Umeå University
Institution för Datavetenskap

DV2: Algorithms and Problemsolving 7.5 p
DV169VT16

OU5 Automata

Submitted 2016-03-28
Author: Lorenz Gerber (dv15lgr@cs.umu.se lozger03@student.umu.se)
Instructor: Lena Kallin Westin / Erik Moström / Jonathan Westin

Contents

1 Introduction

The subject of this assignment was to design and implement *Push Down Automata* (PDA) as a general datatype and apply it to simulate a specific PDA that can process input according to *reverse polish notation* (RPN) rules.

The formal defintion of a PDA is a 6-tuple consisting of the *set of States*, the *input alphabet*, the *stack alphabet*, *transition functions*, a *start state* and a set of *accepted states* [4, p. 113]. Push down automata can either be deterministic or non-deterministic. The most prominent feature that distinguishes push-down automata from other finite automata is the stack which allows to store information and process it later.

Reverse Polish Notation or RPN is a way of writing aritmethic expressions. Instead of the common 'in-fix' notation where an operator is written between the two operands, RPN uses 'post-fix' notation with the operator after the operands. This way of indicating arithmetic expressions makes paranthesis to modify the precedence of operators obsolete. RPN has been used readily in financial and scientific pocket calculators [3].

1.1 Push Down Automata Implementation

The lab assignment proposed to represent the PDA either as a table or as a graph. It was further communicated that the implementation shall be *finite* and *deterministic*. *Finite* defines that independent of the input, the PDA will always terminate. The *deterministic* property defines that there can be in each *machine state* maximum one viable *transition* to be taken.

Here it was decided to implement the PDA as a table. Initially, this was not based on any research but more a 'stomach feeling'. To keep the implementation as generic as possible it was further decided to separate checking the expression from calculating it.

2 Program Structure

The main program is used to create and configure the PDA. Hence the whole RPN logic is defined in the main program by creating and assigning states and transitions. Then the PDA is executed. On success, the actual RPN calculation function is called.

2.1 Datatypes

pda - Push Down Automaton

A generic implementation of a push down automaton according to Sipser [4, pp.112-125]. The datatype *pda* is constructed from a struct. It contains a table (from course datatypes [1], constructed from dynamic list) with *states*, a stack (from course datatypes [1], *stack_1cell*). The *states* table contains the transitions. The PDA contains further the variables *currentState* (pointer to the currently active state), *possibleTransition* (pointer to the next possible transition), *input* (pointer to the current position in the input string), *inputLeft* (int for how many input chars left to process), *bailout* (flag that is set stop processing) and *succeed* (marker that becomes true when the input was verified and the automaton is in an accepted state).

The functions in the interface that the user accesses are *create* (returns a new), *pda_addState* (to configure the pda with a new state) and *pda_execute* (to run the PDA). The functions *pda_setStartState*, *pda_getPossibleTransition* and *pda_doTransition* are called from within *pda_execute*.

Alphabets

Generally speaking, the *input* alphabet includes *numbers*, *operators*, *blanks* and *EOF*. The *Stack alphabet* contains single digit *numbers* and the *dollar symbol* to mark the first position in the stack.

The alphabets were implemented as functions. There are two different types of such alphabet functions: The first kind was used to check input and stack for matching letters. In case of a match, those functions return *true*, otherwise *false*. The other kind of functions were used to reproduce the letter to be pushed on the stack. They return the *ASCII* code of the letter to be pushed. This also includes a function that reads the current input letter and returns it.

By separating *recognition* of the input string from *calculating* the expression, it was achieved that the pda could be kept to operate on single *chars* instead of *strings* or multi digit *integers*.

States

The representation of *states* and *transitions* for a table based PDA can be done in various ways and the distinction between *state* and *transition* is less clear than in a graph based model. Here two different ways were considered: States constructed only as a container for transitions, without any reference to the alphabet. This requires a more complex transition datatype. Alternatively, *states* implemented according to the *example 2.14* in Sipser [4, p. 114], represented by triple nested table (or an aggregated array) whith multi-columns for each letter in the *input alphabet* and subgroups as the individual columns for each letter of the *stack alphabet*. Implementing a representation for this model could be done with a nested table, an array, where the logic for accessing the different levels is integrated in the code or by a tree structure. Such a representation has the advantage that it is directly visible whether a transition for a certain state is defined or not as it has a unique location in the data structure. When choosing a representation with states as mere containers for transitions, a control mechanism to prevent assignment of duplicate transitions is needed.

Here, it was chosen to implement states as containers for transitions. A state has an numeric *id* and a directed list with potential transitions to proceed along. The state itself is a *struct*. The interface of the *state* is in the current version very sparse. It contains just the functions *state_create* (returns a new state) and *state_addTransition* (to add transitions to the state).

Transition

A transition needs to ‘know’ whether it matches the current state, it needs to define how to modify the current state and a reference to the new/next state. In the current implementation, the transition is a *struct* with function pointers for the *alphabet* functions. Further, the transition contains an *int* of the destination state, *destInt* and a char *description*. Note that in the current implementation, the transition does not need to know its source state, hence, it could be reused as long as the destination state matches. However, after implementing the datatypes, it was found that reusing transitions causes problems with the memory free functions: It happend that a transition was attempted to be removed multiple times causing memory errors. Hence, in the current version, for simplicity, instead of reusing, multiple identical transitions were created.

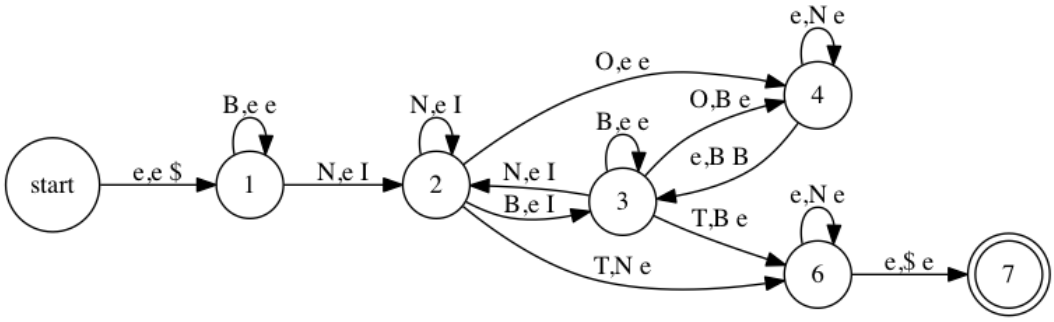

Figure 1: *Push Down Automata for Reverse Polish Notation. The small case 'e' stands for 'epsilon', the empty condition. 'B' is a blank char, 'N' any single-digit number char, 'O' one of the four base arithmetic operators, 'T' is EOF condition and the '$' symbol is used as marker for an empty stack.*

The interface of the transition contains merely one function of interest for the user, *transition_create*. All further functions are used internal to check and apply the transitions. This includes three functions to check for epsilon condition in the transitiona (*transition_checkReadEpsilon, transition_checkPopEpsilon, transition_checkPushEpsilon*) and three functions that wrap the *alphabet* functions for input, stack and push condition.

RPN Function

As mentioned earlier, it was chosen to separate verification of the input by the PDA from the calculation. The calculation is done by the function *rpn_calc*. This function assumes that the input statement is a valid RPN expression, hence the input is not validated and it should as such just be used in conjunction with the PDA datatype.

Compared to the PDA, the stack of the RPN calculator can hold multi digit integers. In fact, integers are written to the stack just after all their digits read from the input. Composing of multi-digit numbers is done by using an 'assembly' register and mulitplication by 10 in every 'round'. Blanks are interpreted as number terminators. In beginning of the input, after operators and after terminating a number read, blanks are disregarded. This behaviour agrees with the implemented rule set in the RPN PDA.

The parsing of *char* type operators into real aritmethic operators was inspired by a *Stack Overflow* blog post using a lookup table [2].

3 RPN Expression Checking

The RPN logic is a use case of the generic PDA datatypes described above. The automaton is represented as a graph in *figure 1*. It consists of seven states of which one is the *start state* 1. *State 6* is the only *accepting state*. There were a total of 15 transition functions defined (*table 2*). Each transition consists of three letters: The condition for *read*, *pop* and the letter to be *pushed* to the stack. Further, for each transition, the destination state in case of fulfilled *read* and *pop* criteria is indicated. The *stack-* and*input alphabets* are shown in *table* **??**. Besides the letters in the alphabet, each of the three positions in a transition statement can also be empty which is usually shown by an lower case *e* or ε (greek epsilon). In the current implementation, epsilon is obtained by *NULL*.

Table 1 *The states of the RPN Push Down Automaton*

Id	Name
0	start
1	initialized
2	non-terminated
3	terminated
4	processing operand
5	check terminal
6	success

Table 2 *The table below shows the transitions used for the Reversed Polish Notation Push Down Automaton. The letter e is used for epsilon, B as blank, the letter I for pushing input to the stack. Further the letter O and N stand for operator and number respectively. T stands for terminal and indicates the end of the input. The dollar sign is used as a letter of the stack alphabet to mark the first position. The column id shows the id number of the transition in the C program.*

id	source	destination	read	pop	push
1	start	1	e	e	$
2	1	1	B	e	e
3_1	1	2	N	e	I
3_2	2	2	N	e	I
5	2	3	B	e	I
4	2	4	O	e	e
6	2	6	T	N	e
3_3	3	2	N	e	I
13	3	3	B	e	e
7	3	4	O	B	e
8	3	6	T	B	e
9	4	4	e	N	e
10	4	5	e	B	B
11	6	6	e	N	e
12	6	7	e	$	e

Table 3 *Alphabets of the the RPN automaton*

alphabet	values
input	blank, number, operator, EOF
stack	'$', number

Table 4 *Test cases for the RPN PDA*

expression	expected	test
“”	Invalid expression	ok
“1”	10	ok
“0”	0	ok
“01”	1	ok
“1 1”	Invalid expression	ok
“a1”	Invalid expression	ok
“0 1/”	0	ok
“ 1 1 + ”	2	ok
“11111 11111*”	1234554321	ok
“111111 111111*”	-539247567	ok
“10 20-”	-10	ok
“1 2 + 3 - 4 * 5 /”	0	ok
“1 2 3 +”	Invalid expression	ok
“*”	Invalid expression	ok

Problem Solving Strategy

In the RPN automaton's first transition from *start* to *state 1*, a dollar symbol is pushed on the stack to mark it as *empty*. In the first state, the machine can read an arbitrary number of blanks ($B, \varepsilon \rightarrow \varepsilon$).

The main logic is in *states 2,3,4*. *State 2* and *3*, 'non-terminated' and 'terminate' refer to reading an operand from the input. It is considered 'terminated' when a number is followed by either an *operator* or a *blank*. In the 'terminated' state, either an *operator* or an *EOF* is accepted. When an *operator* is accepted from the input, *state 4* is accessed. Here, the last number (single or multi-digit) is popped sequentially from the stack ($\varepsilon, N \rightarrow \varepsilon$). The other possible transition requires to pop an blank from the stack ($\varepsilon, B \rightarrow B$). As blanks are just pushed between numbers, it guarantees that the automaton just proceeds after an operator if there are at least two numbers on the stack. The second operand is not popped from the stack as it represents now the result of the arithmetic operation. Instead, the number is 'terminated' again with a blank.

When the input string is used up, the automaton tranistions to *state 6*. This is possible either from 'terminated' or 'non-terminated' state. When coming from the 'terminated' state, the blank is popped. To make sure that there is just the the result left on the stack, from *state 6* only numbers (or finally the dollar symbol) can be popped. After popping the dollar symbol, the automaton ends up in the only accepted state.

Hence, the processing stops when there is no more viable transition to take. If the automaton is in a accepted state, the *succeed* flag is raised in the *pda* datatype.

3.1 Testing

A range of different inputs where tested. The testcases can be found in *table 4*. Most interesting cases are related to blanks either before, between or after numbers and operators. All cases that could be thought of succeeded. Neither the *pda* automaton nor the *rpn_calc* function check for the size of numbers. Hence it can happen that a result is too big for the datatype *int* and 'goes around'.

4 Extra Assignment - Bracket Matching

After the basic datatypes and the RPN automaton were implemented, it was investigated how flexible the program would be for another automaton. For this, the *Bracket Matching* automaton was implemented. After copying the *main* program of the RPN automaton, adjusting took just a few minutes. Basically, three states and five transitions had to be defined. Addiontally, three new *alphabet* functions were written: *isOpeningBracket*, *isClosingBracket* and *openingBracket*. Finally, the RPN calculation function was removed from main. The program uses all the same header files and is available in the file *bracket_automat.c*. The input expressions given in the lab description were tested and succeded.

5 Discussion

The chosen design is very flexible for implementing generic automata as demonstrated with the *Bracket Matching* automaton. The chosen separation of datatypes works well and allows for quick reconfiguration of the automaton. It can be argued whether the *State* datatype is needed at all. Instead, one could include also the source state in the transitions. Also the information about accepted / non-accepted state could be included in the transition itself. This would however result in more redundant information and the need for validating it to prevent inconsistent user input.

Admittedly, as mentioned in the description to the lab assignment, a grpah would probably have been a more natural basis for this datatype.

During the design and plannig phase, the largest concerns were about conversions between *char*'s and *int*'s aswell as single and multi-digit numbers on the stack. The *PDA* is easiest to implement when each stack register just holds one *char*. This corresponds also to the theoretical description of a *PDA*. However, for the calculation, multi-digit numbers have to be handled. It was tempting to implement multi-digit numbers on the stack but it would possibly make the pda less generic. After various rejected designs (including using separate *registers* in the *PDA* for calculation), the author finally settled for a puristic *PDA* implementation with a separated calculation function. The *PDA* works so to say as input validator which allows to keep the *RPN* calculator very simplistic as no input validation needs to be done.

References

[1] DV169VT16, cambro course homepage, 'datatypes'. `https://www8.cs.umu.se/kurser/5DV169/datatypes/index.html`, 2016. accessed: 2016-02-28.

[2] Stack Overflow, convert character into arithmetic operator without using switch case. `http://stackoverflow.com/a/22114821`, 2014. accessed: 2016-03-27.

[3] Reverse Polish Notation, wikipedia. `https://en.wikipedia.org/wiki/Reverse_Polish_notation`, 2016. accessed: 2016-03-28.

[4] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, Boston, USA, 2012.