

# Parsing Permutations

Jeffrey Clark

Elon University

[http://frodo.elon.edu  
/presentations/parsingTalk.pdf](http://frodo.elon.edu/presentations/parsingTalk.pdf)

March 21, 2003

## Introduction

I like to play around with algorithms for permutation groups, trying to find ways to handle large groups. Many programming languages have data structures that handle permutations nicely: C++ has `map` in the Standard Template Library, Perl has hashes, and Python has dictionaries. One of the hardest parts about working with permutations is translating their string representations into these data structures. As usual, input/output is dauntingly complicated. This presentation will outline a method for parsing strings into data structures that is accessible to students and faculty alike.

## Requirements for Permutation Input

When I refer to permutations in string representations, I am referring to permutations in cycle notation e.g.,  $(1, 2, 3)(4, 5)$ . In order to be able to parse input strings unambiguously, we will need some sort of rules that they will be required to follow.

- An input string should consist of one or more cycles, with optional spacing before the first cycle, between the cycles, and after the last cycle.
- A cycle consists of a left parenthesis, followed by a range of at least one point, followed by a right parenthesis.
- A range of points consists of at least one point, optionally preceded or succeeded by spacing, separated by at least one space and/or exactly one comma.

- A point is a non-negative integer string, i.e., a string consisting entirely of the digits 0–9.
- Spacing consists of a mixture of at least one space, tab, or new line. For the sake of convenience I will refer to all of these forms as space.

From these rules it is clear that the only permissible characters in our input string consist of forms of spacing, parentheses, commas, and digits. Any other characters appearing in the string should end the parsing with an appropriate error message.

## States

We will start by identifying the states of the parsing at various points in the input string. At the very beginning of the parsing, we have yet to see any cycles. We will call this state NOCYCLES and abbreviate it as  $q_n$ .

When we are in NOCYCLES, we can only read two kinds of characters: a space or a left parenthesis. If we read a space, we are left in the NOCYCLES state. If we read a left parenthesis, we enter a state where we have started a cycle, which we will call STARTCYCLE and abbreviate as  $q_s$ .

When we are in STARTCYCLE, we can read a space or a digit. If we read a space we are still in STARTCYCLE. If we read a digit we are now in a point, i.e., we are in the state INPOINT (written  $q_i$ ).

When we are in INPOINT, we can read a space, a comma, a digit, or a right parenthesis. If we read a space we enter the state AFTERPOINT ( $q_{ap}$ ). If we read a comma we enter the state AFTERPOINTCOMMA ( $q_{apc}$ ). If we read a digit, we remain in state INPOINT. If we read a right parenthesis, we are in AFTERCYCLE ( $q_{ac}$ ).

If we are in AFTERPOINT, we can read a space, a right parenthesis, a comma, or a digit. If we read a space we remain in the AFTERPOINT state. If we read a right parenthesis, we have closed a cycle and enter into the state AFTERCYCLE. If we read a comma, we enter into the state AFTERPOINTCOMMA. If we read a digit, we are now back in the state INPOINT.

If we are in AFTERPOINTCOMMA, we can read a space or a digit. If we read a space, we remain in AFTERPOINTCOMMA. If we read a digit, we are back in the state INPOINT.

If we are in AFTERCYCLE, we can read a space or a left parenthesis. If we read a space, we remain in AFTERCYCLE. If we read a left parenthesis, we enter into the state START-CYCLE. Note that the only state that valid input can end in is AFTERCYCLE.

We will also define, for the sake of completeness, an ERROR state ( $q_e$ ), such that anytime we see a symbol that isn't appropriate for one of the states already defined, we go into that ERROR state. Once there, for any symbol that we read we stay in the ERROR state.

For example, let's parse the following string with the indicated spaces:

(1, 2, 3) (4 5)

State	Next Character
NOCYCLES	space
NOCYCLES	(
STARTCYCLE	1
INPOINT	,
AFTERPOINTCOMMA	space
AFTERPOINTCOMMA	2
INPOINT	,
AFTERPOINTCOMMA	space
AFTERPOINTCOMMA	3
INPOINT	)
AFTERCYCLE	space
AFTERCYCLE	(
STARTCYCLE	4
INPOINT	space
AFTERPOINT	5
INPOINT	)
AFTERCYCLE	space
AFTERCYCLE	

## Deterministic Finite Automata

We've just described in the previous section a *Deterministic Finite Automaton*, or DFA for short. A DFA is a device for checking the syntax of a string, given a set of states, tokens, and transitions. More formally we require:

$Q$  = set of states

$q_0$  = initial state where  $q_0 \in Q$

$F$  = final states where  $F \subseteq Q$

$\Sigma$  = set of tokens

$\delta$  = transition function, where  $\delta: Q \times \Sigma \rightarrow Q$

To parse a string with a DFA, start at the initial state, and apply the transition function to the current state with each token in the string. If at the conclusion of the string the state is a final state in  $F$ , then the string is syntactically correct.

In our example with permutations, we have the following DFA:

$$Q = \{q_{ac}, q_{ap}, q_{apc}, q_e, q_i, q_n, q_s\}$$

$$\text{initial state} = q_n$$

$$F = \{q_{ac}\}$$

The transition function  $\delta$  is given by the following table, where the row corresponds to the input state and the column to the input character:

	,	0-9	(	)	_
$q_{ac}$	$q_e$	$q_e$	$q_s$	$q_e$	$q_{ac}$
$q_{ap}$	$q_{apc}$	$q_i$	$q_e$	$q_{ac}$	$q_{ap}$
$q_{apc}$	$q_e$	$q_i$	$q_e$	$q_e$	$q_{apc}$
$q_e$	$q_e$	$q_e$	$q_e$	$q_e$	$q_e$
$q_i$	$q_{apc}$	$q_i$	$q_e$	$q_{ac}$	$q_{ap}$
$q_n$	$q_e$	$q_e$	$q_s$	$q_e$	$q_n$
$q_s$	$q_e$	$q_i$	$q_e$	$q_e$	$q_s$

## Data Structures

We've outlined a method for determining if a given string is valid input for a permutation, but we haven't done anything yet to actually construct the permutation.

A permutation will be modeled by an associative array Map which is a data structure that acts like a function. Our goal will be to define a function that is bijective and whose finite domain is also its range. In order to ensure efficiency and uniqueness, we will require that Map have (as a data structure) no fixed points, the convention being that if a point is not in the (data structure) domain of Map, it is left fixed by Map.

Along the way we will be accumulating the digits of each point, the points in a cycle, and the current cycle when it is closed by a right parenthesis. We will model the digits by an ordered list `Digits` which will start out empty, will accumulate the digits as they are read, and then will be emptied as the digits are converted to a point.

We will use an ordered list `Points` to accumulate the points of a cycle. We start with an empty `Points` list. When we are done reading digits, we take the contents of `Digits`, concatenate them, and add the resulting point to the end of `Points`. (We will do some error checking here to avoid repeated points in a cycle.) When all of the `Points` have been read in a cycle, we convert them to a `Cycle`, and empty the `Points` list.

We will use an associative array to model a newly read cycle. It will start out as an empty associative array; once all of the points

$$p_1, p_2, \dots, p_n$$

of a cycle have been read, we define

$$\text{Cycle}(p_k) = p_{k+1}$$

for

$$k = 1, \dots, n - 1$$

and

$$\text{Cycle}(p_n) = p_1$$

We will use an ordered list `Cycles` to accumulate the cycles. Each time that we compute a cycle, we will append it to `Cycles`. Upon successful completion of the transitions, we will compose the cycles in `Cycles` to compute `Map`.

It is precisely at this point that we choose whether to treat our permutations as functions acting on the left or as operator acting on the right. In this paper we are choosing the former: when we compose the cycles in `Cycles` to get `Map`, we will treat them as functions acting on the left.

## Syntax versus Semantics

In our DFA, we were concerned only with syntax: what characters go together to make valid input. For each transition, we now want to examine what action to take, i.e., the semantics or meaning of the transition.

Most of the transitions do not change our data structures. The ones that do are listed below.

$\delta(q_{ap}, \mathbf{digit}) = q_i$ :

append the digit to Digits.

$\delta(q_{ap}, \mathbf{right\ parenthesis}) = q_{ac}$ :

convert Points to a Cycle, append it to Cycles, clear Points.

$\delta(q_{apc}, \mathbf{digit}) = q_i$ :

append the digit to the Digits list.

$$\delta(q_i, \mathbf{comma}) = q_{apc}:$$

concatenate elements of Digits into point, add point to Points. Empty Digits.

$$\delta(q_i, \mathbf{digit}) = q_i:$$

append the digit to Digits.

$$\delta(q_i, \mathbf{right\ parenthesis}) = q_{ac}:$$

concatenate Digits, add as new point to Points, empty Digits, make a cycle from Points, append the cycle to Cycles, empty Points.

$$\delta(q_i, \mathbf{space}) = q_{ap}:$$

concatenate Digits, add as a new point to Points, empty Digits.

$$\delta(q_s, \mathbf{digit}) = q_i:$$

add digit to Digits.

## Conclusion

Most mathematical programming requires processing string input. This can take a surprising amount of time and effort to get right. DFA's, as demonstrated with this example of processing permutations in string form, provide a rigorous framework for translating the strings into the appropriate data structures.

## References

This presentation is available at:

[http://frodo.elon.edu  
/presentations/parsingTalk.pdf](http://frodo.elon.edu/presentations/parsingTalk.pdf)

**DFA's:** Peter Linz, *An Introduction to Formal Languages and Automata*, Jones and Bartlett, 2001.

**C++:** Nicolai M. Josuttis, *The C++ Standard Library*, Addison Wesley, 1999.

**Perl:** [www.perl.org](http://www.perl.org)

**Python:** [www.python.org](http://www.python.org)