Vembyr Peg Generator Manual

Version 0.1

September 21, 2018

Jon Rafkind <jon@rafkind.com>

Contents

1 Introduction

Vembyr generates programs that use the Parsing Expression Grammar formalism. Input to Vembyr is specified by a BNF-like syntax. Currently Vembyr can generate C++, Ruby, and Python programs with C++ being the most optimized.

1.1 Current Status

C++ Generator: production quality (with some known bugs). The C++ backend is used in a production manner and is highly optimized.
Python Generator: production quality. The python generator is mainly used to bootstrap the system while parsing BNF input.
Ruby Generator: beta quality. The ruby parser is tested but not heavily used.

2 Files

peg.py The main executable file.
cpp_generator.py Generates c++ parsers.
cpp_header_generator.py Generates header files for c++ parsers.
python_generator.py Generates python parsers.
ruby_generator.py Generates ruby parsers
cpp_interpreter_generator.py Generates c++ parsers that use an interpreter style.
core.py Contains miscellaneous functions and classes that all other files require.

3 Command line usage

Running `peg.py` will produce the following help screen

Options:
-h, *–help*, *help* : Print this help
–help-syntax : Explain syntax of BNF (Backus-Naur form) for grammar files
–bnf : Generate BNF description (grammar language)
–ruby : Generate Ruby parser
–python : Generate Python parser
–cpp, *–c++* : Generate C++ parser
–h, : Generate a header file for the C++ parser
–save=filename : Save all generated parser output to a file, 'filename'. Without this option the output of `peg.py` will be sent to standard out.
–peg-name=name : Name the peg module 'name'. The intermediate peg module will be written as peg_<name>.py. Defaults to 'peg'.

Giving a syntactically correct input file as an argument will result in a message that says everything is ok.

```
$ ./peg.py sample
Grammar file 'sample' looks good!. Use some options to generate a peg
parser.
-h will list all available options.
```

An input file with an error in it will result in a parse error.

```
$ ./peg.py bad
Read up till line 10, column 1
'Mugen.Def
incl ude: '
          ^
Uh oh, couldn't parse bad. Are you sure its using BNF format?
```

Use one of `-cpp`, `-ruby`, `-python` to generate code from the input specification. By default the output program will be printed to standard out. Normal shell redirection can be used to put the output into a file or the `-save=file` option can be used.

4 Input

Grammar files consist of directives at the top of the file followed by BNF rules.

Example:

```
start-symbol: start
include: {{

#include <iostream>
static void got_a(){
    std::cout << "Got an 'aa'!" << std::endl;
}
}}

rules:
start = a* b "\\n"* <eof>
a = "aa" {{
     got_a();
   }}

b = "b"
```

`start-symbol` and `include` are directives whereas `rules` starts the BNF section.

4.1 Directives

Available options:
start-symbol : The starting non-terminal to use when parsing starts.

Example:

```
start-symbol: top
```

options: A list of options that modify the behavior of code generation.
`debug0` - Disable all debugging output when the PEG runs.
`debug1` - Enable some debugging.
`debug2` - Enable even more debugging.
`no-memo` - Disable the use of memoization completely.

Example:

```
options: debug0, no-memo
```

module: Puts all the generated code into a form that physically encapsulates it. For C++ this means namespaces, for Ruby this means the `module` keyword. In C++ the . is converted into nested namespaces so `Foo.Bar` would become `namespace Foo{ namespace Bar{ ... } }`.

Example:

```
module: Mugen.Def
```

include: Adds arbitrary text to the top of the file outside the any namespaces that might exist. This is useful for adding C++ #include directives. Use {{ and }} to delimit the text.

Example:

```
include: {{
#include <string>
#include <vector>
}}
```

code: Add arbitrary text that will appear inside any namespaces that might exist. This is useful for writing helper methods. Use {{ and }} to delimit the text.

Example:

```
code: {{
char * get(){
  return "test";
}
}}
```

4.2 BNF Syntax

The BNF section starts with a *rules* directive and all the following text is parsed as BNF syntax. There is no significance to the order of the rules.

A rule is given by a name followed by an = character and some clauses.

```
rules:
    top = "top"
```

Alternatives can be put on a new line preceded by the | symbol.

```
top = "top"
    | "bottom"
```

Actions can be given after the clause by writing code inside {{ }} enclosers.

```
top = "top" {{ printf("got top!\n"); }}
```

Pattern modifiers can be attached
* — repeat 0 or more times
+ — repeat 1 or more times
? — match 0 or 1

```
top = "top"* "bottom"?
```

A plain identifier will call out to another rule.

```
top = "top" bottom
bottom = "bottom"
```

The results of a pattern can be stored in a variable by prefixing the name of an identifier followed by `:`.

```
top = what:"top" {{ use(what); }}
```

The type of the variable `what` is `Value` which has the following methods on it.

*getValue(): void** — Get's the underlying object the pattern computed in its action.
getValues(): vector<Value> — Gets a list of vector objects when * or + is used.

Matched patterns can also be accessed through the $ variables.

```
top = "top" bottom "another" {{ use($1); // use "top"
                                use($2); // use bottom
                                use($3); // use "another"
                             }}
```

Literal strings can be followed by **{case}**. {case} does a case insensitive match on the string.

```
match_foo = "foo"{case}
```

Will match "foo", "foO", "FOO", or any other variation on "foo" with upper case letters.

Special patterns exist for specific circumstances.

- <**eof**> parses when the end of input is reached.
- <**ascii #**> parses a character with the given ascii code for when you need to parse a character with an unprintable character (such as any character above 128). Put a number where the # goes, anything from 0 to 255.
- <**utf8 #**> parses a utf8 character given as a hexidecimal codepoint.

  This example will parse the copyright sign '©' followed by the greek capital letter delta.

```
stuff = <utf8 a9> <utf8 394>
```

- <**void**> parses nothing.
- <**line**> parses nothing but returns an object that contains information about the current source position. Use the methods **getCurrentLine** and **getCurrentColumn** on this object.

  ```
  stuff = source:<line> "ok" {{ printf("current line %d column
  %d\n",
  getCurrentLine(source), getCurrentColumn(source)); }}
  ```

- <**predicate variable**> only continues with the current parse clause if the predicate is true. The argument to the predicate is a variable name that can be used for the code of the predicate. It starts out as **true** and if set to **false** in the predicate body the entire predicate will fail.

  This example only allows positive numbers to be parsed

  ```
  only_positive = x:number <predicate ok>{{ if (negative(x)){
                                              ok = false;
                                            }
                                         }}
  ```

5 Complete Example

Here is a complete example of a simple calculator. The non-peg code is C++.

```
start-symbol: start
code: {{
static Value add(const Value & a, const Value & b){
    return Value((void*)((int) a.getValue() + (int) b.getValue()));
}

static Value sub(const Value & a, const Value & b){
    return Value((void*)((int) a.getValue() - (int) b.getValue()));
}

static Value multiply(const Value & a, const Value & b){
    return Value((void*)((int) a.getValue() * (int) b.getValue()));
}

static Value divide(const Value & a, const Value & b){
    return Value((void*)((int) a.getValue() / (int) b.getValue()));
}

}}

rules:
        start = expression sw <eof> {{ value = $1; }}
        expression = expression2 expression1_rest($1)
        expression1_rest(a) = "+" expression2 e:{{value =
add(a,$2);}} expression1_rest(e)
                            | "-" expression2 e:{{value =
sub(a,$2);}} expression1_rest(e)
                            | <void> {{ value = a; }}

        expression2 = expression3 expression2_rest($1)
        expression2_rest(a) = "*" expression3 e:{{value = multi-
ply(a,$2);}} expression2_rest(e)
                            | "/" expression3 e:{{value = di-
vide(a,$2);}} expression2_rest(e)
                            | <void> {{ value = a; }}

        expression3 = number
                    | "(" expression ")" {{ value = $2; }}

        inline number = digit+ {{
            int total = 0;
            for (Value::iterator it = $1.getValues().begin(); it !=
$1.getValues().end(); it++){
                const Value & v = *it;
                char letter = (char) (int) v.getValue();
                total = (total * 10) + letter - '0';
            }
            value = (void*) total;
        }}
        inline sw = "\\n"*
        inline digit = [0123456789]
```