Generating Strings from PEGs

This document describes the String generation process from a well formed PEG.

The main utility for such a process is to be able to use test-base property on PEGs to

demonstrate the existence of errors on desired properties of PEGs.

There where 3 attempts to develop a String generator for PEGs, one which attempt

to develop a combinator-based generator, one based on the derivative of a PEG and finally an alternative semantic for the Ierusalimschy's Virtual Machine. This last alternative will be described in this document.

The Problem

The problem seems innocent enough : Given a PEG G , generate a String (preferable at random) that

will be accepted by G.

At a first glance this problem seems quite trivial. Consider the PEG G_A given below:

$$A \leftarrow aA/.$$
$$A$$

Generate a string for the G_A is trivial enough that even the methods for the GLC can be used here.

We start with the non-terminal A and follow the replacement process, i.e replacing the only occurrence of A by the its body. For this grammar this will work :

PEG	Action
A	
aA	Expand left alternative A
aaA	Expand left alternative A
aaaA	Expand left alternative A
aaa.	Expand Right alternative A

This process dos not always work and can produce incorrect results. Consider the grammar G_B

$$A \leftarrow aAa/a$$
$$A$$

This problem might, at first, look similar to the generation of a sentential form a context free grammar, however we can easily distinguish this problem from that one.

Suppose that you are given the following PEG, supposed

The Ierusalimschy's Virtual Machine:

This text provides a description of the PEG Virtual Machine , it's state and instructions and the brief

description of the semantics of the instructions. The optimizations are not discussed.

The PEG Virtual Machine is composed of a State the sequence of instructions and the input

String. The machine state has the following components:

$$\langle p, i, stk, cap \rangle$$

Where

- p : Is a pointer to the next instruction to be executed. This field cal also assume the value **Fail** to denote a failed state.
- i : Is a pointer to the current character (also called subject) on the String.
- stk : Is a stack that can contain either a single return address from a non-terminal call or a full execution context (code address, position on the String and a capture environment) representing a backtracking point.
- cap : Contains references to the start and end points of a partial matching of interest (captures).

In order to keep the notation short we will

The instructions of the PEG Virtual Machine are:

- $\langle p, i, stk, cap \rangle$ $\mathbf{Char\ c} \rightarrow \langle p+1, i+1, stk, cap \rangle$ whenever $\mathbf{c} = Str[i]$:
- $\langle p, i, stk, cap \rangle$ $\mathbf{Char\ c} \rightarrow \langle Fail, i, stk, cap \rangle$ whenever $\mathbf{c} \neq Str[i]$
- $\langle p, i, stk, cap \rangle$ $\mathbf{Jump\ l} \rightarrow \langle p+l, i, stk, cap \rangle$
- $\langle p, i, stk, cap \rangle$ $\mathbf{Choice\ l} \rightarrow \langle p+1, i, (p, i, cap) : stk, cap \rangle$

- $\langle p, i, stk, cap\rangle$ **Call l** $\rightarrow \langle p + l, i, (p + 1) : stk, cap\rangle$
- $\langle p, i, p' : stk, cap\rangle$ **Return** $\rightarrow \langle p', i, stk, cap\rangle$
- $\langle p, i, (p', i', cap') : stk, cap\rangle$ **Commit l** $\rightarrow \langle p + l, i, stk, cap\rangle$
- $\langle p, i, stk, cap\rangle$ **Capture k** $\rightarrow \langle p + 1, stk, (i, p) : cap\rangle$
- $\langle p, i, stk, cap\rangle$**Fail** $\rightarrow \langle Fail, i, stk, cap\rangle$
- $\langle Fail, i, p : stk, cap\rangle$ **any instruction** $\rightarrow \langle Fail, i, stk, cap\rangle$
- $\langle Fail, i, (p, i', cap') : stk, cap\rangle$ **any istruction** $\rightarrow \langle p, i', stk, cap\rangle$

There is also a instruction **End** that stops the machine signaling that the program has run to completion.

The compilation of a PEG expression to a program we use the *compile* described below:

code($e_1 e_2$) = code(e_1) code(e_2)

code(e_1 / e_2) = **Choice** L_1

code(e_1)

Commit L_2

L_1 : code(e_2)

L_2 : ...

code ($e^\star$) = L_1 : **Choice** L_2

code(e)

Commit L_1

L_2 : ...

code(!e) = **Choice** L_2

code(e)

Commit L_1

L_1 : **Fail**

L_2 : ...

A non-terminal call is compiled to a reference to the non-terminal name, which is later replaced by it" s address on the code. However for the sake of brevity and also to make code more

related to the original grammar we will use the non-terminal names as labels.

code(A) = **call** A

code($A \leftarrow e$) = A : code(e)

Return

Finally a full PEG (Σ, V, R, e) can be compiled as:

code(e)

End

for each $(V \leftarrow e) \in R$, code($V \leftarrow e$)

Flipping the Machine Semantics

In order to generate a String acceptable by the a PEG, we investigate the use of the PEG Virtual Machine's code in other to generate an input. We assume that the code was compiled from a parsing expression according to the compilation described by Ierusalimschy. We also assuming that the compiled PEG expression was *well formed*.

Our approach consists on the use of an alternative semantics for the PEG Virtual Machine so that it prints to an input instead of reading from it. This semantics must defined in a way that if a program prints an input, the same program recognizes the same input on the traditional semantics.

The sate is $\langle p, i, stk, stk \rangle$ where we got rid of the captures, and replace it with

another *stk* (stack). A special treatment for the input is needed to allow characters to be emitted to it. Each symbol of the input can now be a white symbol, written *w*, a

negation of a char, written $\neg\ char$, a simple char written as the character itself.

Whenever there is white in the input, anything can written to it. If there is character on the input, the that is the only character that can be written. If there is the negation of character, then any character except for that one can be written to input.

Before we move to the formal definition of the semantics considers the following example :

$!(ab)ac$

This code wold be compiled as :

Choice Seq

Char a

Char b

Commit N_{fail}

N_{fail} : **Fail**

Seq : **Char** a

Char c

End

Consider that the input "ab" is presented to this program, under the normal semantics of the machine. In this case the program progress until the instruction **Commit** N_{fail} which removes the backtracking point created by first instruction. The next instruction

causes the program to fail without any chance of recovery.

Consider now that the input "ac" is presented to this program, under the normal semantics of the machine. In this case the program progress until **Char** b where the program fail to match the input. At this point the program backtracks to the begin of the input and resumes its execution from the label Seq, reading the input to completion.

Suppose that now we want to use this program to generate an acceptable string.

We start with the same program and an string containing only a single white *w*.

Choice Seq

Char a

Char b

Commit N_{fail}

N_{fail} : **Fail**

Seq : **Char** a

Char c

End

w			
^			

1. The **Choice** executes normally.
2. The **Char** a writes a to the string and advances the input position pointer:

a	***w***		
	^		

3. The **Char** b writes b to the string and advances the input position pointer:

a	b	***w***	
		^	

4. At this point we ran the entire code for the Not operand of the PEG. Therefore the input recorded the pattern that would not be accepted. Now we execute then **Commit** instruction and it will remove the previous backtrack point and save it to the second stack.
5. Immediately after committing we **Fail**, a behavior only observe when we compile a Not expression. This is detected by the backtrack point on the second stack and therefore the fail backtracks on the input negating all characters on the input up to the backtrack point. The program also resumes from the backtrack point on the second stack, i.e from the Seq label.

$\neg a$	$\neg b$	***w***	
^			

6. The **Char** a wants to write a to the string and advances the input position pointer. Here we encounter a problem with our initial idea of the semantics. We can not write a to the input. **Lets assume a small modification on our initial idea**. We are only forbidden of write a negation if the next symbol is *w*.

a	$\neg b$	***w***	
	^		

7. The **Char** c writes c to the string and advances the input position pointer. Notice that now we cold not write b to the input.

a	c	***w***	
		^	

8. The **End** instruction ends the process, leaving ac on the input.

An important observation is that a PEG that is only a predicate Not operator, the result of this process will be a pattern left on the input. For example if instead of $!(ab)ac$ the PEG was $!(ab)$ the result string would have been $\neg a \neg b$.

The backtracking and Not problem

The approach based on the code has one mainly difficulty is to detected the not operation, and define the semantics to it in a more orthogonal way. Notice that the discussed approach works for the $!!(ab)ab$

This code wold be compiled as :

Choice Seq

Choice Not

Char a

Char b

Commit N_{fail} N_{fail} : **Fail**

Not : **Commit** N_{fail2} N_{fail2}: **Fail**

Seq : **Char** a

Char c

End

In order to the approach works correctly the **Commit** needs to check whether or not there is a backtrack point on the second stack (that will only contain one backtrack point at the time).