Librería PEG en C++

Qué es PEG

PEG (Parsing Expression Grammar) es un formalismo para especificar un parser recursivo descendente con backtracking ilimitado. La teoría puede verse en Wikipedia o en el paper original http://bford.info/pub/lang/peg.pdf de Bryan Ford, quien desarrolló una técnica llamada "packrat parsing" que permite que un parser PEG procese su entrada en tiempo lineal.

Como ejemplo práctico, veamos la gramática PEG de una calculadora que reconoce expresiones aritméticas con las cuatro operaciones básicas y paréntesis sobre números enteros:

```
WS            <- [ \t\f\r\n]*
SIGN          <- [+-]
DIGIT         <- [0-9]
NUMBER        <- SIGN? DIGIT+ WS
LPAR          <- '(' WS
RPAR          <- ')' WS
ADD           <- '+' WS
SUB           <- '-' WS
MUL           <- '*' WS
DIV           <- '/' WS

calc          <- WS expression
expression    <- term ( ADD term / SUB term )*
term          <- factor ( MUL factor / DIV factor )*
factor        <- NUMBER / LPAR expression RPAR
```

Esta calculadora acepta expresiones como

```
((1 + 2) * (-3 + 4) + 1) / 3 + 5
```

Si se le da una entrada correcta, el parser la consume y reporta éxito. Si la entrada no tiene la sintaxis adecuada, el parser no la consume y fracasa.

A semejanza de YACC, que es un compilador de gramáticas independientes del contexto (CFGs), hay compiladores de PEGs. Uno muy bueno es el paquete PEG/LEG de Ian Piumarta.

Aquí presentamos una librería de C++ cuya funcionalidad y estilo se inspiran en LEG. A diferencia de LEG, que es un compilador que toma la gramática y genera código en C, esta librería permite embeber una gramática PEG y parsearla directamente en código C++ sin procesamiento intermedio.

La librería está contenida en un solo encabezado (peg.h). Es conceptualmente similar a boost::spirit, pero mucho más pequeña, sencilla y menos eficiente.

Al igual que LEG, esta librería no implementa ninguna optimización del tipo "packrat parsing". En teoría, un parser recursivo descendente con backtracking ilimitado puede incurrir en tiempos exponenciales al parsear su entrada. En la práctica, sin embargo, la mayoría de las gramáticas requiere un backtracking modesto, pues de otra forma serían dificiles de comprender. Por lo tanto, el efecto de la optimización puede ser pequeño o incluso negativo. El lenguaje Pascal, por ejemplo, tiene una gramática LL(1) y no requiere backtracking.

Librería peg

Esta librería requiere como mínimo C++11.

Hemos tratado de que la sintaxis sea lo más parecida posible a la de LEG, aunque estamos restrigidos a los operadores disponibles en C++, con sus asociatividades y precedencias.

He aquí la misma calculadora del ejemplo anterior implementada con nuestra librería. Incluye acciones, para que además de reconocer la forma de la entrada evalúe e imprima el resultado de las expresiones ingresadas.

```
#include <iostream>
#include <string>

#define PEG_USE_SHARED_PTR

#include "peg.h"

using namespace std;
using namespace peg;

int main(int argc, char *argv[])
{
	matcher m;
	vect<int> val;

	// Reglas lexicográficas

	Rule WS, SIGN, DIGIT, NUMBER, LPAR, RPAR, ADD, SUB, MUL, DIV;

	WS		= *Ccl(" \t\f\r\n");
	SIGN		= Ccl("+-");
	DIGIT		= Ccl("0-9");
	NUMBER		= (~SIGN >> +DIGIT)-- >> WS >> Do([&] { val.push(stoi(m.text())); });
	LPAR		= Lit('(') >> WS;
	RPAR		= Lit(')') >> WS;
	ADD		= Lit('+') >> WS;
	SUB		= Lit('-') >> WS;
	MUL		= Lit('*') >> WS;
	DIV		= Lit('/') >> WS;

	// Calculadora

	Rule calc, expression, term, factor;

	calc		= WS >> expression	>> Do([&] { cout << val.top() << endl; val.pop(); });

	expression	= term >> *(
				  ADD >> term	>> Do([&] { val.top(-1) += val.top(); val.pop(); })
				| SUB >> term	>> Do([&] { val.top(-1) -= val.top(); val.pop(); })
				);

	term		= factor >> *(
				  MUL >> factor >> Do([&] { val.top(-1) *= val.top(); val.pop(); })
				| DIV >> factor >> Do([&] { val.top(-1) /= val.top(); val.pop(); })
				);

	factor		= NUMBER
			| LPAR >> expression >> RPAR;

	while ( calc.parse(m) )
		m.accept();
}
```

La definición de una regla utiliza = en lugar de <-.

La secuencia de expresiones, que en la forma original se expresa por simple concatenación, en C++ requiere un operador. Se usa el operador binario >>.

El operador de alternativa o elección priorizada (“prioritized choice”) es | en lugar de /.

Estas son las primitivas lexicográficas:

Lit(c) se satisface si se lee de la entrada el carácter c.

Lit(s) se satisface si se lee el string s.

Ccl(s) se satisface con cualquier carácter contenido en el string s. Dentro de s, el carácter - sirve para definir rangos de caracteres, salvo que sea el primero o el último carácter de s, en cuyo caso vale por sí mismo. Ccl significa “character class”. Ccl(“0-9”), por ejemplo, es la clase de los dígitos decimales. Si una clase comienza con el carácter ^ es una clase inversa o negada, que contiene todos los caracteres que no están en el string. Ccl(“^”) es una clase vacía negada. Se satisface con cualquier carácter y es un sinónimo de Any().

Any() se satisface con cualquier carácter. En la sintaxis original este elemento se representa con un punto (.). Solamente fracasa si el parser está parado al final de la entrada y no puede leer más (end-of-file).

Los operadores de repetición son prefijos y no posfijos como en el original.

> * indica cero o más veces.
>
> + indica una o más veces.
>
> ~ indica cero o una vez (opcional). Es ? posfijo en la notación original.

La definición de cada regla debe terminar con un punto y coma (;) pues es una sentencia de C++.

La primitiva Do() se satisface siempre sin consumir entrada. Agenda una acción para ser ejecutada si se satisface la rama de la gramática donde está colocada. Toma un argumento de tipo std::function<void()>, es decir, cualquier función, objeto funcional o lambda (como en este ejemplo) sin argumentos y que no retorna nada.

La primitiva Pred() designa un predicado semántico. Toma un argumento de tipo std::function<void(bool &)>, es decir, una función, objeto funcional o lambda que recibe una referencia a una variable booleana y no retorna nada. La función que se pasa a Pred() no se agenda para su ejecución posterior, se ejecuta en forma inmediata en el proceso de parseado y Pred() triunfa sin consumir entrada o fracasa, dependiendo del valor de la variable cuando la función retorna. La variable se inicializa en true (éxito) antes de llamar a la función, de modo que Pred() se satisface por defecto si la función no modifica o ignora su argumento.

Los predicados sintácticos utilizan los mismos operadores de prefijo que en la sintaxis original:

> &exp ("and-predicate") triunfa si en este punto de la entrada puede parsearse exp. No consume entrada ni agenda ninguna de las acciones que pueda contener exp.
>
> !exp ("not-predicate") es igual que el anterior, pero triunfa si exp fracasa, siempre sin consumir entrada ni agendar acciones. Este es el tipo de predicado que más se utiliza. La expresión !Any() significa end-of-file.

Para capturar el texto consumido por una o más expresiones, se utiliza el operador prefijo o posfijo --. Por ejemplo, la regla NUMBER utiliza la expresión (~SIGN >> +DIGIT)-- >> WS. Esto significa un signo opcional, seguido de uno o más dígitos, capturando la cadena de caracteres que contiene el eventual signo y los dígitos. El espacio en blanco que puede venir a continuación no está incluido en el texto capturado. Este texto está disponible como string en las acciones Do() y Pred() posteriores a la captura. La acción incluida en la regla NUMBER obtiene el texto capturado llamando a m.text(). Esto se explicará más adelante.

Las reglas

Cada regla es una instancia de la clase Rule. Definir una regla implica construir un árbol sintáctico, alocando memoria dinámicamente para las estructuras que forman los nodos del árbol. Si una gramática se define solamente una vez, probablemente no valga la pena preocuparse por la memoria alocada. En ese caso cada nodo del árbol sintáctico apunta a sus hijos utilizando punteros comunes. Si se define la macro PEG_USE_SHARED_PTR antes de incluir peg.h, estos punteros se reemplazan por punteros inteligentes del tipo std::shared_ptr, que se encargan de liberar la memoria cuando las reglas ya no se utilizan más.

En nuestro ejemplo hemos definido esta macro básicamente para comprobar que la versión con punteros inteligentes funciona, aunque en realidad es innecesario (la gramática se define una sola vez, en la función main).

Cuando se construye una gramática (conjunto de reglas), el orden de definición de las reglas puede ser cualquiera. Cualquier regla puede hacer mención a cualquier otra, con independencia de que haya sido previamente definida o no. Esto es necesario para poder definir gramáticas recursivas, que son muy comunes.

La clase Rule está definida de tal manera que, cuando se utiliza como una expresión, una regla se transforma en una referencia a sí misma. Por eso el uso de la clase es muy restringido: las reglas solamente puede construirse por defecto y luego puede asignársele una expresión o un tipo que se transforme implícitamente en una expresión. No pueden generarse temporarios, pues producirían expresiones con referencias flotantes. Por eso no existen constructores salvo el constructor por defecto, y se ha eliminado (delete) el constructor de copia. El único temporario posible es Rule(). Debe evitarse su uso, aunque de todas maneras no serviría para nada.

El operador de asignación de copia tampoco se comporta de la manera tradicional. Sería de esperar que una sentencia como r = r no hiciera nada. Pero si r es una regla, esta asignación la transforma en recursiva por izquierda, pues se llama a sí misma sin leer nada de la entrada. Si se intenta parsear esta regla, desborda el stack.

La recursión izquierda también puede surgir por ciclos de referencias. En la calculadora hay un ciclo:

> expression → term → factor → expression

Si bien las primeras dos referencias (expression → term → factor) son directas, la que cierra el ciclo (factor → expression) requiere la lectura previa de un token LPAR (paréntesis izquierdo). Si no fuera así, la gramática sería recursiva por izquierda y el parser caería en un lazo infinito. Esta es una limitación de las gramáticas PEG: no pueden existir ciclos de referencias que se cierren sin la lectura de por lo menos un carácter de la entrada.

Una vez construida, la gramática puede utilizarse llamando al método parse() de la regla de inicio (en nuestro ejemplo calc). Parse() toma un argumento de tipo matcher (ver más abajo). Retorna true si la entrada satisface la gramática, y

false en caso contrario. En ambos casos, cuando parse() retorna el matcher contiene en su interior toda la entrada que fue leída durante el parseado, tanto la parte que satisface la gramática como la que no. Por ejemplo, supongamos que se ingresa el siguiente texto a la calculadora:

(1 + 2) * 3 Hola

La llamada a calc.parse(m) retorna true. En el proceso de parseado, el matcher avanzó el puntero de entrada hasta la H de Hola (el primer carácter que no satisface la gramática). Este es el límite de la parte de la entrada que será consumida por la gramática. También contiene un vector de acciones (agendadas por Do() durante el parseado) que deben ejecutarse para evaluar la expresión. La llamada m.accept() ejecuta las acciones, resetea el vector que las contiene y luego descarta la parte de la entrada que ha sido consumida. La entrada ahora comienza en la H de Hola. La siguiente llamada a calc.parse() retorna false, pues Hola no satisface la gramática.

La clase matcher

Cada instancia de un parser PEG utiliza una gramática (conjunto de reglas) y una instancia de la clase matcher. Esta clase proporciona servicios al parser. La mayoría de sus métodos son de uso interno de la librería, solamente unos pocos son públicos y pueden ser llamados directamente:

- El constructor toma un argumento opcional del tipo std::istream, cuyo valor por defecto es std::cin. Pasando un std::istream adecuado al constructor la entrada puede tomarse desde cualquier origen.
- El método accept() debe llamarse después de parsear, para ejecutar las acciones agendadas durante el parseado y descartar la entrada consumida.
- El método clear() descarta toda la entrada ingresada y borra las acciones agendadas.
- Tanto desde las acciones como desde los predicados semánticos puede llamarse al método text(), que devuelve un string con la última captura de texto previamente cerrada. Las capturas de texto pueden anidarse (esto resulta especialmente útil para depuración).

La clase vect

En general, las gramáticas recursivas requieren el uso de un stack de valores. Para eso puede usarse la clase vect, un sencillo envoltorio alrededor de std::vector que introduce algunas comodidades.

El operador [] hace crecer el vector automáticamente cuando se lo indexa fuera de su límite actual.

Hay métodos que facilitan el uso del vector como stack:

- El método push() agrega un elemento al final del vector.
- El método top() permite direccionar con respecto al tope del stack. Top() o top(0) accede al último elemento del vector, top(-1) al penúltimo, etc.
- El método pop() permite eliminar elementos del final del vector. Pop() o pop(1) elimina el último elemento, pop(2) los dos últimos, etc.

Los métodos reserve(), size(), resize() y clear() son los mismos de std::vector.

Manejar el stack de valores manualmente es fácil en un ejemplo sencillo como el de la calculadora, pero en casos complicados conviene automatizarlo. Para eso se cuenta con clases especiales que manejan sus índices en forma automática con ayuda del matcher.

Manejo automático del stack de valores – clases value_stack y value_map

La siguiente versión de la calculadora maneja el stack de valores automáticamente utilizando la clase value_stack, y elimina todas las llamadas a las primitivas, salvo Ccl(), utilizando un mecanismo de sobrecarga de operadores que se explicará más adelante:

```
#include <iostream>
#include <string>

#define PEG_USE_SHARED_PTR

#include "peg.h"

using namespace std;
using namespace peg;

int main(int argc, char *argv[])
{
	matcher m;
	value_stack<int> val(m);

	// Reglas lexicográficas

	Rule WS, SIGN, DIGIT, NUMBER, LPAR, RPAR, ADD, SUB, MUL, DIV;

	WS		= *Ccl(" \t\f\r\n");
	SIGN		= Ccl("+-");
	DIGIT		= Ccl("0-9");
	NUMBER		= (~SIGN >> +DIGIT)-- >> WS   >> [&] { val[0] = stoi(m.text()); };
	LPAR		= '(' >> WS;
	RPAR		= ')' >> WS;
	ADD		= '+' >> WS;
	SUB		= '-' >> WS;
	MUL		= '*' >> WS;
	DIV		= '/' >> WS;

	// Calculadora

	Rule calc, expression, term, factor;

	calc		= WS >> expression		>> [&] { cout << val[1] << endl; };

	expression	= term >> *(
				  ADD >> term		>> [&] { val[0] += val[2]; }
				| SUB >> term		>> [&] { val[0] -= val[2]; }
				);

	term		= factor >> *(
				  MUL >> factor		>> [&] { val[0] *= val[2]; }
				| DIV >> factor		>> [&] { val[0] /= val[2]; }
				);

	factor		= NUMBER
			| LPAR >> expression >> RPAR  >> [&] { val[0] = val[1]; };

	while ( calc.parse(m) )
		m.accept();
}
```

La clase value_stack se construye con una referencia al matcher (m). El constructor, el operador [] y el método clear() son su única interfaz pública.

Cada expresión de una regla puede retornar un valor en un lugar del stack, de acuerdo a su posición en la secuencia. Las acciones de la regla acceden a estos valores usando índices relativos al inicio de la misma. Las reglas retornan un valor asignando val[0] (siendo val el stack de valores). Si las acciones de una regla no asignan val[0], la regla retorna el valor de su primera expresión.

Para conocer el índice de cada expresión dentro de una regla, basta con contar la cantidad de >> que la separan del inicio de la misma. Por ejemplo, en la regla term, el primer factor está en la posición 0 y el segundo en la posición 2. Si una expresión tiene alternativas, ocupa en el stack la cantidad de posiciones de su alternativa más larga. Por ejemplo:

```
	r = e0 >> (
		  a1 >> a2
		| b1 >> b2 >> b3 >> b4
		)			>> e5;
```

La expresión entre paréntesis ocupa 4 posiciones en el stack. El resultado de e5 aparece siempre en val[5], con independencia de cuál de las alternativas (a1 … a2 o b1 … b4) se satisface al parsear.

Esta manera de manejar el stack de valores es similar a la que utiliza YACC, siendo val[0] equivalente a $$ y a $1, val[1] a $2, etc.

El manejo automático del stack con value_stack es más fácil que el manejo manual pero puede ser menos eficiente, porque hay posiciones que no se utilizan. La clase value_stack está implementada con vect<T>. El tamaño del vector crece automáticamente hasta incluir el mayor índice utilizado. Al crecer llena todos los lugares intermedios con el valor por defecto T(). También puede verse obligado a realocar memoria y copiar los valores anteriores. Esto último puede mitigarse o evitarse si se configura el stack con una cantidad inicial suficiente de posiciones de memoria. El constructor de value_stack acepta un segundo argumento opcional con la cantidad de posiciones de memoria a reservar, por defecto 128.

La clase value_map está implementada con std::map<int, T> y tiene la misma interfaz pública que value_stack, salvo que el constructor no acepta el segundo argumento pues no necesita reservar memoria. Como la implementación utiliza un mapa en vez de un vector, acepta cualquier índice sin abrir entradas para los índices no utilizados, no hace realocaciones de memoria ni copia valores anteriores. Resulta más eficiente que value_stack si el stack tiene muchas posiciones sin utilizar y el constructor por defecto o la asignación por copia son costosos para el tipo T.

Es posible utilizar más de un stack de valores con el mismo matcher; en ese caso corren en forma paralela.

Eliminación de primitivas

Los operadores binarios de las reglas están sobrecargados para aceptar en uno de sus argumentos (pero no en ambos) valores que corresponden al tipo del argumento de una de las primitivas y generan automáticamente el llamado a la primitiva correspondiente. El operador de asignación de la clase Rule está sobrecargado de la misma manera.

std::string s	se convierte en Lit(s)
const char *s	se convierte en Lit(std::string(s))
char c	se convierte en Lit(c)
std::function<void()> f	se convierte en Do(f)
std::function<void(bool &)> f	se convierte en Pred(f)

Estas conversiones automáticas permiten en muchos casos eliminar los llamados explícitos a las primitivas y reemplazarlas por sus argumentos. En la segunda versión de la calculadora hemos eliminado de esta manera todas las primitivas Lit() y Do(). Nótese que Ccl(s) no puede eliminarse, pues s se convierte en Lit(s).

Las llamadas a Lit(c), Lit(s) y Ccl(s) con argumentos literales pueden también reemplazarse usando literales definidos por el usuario:

‘c’_lit	se convierte en Lit(c)
“ab”_lit	se convierte en Lit(std::string(“ab”, 2))
“a-z”_ccl	se convierte en Ccl(std::string(“a-z”, 3))

El uso de estos literales es básicamente una cuestión de preferencias estéticas. Adicionalmente, los operadores literales de string permiten incluir caracteres nulos en los literales, pues el compilador llama al operador literal correspondiente pasándole el tamaño. Los operadores están declarados en el espacio de nombres inline peg::literals.

Expresiones adjuntas

Una expresión puede adjuntarse a otra utilizando la sintaxis e1 (e2). Esto hace que e2 se parsee inmediatamente después de parsearse exitosamente e1. El comportamiento de e1 (e2) es similar al de e1 >> e2, salvo que e2 no ocupa lugar en el stack de valores. Esto resulta especialmente útil para la depuración en casos en los que se esté utilizando manejo automático del stack de valores, pues pueden insertarse acciones en cualquier lugar de las reglas sin modificar los índices de las expresiones. Por ejemplo, supongamos que deseamos mostrar la presencia de signos + en una expresión aritmética parseada con la calculadora. Esto puede hacerse fácilmente:

```
expression     = term >> *(
                          ADD                 ([&] { cout << “add\n”; })
                              >> term         >> [&] { val[0] += val[2]; }
                        | SUB >> term         >> [&] { val[0] -= val[2]; }
                        );
```

Hemos adjuntado una acción a la expresión ADD, sin perturbar la ubicación del segundo term en la posición 2 del stack. Si se ingresa la expresión ((1+2)*3+4)*5+6, la salida de la calculadora es ahora la siguiente:

```
add
add
add
71
```

Las expresiones adjuntas pueden ser arbitrariamente complejas. El operador () se ha sobrecargado para los mismos tipos que los operadores binarios, lo que permite eliminar la llamada a Do() en el ejemplo anterior.

Adjuntar acciones ahorra posiciones no utilizadas en el stack de valores, aunque esto es irrelevante si se utiliza value_map en lugar de value_stack. La regla anterior puede reescribirse con todas las acciones adjuntas sin alterar su funcionamiento:

```
expression     = term >> *(
                           ADD                 ([&] { cout << “add\n”;  })
                               >> term         ([&] { val[0] += val[2]; })
                         | SUB >> term         ([&] { val[0] -= val[2]; })
                         );
```

Macros para acciones y predicados semánticos

Peg.h define un par de macros que permiten el uso de lambdas en acciones y predicados semánticos utilizando una notación muy concisa. Esta es la macro para acciones:

```
#define _(code...)    (peg::Do([&]{ code }))
```

El argumento de la macro es el cuerpo de una función de tipo void(). Reescribiendo el ejemplo anterior utilizando este macro se obtiene un código muy conciso:

```
expression     = term >> *(
                           ADD                 _( cout << “add\n”;  )
                               >> term         _( val[0] += val[2]; )
                         | SUB >> term         _( val[0] -= val[2]; )
                         );
```

El segundo macro es para predicados semánticos:

```
#define __(code...)   (peg::Pred([&](bool &__r){ __r = ([&]()-> bool { code })(); }))
```

En este caso el argumento de la macro es el cuerpo de una función de tipo bool(). Si esta función retorna true el predicado parsea exitosamente, en caso contrario fracasa. Por ejemplo, la siguiente regla siempre fracasa:

```
Rule fail;
fail = __( return false; );
```

Nótese que estos macros se expanden como una expresión entre paréntesis. Esto permite ubicarlos en cualquier posición dentro de una regla. Si se los coloca inmediatamente a la derecha de otra expresión se adjuntan a la misma, pues los paréntesis se interpretan como el operador de llamada a función. En caso contrario los paréntesis son redundantes y no tienen efecto.

Versión final de la calculadora

Esta versión final de la calculadora reemplaza las llamadas a Ccl() por literales definidos por el usuario y adjunta todas las acciones usando la macro _(...).

```
#include <iostream>
#include <string>

#define PEG_USE_SHARED_PTR

#include "peg.h"

using namespace std;
using namespace peg;

int main(int argc, char *argv[])
{
	matcher m;
	value_stack<int> val(m);

	// Lexical rules

	Rule WS, SIGN, DIGIT, NUMBER, LPAR, RPAR, ADD, SUB, MUL, DIV;

	WS		= *" \t\f\r\n"_ccl;
	SIGN		= "+-"_ccl;
	DIGIT		= "0-9"_ccl;
	NUMBER		= (~SIGN >> +DIGIT)-- >> WS   _( val[0] = stoi(m.text()); );
	LPAR		= '(' >> WS;
	RPAR		= ')' >> WS;
	ADD		= '+' >> WS;
	SUB		= '-' >> WS;
	MUL		= '*' >> WS;
	DIV		= '/' >> WS;

	// Calculator

	Rule calc, expression, term, factor;

	calc		= WS >> expression		_( cout << val[1] << endl; );

	expression	= term >> *(
				  ADD >> term		_( val[0] += val[2]; )
				| SUB >> term		_( val[0] -= val[2]; )
				);

	term		= factor >> *(
				  MUL >> factor		_( val[0] *= val[2]; )
				| DIV >> factor		_( val[0] /= val[2]; )
				);

	factor		= NUMBER
			| LPAR >> expression >> RPAR  _( val[0] = val[1]; );

	while ( calc.parse(m) )
		m.accept();
}
```

Verificación de una gramática

Si se define la macro PEG_DEBUG antes de incluir peg.h, las reglas se compilan con un método que permite verificar una gramática. Se aplica sobre la regla de inicio, en nuestro ejemplo:

```
	calc.check()
```

Check() sigue todos los caminos de la gramática que se originan en la regla de inicio, y genera una excepción del tipo Rule::bad_rule si detecta reglas no inicializadas o potencial recursión por la izquierda, es decir, caminos cerrados que pueden recorrerse sin consumir entrada.

No puede llamarse más de una vez sobre una misma gramática, aunque esto carecería de utilidad. Una vez verificada una gramática, pueden eliminarse la macro y la llamada a check().

Además, pueden asignarse nombres a las reglas (del tipo const char *). Por ejemplo,

```
	calc.name = “calc”;
```

En modo debug peg.h define the macro peg_debug():

```
	#define peg_debug(rule) rule.name = #rule
```

De modo que para asignar un nombre a calc, lo más fácil es:

```
peg_debug(calc);
```

Las reglas con nombres asignados imprimen mensajes de depuración cuando check() las visita. Esto es útil para entender cómo recorre check() la gramática y determinar el motivo de eventuales fallas.

Por ejemplo, definimos PEG_DEBUG en la calculadora. Después de construir la gramática, escribimos:

```
peg_debug(NUMBER);
peg_debug(LPAR);
peg_debug(RPAR);
peg_debug(ADD);
peg_debug(SUB);
peg_debug(MUL);
peg_debug(DIV);
peg_debug(calc);
peg_debug(expression);
peg_debug(term);
peg_debug(factor);

calc.check();
```

La ejecución del programa produce la siguiente salida de error:

```
calc
| expression
| | term
| | | factor
| | | | NUMBER
| | | | LPAR
| | | | expression (r)
| | | | RPAR
| | | MUL
| | | factor (v)
| | | DIV
| | | factor (v)
| | ADD
| | term (v)
| | SUB
| | term (v)
calc: check OK
```

El nombre de cada regla se imprime cuando se la visita, con una indentación que refleja el nivel de anidamiento de las llamadas. El árbol sintáctico de cada regla se visita solamente una vez. Las reglas que ya fueron visitadas se marcan con (v) y las llamadas recursivas con (r).

Ahora modificamos la gramática en la última regla (factor), haciendo LPAR opcional (~LPAR). Con este cambio la gramática se vuelve recursiva por izquierda, pues el ciclo expression → term → factor → expression puede cerrarse sin consumir entrada. Check() detecta el problema:

```
calc
| expression
| | term
| | | factor
| | | | NUMBER
| | | | LPAR
| | | | expression (r)
terminate called after throwing an instance of 'peg::Rule::bad_rule'
  what():  Left-recursive rule
```

No se han depurado algunas reglas lexicográficas de bajo nivel como WS (white space), porque complicarían la salida del debugger. Check() siempre verifica toda la gramática, con independencia de qué reglas se depuren (o ninguna). Normalmente se llama primero a check() sin depuración; si la gramática está OK no se necesita nada más. Si check() arroja una excepción, pueden incorporarse graualmente reglas a la depuración hasta que la causa del problema quede clara.

Soporte de Unicode

Esta librería soporta Unicode.

Los strings y los streams de entrada con caracteres no ascii deben estar codificados en UTF8. Esto es así por defecto en Linux; en otros entornos, puede ser necesario usar el prefijo u8 en los literales de string para forzar la codificación UTF8. Los prefijos u (16 bits), U (32 bits) y L (carácter ancho) no pueden aplicarse a los literales de string, pues solamente se soportan strings con caracteres de 8 bits. Esto produce un error de compilación.

```
"aa"                        (ok, string ascii)
u8"áá"                      (ok, UTF8 explícito: 0xc3, 0xa1, 0xc3, 0xa1, 0)
u8"\U000000e1\U000000e1"    (idem, usando puntos de código)
"áá"                        (ok en Linux)

U"áá"                       (error de compilación, strings anchos no soportados)
```

Los literales de string válidos aceptan los sufijos _lit y _ccl según sea necesario.

Los caracteres aislados son de 32 bits (char32_t) y pueden contener cualquier punto de código Unicode. Los literales ascii (7 bits) como 'a' se promueven automáticamente a 32 bits.

Los literales no ascii deben definirse explícitamente como de tipo char32_t anteponiéndoles el prefijo U. La omisión del prefijo es siempre un error. Dependiendo del contexto, esto puede generar errores de compilación, advertencias sobre el uso de constantes multichar, o pasar desapercibido.

```
'a'                         (ok, valor ascii promovido implícitamente char32_t)
U'á'                        (ok, char32_t explícito)
U'\xe1'                     (idem, usando punto de código)
U'á'_lit                    (ok, char32_t explícito)

'á'                         (incorrecto, posible advertencia del compilador)
'á'_lit                     (error de compilación, operator""_lit(int) no definido)
```

Los literales de carácter válidos aceptan el sufijo _lit.