

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 el utilitario LEG de Ian Piumarta. 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 difíciles 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 restringidos 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      = *Ccl(" \t\f\r\n");
    Rule SIGN    = Ccl("+ -");
    Rule DIGIT   = Ccl("0-9");
    Rule NUMBER  = (~SIGN >> +DIGIT)-- >> WS >> Do([&] { val.push(stoi(m.text())); });
    Rule LPAR    = Chr('(') >> WS;
    Rule RPAR    = Chr(')') >> WS;
    Rule ADD     = Chr('+') >> WS;
    Rule SUB     = Chr('-') >> WS;
    Rule MUL     = Chr('*') >> WS;
    Rule DIV     = Chr('/') >> 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 /.

Para definir las primitivas lexicográficas se utilizan funciones constructoras:

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

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. No tiene sentido definir clases de un solo carácter, ya que Chr() resulta mucho más económico. Por lo tanto, 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).

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

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 --, que puede ser 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, allocating 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 asignada. En ese caso cada nodo del árbol sintáctico apunta a sus hijos utilizando punteros comunes. Si se define el 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 este 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.

En la calculadora hay un ciclo de referencias:

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

```
Rule 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 constructor y el operador de asignación de la clase `Rule` están sobrecargados de la misma manera.

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

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 `Chr()` y `Do()`. Nótese que `Ccl(s)` no puede eliminarse, pues `s` se convierte en `Lit(s)`.

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

<code>"ab"_lit</code>	se convierte en <code>Lit(std::string("ab", 2))</code>
<code>"a-z"_ccl</code>	se convierte en <code>Ccl(std::string("a-z", 3))</code>
<code>'c'_chr</code>	se convierte en <code>Chr(c)</code>

El uso de estos literales es básicamente una cuestión de preferencias estéticas. Adicionalmente, `_lit` y `_ccl` 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.

Adosar 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 adosadas sin alterar su funcionamiento:

```

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

```

Versión final de la calculadora

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

```

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

```