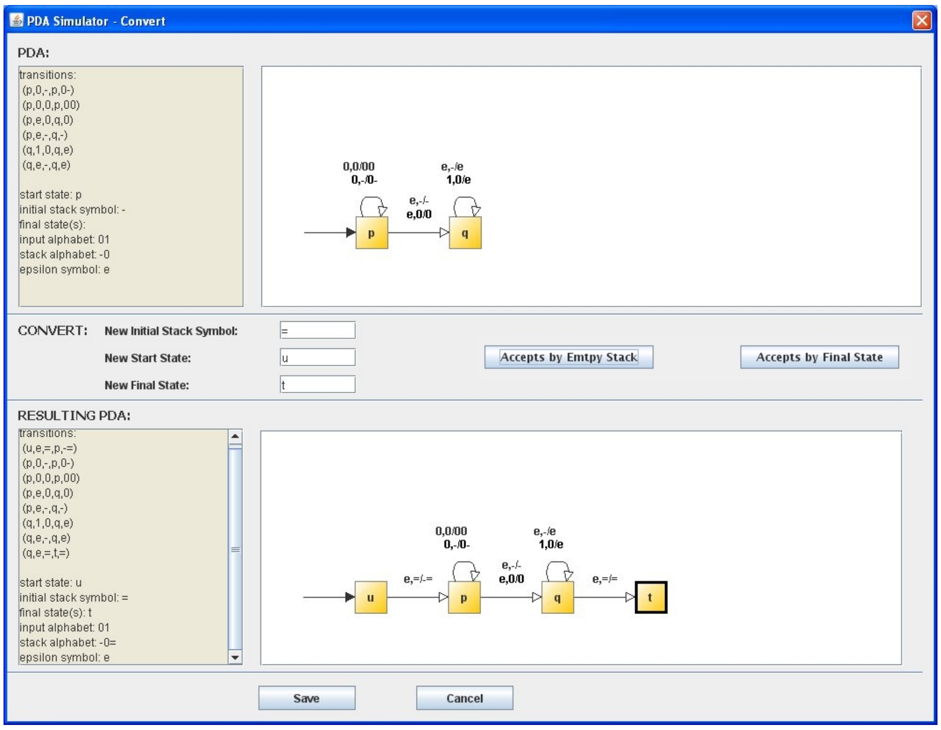
(p,0,-,p,0-)
(p,e,-,q,-)

Figure 5.4: Choosing a transition

in which transitions should be applied if more than one transition is applicable, it can turn of this feature by clicking on the menu entry "select transitions automatically" located in the menu item "Options".

5.5 Converting a PDA

In order to change the acceptance mode of a PDA one has to click on the menu entry Convert. As soon as some PDA has been selected the input file is checked for errors. If everything is fine a new window, filled with the information of the chosen PDA, appears (see Figure 5.5).


PDA Simulator - Convert

PDA:

```
transitions:
 (p,0,-,p,0-)
 (p,0,0,p,00)
 (p,e,0,q,0)
 (p,e,-,q,-)
 (q,1,0,q,e)
 (q,e,-,q,e)

start state: p
initial stack symbol: -
final state(s):
input alphabet: 01
stack alphabet: -0
epsilon symbol: e
```

CONVERT:

New Initial Stack Symbol: =
New Start State: u
New Final State: t

RESULTING PDA:

```
transitions:
 (u,e,=,p,-=)
 (p,0,-,p,0-)
 (p,0,0,p,00)
 (p,e,0,q,0)
 (p,e,-,q,-)
 (q,1,0,q,e)
 (q,e,-,q,e)
 (q,e,=,t,=)

start state: u
initial stack symbol: =
final state(s): t
input alphabet: 01
stack alphabet: -0=
epsilon symbol: e
```

Figure 5.5: Converting a PDA

At next the user has to enter a new initial stack symbol, a new start state, and a new final state. Note that these information are needed for both conversions (see Section 2.3). So it is irrelevant whether the given PDA should be converted into a PDA that accepts by empty stack or by final state. After that the user has to choose whether the given PDA accepts by empty stack or by final state, by clicking on the button Accepts by Empty Stack or Accepts by Final State. If the first button is selected, the given PDA is converted into a PDA that accepts by final state. Otherwise a PDA that accepts by empty stack is constructed. As soon as one of those buttons is pressed the new PDA is displayed in the bottom of the window. Finally, via the Save button it is possible to store the created PDA.

Bibliography

[1] Cup parser generator for java. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`. last visited on September 8, 2009.

[2] Eclipse ide. available at `http://www.eclipse.org/`. last visited on September 21, 2009.

[3] John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages and Computation*. Pearson, 3rd edition, 2006.

[4] Jflex - the fast scanner generator for java. available at `http://jflex.de/`. last visited on September 8, 2009.

[5] Dexter Kozen. *Automata and Computability*. Springer, 1997.

[6] Robert Liguori and Patricia Liguori. *Java - kurz und gut*. O'Reilly, 2006.

[7] JGraph Ltd. Jgraph, java graph visualization. available at `http://www.jgraph.com/`. last visited on August 28, 2009.

[8] Georg Moser and Arne Duer. *Diskrete Mathematik 2*. 2008.

[9] Netbeans ide. available at `http://www.netbeans.org/`. last visited on September 21, 2009.

[10] Another non-deterministic push-down automaton. available at `http://www.cs.binghamton.edu/~software/pda/pdadoc.html`. last visited on September 8, 2009.

[11] Automata simulator. available at `http://ozark.hendrix.edu/~burch/proj/autosim/`. last visited on September 8, 2009.

[12] npda. available at `http://www4.wittenberg.edu/academics/mathcomp/bjsdir/software.shtml`. last visited on September 8, 2009.

[13] Sun. Java programming language. available at `http://www.java.com/`. last visited on August 28, 2009.

[14] Christian Ullenboom. *Java ist auch eine Insel*. Galileo Press, 2008.