Parsing Expresions Grammars . Desarrollo de EgLib y EgParserGen

Betina Eloisa Peredo (bperedo84@gmail.com)
Departamento de Computación
Universidad Nacional de Río Cuarto
Río Cuarto, Córdoba Argentina

Yanina Soledad Boccardo (yani.boccardo@gmail.com)
Departamento de Computación
Universidad Nacional de Río Cuarto
Río Cuarto, Córdoba Argentina

1 de julio de 2010

Resumen

Las *Parsing expression grammars*[1], son un formalismo que describen un lenguaje formal en términos de un conjunto de reglas, para el reconocimiento de cadenas en el lenguaje. Dichos reconocedores de lenguajes (parsers) son especificados de tal forma que no es necesario dividir la especificación en la parte léxica y la parte libre de contexto. Una especificación de una parsing expression grammar puede verse como la descripción de un parser descendente recursivo con backtracking. Las alternativas en la gramática se procesan en el orden dado en la especificación, lo que se conoce como *priorized choice*. Las especificaciones basadas en este formalismo son más compactas y los parsers aceptan una clase de lenguajes más amplia que la mayoría de los parsers clásicos determinísticos, como los basados en gramáticas LL(k) o LR(k).
Si bien existen varias implementaciones de *expression grammars*, generalmente no incluyen características deseables como informe y recuperación de errores o generación automática de ASTs (Abstract Syntac Tree).
En particular este trabajo describe las Parsing Expression Grammars y el desarrollo de EgLib y EgParserGen y sus características que las diferencian de herramientas similares.

1. Introducción

Parsing o análisis sintáctico, es una de las ramas de las Ciencias de la Computación más estudiadas, debido a su utilidad en distintas disciplinas. Los aportes más importantes para estas técnicas fueron introducidas en 1970, por los científicos Aho, Ullman, Knuth quienes dieron consistencia los conceptos y varias de las técnicas de parsing, momento en donde fueron detectados sus numerosos patrones de funcionamiento. Las gramáticas libres de contexto (CFGs) y los parsers correspondientes a subfamilias de CFGs (LL o LR) ofrecen un limitado poder de expresión y extensibilidad y esto representa un inconveniente para el manejo de la evolución de los lenguajes de programación y sus implementaciones.

Las PEGs comparten muchas de sus construcciones con la notación EBNF, con la diferencia que definen su sintaxis en términos de reconocimiento de cadenas en vez de estar basados en sistemas generativos. Otra clave diferente recae en las alternativas con prioridad, en vez de las alternativas desordenadas usadas en las CFGs. Como resultado, las PEGs evitan las ambigüedades innecesarias y son fácilmente modificables.

Otra diferencia son los predicados sintácticos de las PEGs, que ofrecen mayor expresividad, haciendo corresponder expresiones con datos de entrada sin consumir ningún caracter.

Algunas características de las PEGs son que proveen ilimitada cantidad de look-ahead e integran la parte léxica con la parte sintáctica, lo cual evita la adición de nuevos tokens a la gramática.

Por otro lado, los analizadores sintácticos o parsers están cada vez más al alcance de los programadores. En un comienzo tales herramientas eran escritas a mano, pero luego esta tarea fue reemplazada por los generadores de parsers (ej.: YACC), y por la integración de las gramáticas y dichos generadores directamente en el lenguaje receptor.

El objetivo principal de este proyecto fue desarrollar un generador de parser que acepte las gramáticas PEGs. Para ello se implementaron la biblioteca EgLib y la herramienta EgParserGen.

EgLib [4] implementa un DSL (Domain Specific Language) interno, ya que permite escribir una gramática en C++ aproximándose a la sintaxis de las PEGs, permitiendo que las especificaciones de las gramáticas PEGs se pueden mezclar libremente con código C++.

EgParserGen implementa un DSL externo, ya que procesa las PEGs escritas con la sintaxis original descrita por Ford[2]. Luego reescribe una nueva gramática en C++ que EgLib pueda procesar.

También se plantearon objetivos específicos con respecto a:
Funcionalidad : Realizar la mayor cantidad de tareas posibles. Para ello la biblioteca EgLib ofrece, además de la generación de parsers, acciones semánticas, generación de AST e informe y recuperación de errores.
Usabilidad: Proveer facilidad de operación y aprendizaje a los usuarios, para que puedan utilizar las herramientas de forma rápida y sencilla. Para lograrlo, la biblioteca utilizó la sobre carga de operadores en C++ para aproximarse a la sintaxis de PEG. Por otro lado se implementó la herramienta EgParserGen, la

cual con total transparencia acepta una PEG.
Eficiencia: Tratar de utilizar la menor cantidad de recursos tales como tiempo y memoria de ejecución. Este es uno de los motivos por el cual se eligió el lenguaje de programación C++, ya que es extremadamente eficiente. Además los algoritmos estan implementados cuidadosamente de manera que se reduzcan las llamadas al sistema.

Cabe destacar que si bien existen varios generadores de parsers para las PEGs, existen algunas características deseables por los usuarios que estos no las proporcionan implícitamente, esto fue lo que nos motivo a desarrollar EgLib y EgParserGen, logrando transparencia y sobre todo que se diferencien de las actuales, combinando las funcionalidades que estas ofrecen(acciones semánticas), con otras mucho mas requeridas por los usuarios (generación automática de AST e informe y recuperación de errores).

En este artículo se describen en la sección 2 los conceptos básicos, características, propiedades y ejemplos de las Parsing Expression Grammars. En la sección 3 se describe la biblioteca EgLib se detallan las etapas de diseño e implementación de la librería. En la sección 4 se detalla la herramienta EgParserGen. En la sección 5 se compara EgLib con otras implementaciones, destacando los aspectos más importantes de las distintas tecnologías y destacando el nivel de performance de cada uno de ellos. En la sección 6 se presentan los trabajos futuros y conclusiones. Por último en los apéndices se muestran una serie de ejemplos tanto de EgLib como de EgParserGen.

2. Parsing Expression Grammars

Las *Parsing Expression Grammars* son un formalismo que permiten la especificación de reconocedores de lenguajes (parsers). Proveen una alternativa a las CFG, para la especificación formal de sintaxis similares a las BNF, pero definen su sintaxis en términos de reconocimiento de cadenas, en vez de construirlas y solucionan el problema de la ambigüedad de manera muy particular: no introduciéndola.

Las gramáticas BNF, están basadas en sistemas generativos, en los cuales un lenguaje es definido mediante un conjunto de reglas aplicadas recursivamente para generar cadenas de lenguaje; en contraste las PEGs están basadas en sistemas de reconocimiento de tokens, y consisten en un conjunto de reglas y predicados que deciden si una cadena pertenece o no al lenguaje.

Este formalismo fue introducido por Bryan Ford [1] en el 2007, aunque la mayor parte de la teoría formal existía anteriormente [2]. Desde entonces las PEG son adoptadas para la construcción de parsers para varios lenguajes de programación, como ser Phyton, C, C++, Java, Eiffel, Ruby, JavaScript, entre otros.

La gramática consiste en un conjunto de reglas de la forma 'A$\leftarrow e$' donde A es no-terminal y e es una parsing expression. En la tabla 1 se ven los operadores y predicados de las PEGs.

- Las comillas simples y dobles, delimitan los literales de una cadena, y los

Operador	Tipo	Prescedencia	Descripción
‘ ’	primario	5	Literal de un String
“ ”	primario	5	Literal de un String
[ ]	primario	5	Clase de un Caracter
.	primario	5	Cualquier caracter
(e)	primario	5	Agrupación
e?	sufijo unario	4	Opcional
e^*	sufijo unario	4	Cero o más
e^+	sufijo unario	4	Uno o más
& e	sufijo unario	3	Predicado and
!e	sufijo unario	3	Predicado not
e_1e_2	binario	2	Secuencia
e_1/e_2	binario	2	Alternativo con prioridad

Cuadro 1: Operadores de las PEGs y sus precedencias .

corchetes indican una clase de caracteres. Las clases de caracteres pueden ser o bien un conjunto conteniendo los elementos de esa clase, o un rango por ejemplo 1-9 o la constante ‘.’ .

- Los operadores ?, * y + se comportan de la misma manera que en las Expresiones Regulares, excepto que tiene evaluación *greedy* en vez de ser no-determinísticas. La expresión $e?$ incondicionalmente consume el texto correspondiente a e, si e no falla. Las expresiones de repetición e^* y e^+ siempre consumen la mayor cantidad de texto correspondiente a e como sea posible. Se debe tener cuidado con las expresiones de repetición, ya que por la forma de evaluación (greedy), por ejemplo la expresión a^*a nunca será exitosa.

- Los operadores & y ! denotan predicados sintácticos, los cuales proveen un alto poder expresivo de las PEGs. La expresión $\&e$ intenta corresponderse con e, luego hace backtracking al punto inicial, preservando únicamente el valor de falla o éxito, es decir no consume el caracter e. De manera opuesta sucede con la expresión $!e$, falla si tiene éxito y viceversa, pero sin consumir el caracter. Por ejemplo la expresión ‘!EndOfLine’ en la definición de Comment, se corresponde con cualquier caracter, siempre y cuando el no-terminal EndOfLine no busque corresponderse con algún caracter comenzando en la misma posición.
  En contraste, la expresión ‘Identifier !LEFTARROW’ en la definición de Primary se corresponde con cualquier identificador Identifier, que no esté seguido de LEFTARROW. Este último previene que la expresión del *right-hand-side* al comienzo de una definición, consuma el *left-hand-side* del Identifier de la próxima definición, eliminado la necesidad de un delimitador explícito. Algunos predicados pueden involucrar a otras expresiones en el momento de parsing pero esto se soluciona con los *look-ahead*

- La secuencia de la expresión e_1e_2, busca que se cumpla e_1 y luego e_2,

haciendo backtracking al punto de partida, si alguna de las dos expresiones falla.

2.1. Características

Scanner less. Permiten especificar parsers sin necesidad de dividir la parte léxica de la sintáctica.

Sin ambigüedad. Las alternativas en la gramática se procesan en el orden dado en la especificación, lo que se conoce como *prioritized choice*.

Predicados. Las especificaciones basadas en este formalismo son más compactas y los parsers aceptan una clase de lenguajes más amplia que la mayoría de los parsers clásicos determinísticos, como los basados en las gramáticas LL(k) o LR(k).

2.2. Propiedades de las PEGs

- Definen su sintaxis en términos de reconocimiento de cadenas, a diferencia de las BNF que están basadas en sistemas generativos.
- La descripción de una PEG puede verse como la descripción de un parser descendente recursivo con backtracking.
- Los predicados no consumen ningún caracter.
- No permiten directa o indirectamente reglas recursivas a izquierda.
- Son más poderosas que otras gramáticas pero consumen más memoria y podrían llegar a completar el proceso de parsing en tiempo exponencial en el peor caso.
- Sin embargo con técnicas de *memoization* las PEGs pueden ser parseadas en tiempo lineal.

2.3. Ejemplos

Algunos ejemplos de PEGs son:

- La siguiente Parsing Expression Grammar describe la cadena $a^n b^n c^n : n \geq 1$ (no expresable por las CFG).
  S ← & (A c) a+ B !(a/b/c)
  A ← a A? b
  B ← b B? c

- La siguiente regla recursiva se corresponde con el "if /then/else sentencia"de C. Debido a los *proiritized choice* o alternativos con prioridad, la cláusula "else"siempre se analizará primero que la "if"de más abajo:
  stmt ← "if" '('cond')' "then" stmt "else" stmt
  / "if" '('cond')' "then" stmt
  Cabe destacar que estas herramientas que eliminan la ambigüedad no hacen que la sintaxis del lenguaje sea sencilla, con los alternativos de las

CFGs, simplemente se deben elegir entre dos alternativas, en cambio con el *proiritized choice* de las PEGs, el diseñador del lenguaje debe tener en cuenta que las alternativas se elegirán de acuerdo al orden en que son colocadas, lo cual podría introducir errores si la gramática no esta bien diseñada. Un ejemplo de este problema puede ser si cambiamos el orden de prioridad del ejemplo anterior, lo que se conoce como captura de prefijos.
stmt ← "if" '('cond')' "then" stmt
"if" '('cond')' "then" stmt "else" stmt

2.4. Definicion Formal de una PEG

Una *Parsing Expression grammar* está definida formalmente como la 4-upla $G = (V_N, V_T, R, e_S)$, donde V_N es un conjunto finito de símbolos no-terminales, V_T es un conjunto finito de símbolos terminales, R es un conjunto finito de reglas y e_S es una *parsing expression* denominada expresión inicial.
Luego $V_N \cap V_T = \emptyset$. Cada regla r $\in$ R, es un par (A,e) el cual se escribe A$\leftarrow e$ donde A $\in V_N$ y e es una parsing expression. Para cualquier no-terminal A, existe exactamente un e tal que A$\leftarrow e \in$ R. R entonces es una función que va de un no-terminal a expresiones, y las escribimos R(A) para denotar la única expresión e tal que A$\leftarrow e \in$ R.

Definición Definimos entonces las *Parsing Expression* inductivamente como sigue: sean e, e_1 y e_2 parsing expressions:

1. ε es la cadena vacía.
2. a es cualquier terminal, donde $a \in V_T$
3. A es cualquier no-terminal, donde A$\in V_N$
4. $e_1 e_2$ es una secuencia.
5. e_1 / e_2 es un alternativo con prioridad o *prioritized choice.*
6. e^* simboliza cero o más repeticiones.
7. $!e$ es un predicado *not.*

3. Biblioteca EgLib

EgLib [4] es una biblioteca para el reconocimiento de lenguajes descrita en términos de Parsing Expression Grammars (PEG). La función principal de la biblioteca es la generación de parsers descendentes recursivos para reconocer gramáticas PEGs.

EgLib implementa un DSL interno, ya que permite escribir una gramática en C++ que se aproxime a la sintaxis de las PEGs, permitiendo que las especificaciones de las gramáticas PEGs se pueden mezclar libremente con código C++.

Además de la generación de parsers, otras funcionalidades de EgLib son generación de Árboles Abstractos Sintácticos; acciones Semánticas asociadas a los nodos del árbol; y proveer un mecanismo de Informe y Recuperación de errores.

Cada expresión PEG posee una clase dentro de EgLib y todas las clases en conjunto implementa un parser descendente recursivo con *backtracking*. Las gramáticas deben ser "bien formadas", como se describió en el capítulo 3, esto es, que cualquier procedimiento debe terminar bajo cualquier cadena de entrada si la gramática no posee recursión a izquierda, y las expresiones E^* y E^+ no están especificadas para cadenas de entrada vacías. Permite escribir una gramática en C++, aproximándose a la sintaxis de las PEGs. En la tabla 4 se comparan a las PEGs con las reglas que permite especificar EgLib.

Expresión EgLib	**PEG**
$e_1 >> e_2 >> ... >> e_n$	$e_1 e_2 ... e_n$
$e_1/e_2/.../e_n$	$e_1/e_2/.../e_n$
$^*e$	e^*
^+e	e^+
$maybe(e)$	$e?$
and_p(e)	& e
!e	!e
a_char('s')	's'
in("s")	$[s]$
in ('a', 's')	$[a-s]$
a_word(" s")	" s"
some_char()	.

Cuadro 2: Comparación entre PEG y EgLib.

Cada una de las expresiones son objetos que poseen un código C++ que puede ser ejecutado, por ejemplo la función Parse(...) hace el *parsing* de la clase que lo implementa.

Como también se observa, se hicieron algunas modificaciones con respecto a la sintaxis de las PEG, con el fin de utilizar los operadores de C++.

Notablemente vemos la abundancia de el operador de shifteo ($\gg$). Dado que no existe el operador 'vacío' en C++, no es posible escribir la expresión : **expr1 expr2** que en la sintaxis de PEG significa *expr1 seguida de expr2* .
Nuestra biblioteca en cambio usa el operador shift ($\gg$) para este propósito. Por lo tanto el operador ($\gg$) en nuestro contexto significa *seguido de*.

En el caso de la estrella de Kleene '*' la cual es usada como operador postfijo en las PEGs se convierte en operador prefijo, es decir : **(expr)*** en PEG, la escribimos en EgLib : ***(expr)** debido a que no existe el operador * postfijo en C++. Lo mismo ocurre con el operador '+'.

A continuación vemos la lista completa de las primitivas de parser:

- **Básicos :**
  a_char : Caracter Literal . Ejemplo : a_char('a');
  in : Clase Caracter . Ejemplo : in("abcde");

in : Rango Caracter . Ejemplo : in("a,z");
a_word : Cadena de literales . Ejemplo : a_word("Hola");

- **Parsers Predefinidos :**
  blanks : Espacios en blanco y saltos de línea.

- **Operadores :** Los operadores son usados como tales por los objetos. Mediante la sobrecarga de operadores, nos aproximamos a la sintaxis de las PEGs.

  1. *Operadores de Iteración :*
     ***** : Cero o más veces . Ejemplo: *a;
     + : Una o más veces . Ejemplo: +a;
     maybe : Cero o una vez. Ejemplo : maybe(a);
     El operador '?' no puede ser sobrecargado en C++, por tal motivo le cede el lugar al parser denominado maybe .
  2. *Operadores de Secuencia y Alternativa:*
     >> : Secuencia . Ejemplo: a >> b ;
     / : Alternativa con prioridad . Ejemplo: a / b ;
  3. *Predicados :*
     and_p : Predicado de conjunción .Ejemplo: and_p(a >> b);
     ! : Predicado de negación . Ejemplo: !a ;
  4. *Precedencia de Operadores :*
     Dado que definimos nuestro metalenguaje en C++, seguimos las reglas de precedencia y asociación de C++. Por ejemplo *(a >> b) se lee como *la expresión **a** seguida de la expresión **b**, cero o más veces.*

En el apéndice A.6 se muestra la representación de cada uno de los operadores mencionados anteriormente en un lenguaje de programación.

3.1. Diseño e Implementación

El diseño de esta implementación está basado en una jerarquía de clases de parsers de tres niveles y en la cual cada expresión PEG está representada por una clase parser específica.

Como se ve en la figura 1, en el primer nivel se encuentra la clase abstracta parser que declara métodos que serán definidos por cada clase concreta. En en el segundo nivel se encuentran todos los parser básicos, que son parsers concretos y dos parsers abstractos: los unarios (que representan expresiones unarias) y los compuestos (que representan secuencias de parsers y alternativos con prioridad). Por último, en el tercer nivel se encuentran los parsers concretos que heredan de los unarios y compuestos.

Para una mejor visualización de la relación entre las Parsing Expression Grammars y las clases Parsers de EgLib, podemos ver el cuadro 3 , que contiene las clases principales y expresiones de EgLib, junto con las PEG que representan.

En el apéndice A.1 vemos un ejemplo de una gramática en C++ utilizando la sintáxis de EgLib.

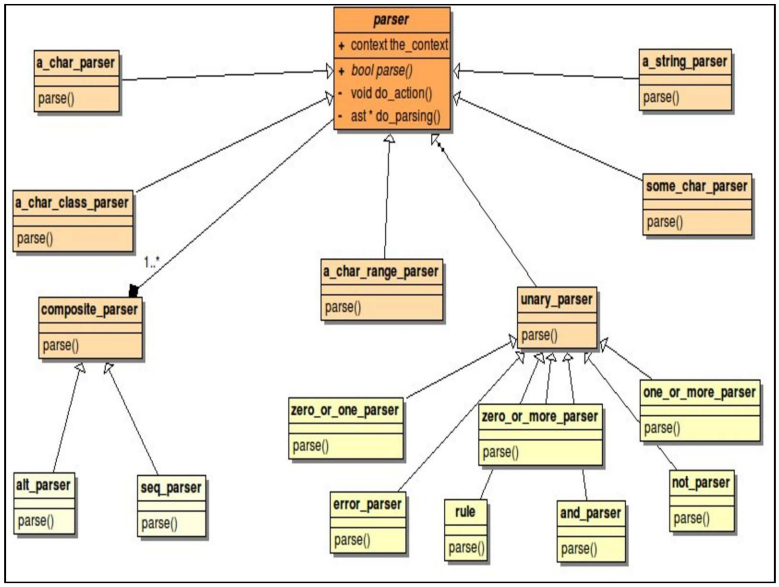

Figura 1: Jerarquía de clases

Clase EgLib	**Expresión EgLib**	**PEG**
alt_ parser	$e_1 >> e_2 >> ... >> e_n$	$e_1 e_2 ... e_n$
seq_ parser	$e_1/e_2/.../e_n$	$e_1/e_2/.../e_n$
zero_or_more_parser	$^*e$	e^*
one_or_more_parser	^+e	e^+
zero_or_one_parser	$maybe(e)$	$e?$
and_parser	and_p(e)	& e
not_parser	!e	!e
a_char_parser	a_char('s')	's'
a_char_class_parser	in("s")	$[s]$
a_char_range_parser	in ('a', 's')	$[a-s]$
a_string_parser	a_word(" s")	" s"
some_char_parser	some_char()	.

Cuadro 3: Comparación entre PEG y EgLib.

3.2. Generación de ASTs

Cuando se realiza el análisis sintáctico de algún dato de entrada, puede ser muy útil para un posterior análisis representar las características esenciales de la estructura de la cadena de entrada. EgLib permite realizar esta tarea mediante la generación de ASTs durante el parsing .

Cada regla de la gramática se corresponde con un nodo del árbol generado por el parser . A su vez, los nodos brindan la posibilidad de poseer información genérica a través del polimorfismo paramétrico.

El polimorfismo permite programar de manera general en lugar de manera específica. Una forma es crear herramientas de propósito general (clases, métodos) y especializarlas para situaciones específicas. Esta posibilidad es especialmente interesante a la hora de implementar estructuras de datos complejas que contienen elementos de otro tipo tales como pilas, arboles, etc.

En la librería EgLib es posible definir ASTs mediante polimorismo estático, permitiendo definir todas las clases que la componen genéricamente utilizando templates.

Las clases genéricas toman como argumento una clase definida por el usuario, aunque existe un parámetro de plantilla por defecto que representa una clase vacía, en caso que el usuario decida no aprovechar el polimorfismo.

La clase AST contiene un atributo genérico que se corresponderá con el tipo de la clase que el usuario defina, usando este atributo como un contenedor y guardando la información que desee.

El apéndice A.2 contiene un ejemplo concreto de una gramática y el AST generado para una cadena de entrada.

Cada vez que se instancia una clase o método genérico, el compilador crea una nueva clase o método, reemplazando los argumentos genéricos por los que indica el programador.

3.3. Acciones Semánticas

Las acciones semánticas son trozos de código que pueden asociarse a reglas gramaticales. Cuando el analizador de sintaxis determina que se ha utilizado una regla gramatical, ejecutará la acción semántica asociada.

En un compilador por ejemplo, se pueden utilizar acciones para construir una representación interna de la entrada de información, realizar verificaciones semánticas, emitir código, por citar algunas. El diseño de este módulo cuenta con una clase abstracta que posee únicamente la declaración del método eval() y cada acción concreta deberá implementarlo.

Como se vé, se implementa el patrón *Strategy*, el cual permite encapsular la lógica de un programa en una una clase para hacerla intercambiable. Cada acción es una *estrategia* que tendrá que sobrescribir el método eval() y proveer un algoritmo concreto para dicha estrategia.

Cómo se especifican las acciones semánticas?

El operador de indexación [ ] toma como parámetro una referencia a una instancia de la clase action. Luego asocia la acción al nodo del árbol correspondiente a la regla.

Si el parser tuvo éxito, ejecuta un método que se encarga de ejecutar las acciones semánticas. La forma en que se ejecutan las acciones implementan el patrón *Visitor*. Se van visitando los nodos del árbol y si tienen asociada una acción ejecutan el método eval() .

El apéndice A.3 contiene un ejemplo concreto de una gramática y una acción semántica asociada a una regla de la gramática.

3.4. Informe y Recuperación de errores

Normalmente no es aceptable que un programa termine ante un error de análisis sintáctico. Por ejemplo, un compilador debería recuperarse lo suficiente como para terminar de analizar el resto del archivo de entrada y comprobar sus errores; una calculadora debería aceptar otra expresión; en el caso de un parser, debería continuar analizando la mayor cantidad de caracteres de entrada, luego de receptar uno o varios errores.

EgLib también provee un mecanismo de informe y recuperación de errores. El mismo se invoca utilizando una regla predefinida error la cual deberá ser la última regla de un order choice. Recibe como parámetro dos parsers : un punto de recuperación y un look ahead . A continuación se describe cómo funciona este mecanismo:

- El objeto error recibe dos objetos parsers como parámetros. El primero para representar el punto de recuperación, es decir hasta dónde debe considerarse el error para luego continuar con el análisis y el segundo para el look-ahead, es decir, que reglas suceden al error, para no ser consideradas como tales .

- En el momento de su creación, la clase error crea un objeto de tipo error_parser . Sus atributos y métodos son útiles para la implementación del algoritmo que manejará la recuperación de errores.

- Como atributo fueron necesarios dos objetos parser para representar el punto de recuperación y el look-ahead. Por este motivo se encontró conveniente que la clase error_parser heredase de unary_parser , y así usar el objeto parser para el look-ahead, y otro objeto parser propio de la clase como punto de recuperación.

- El método parse () implementa el algoritmo de informe y recuperación, verificando primero que la regla que se está analizando no coincida con el look-ahead, para prevenir que no se esté tomando la regla como error por equivocación. En caso de pasar la verificación, el algoritmo ignora todos los caracteres de la cadena de entrada hasta encontrar el punto de recuperación, en ese momento informa la línea en que ocurrió el error y continúa con el parser. Luego de encontrar al menos un error, significa que el análisis ha fallado para esa cadena de entrada, por lo tanto no se construye el AST correspondiente, y el análisis falla .

El apéndice A.4 contiene el ejemplo de una gramática que utiliza este mecanismo y el output obtenido.

entrada EgParserGen	**salida EgParserGen**	**entrada EgLib**
$e_1e_2...e_n$	$e_1 >> e_2 >> ... >> e_n$	$e_1 >> e_2 >> ... >> e_n$
$e_1/e_2/.../e_n$	$e_1/e_2/.../e_n$	$e_1/e_2/.../e_n$
e^*	$^*e$	$^*e$
e^+	^+e	^+e
$e?$	$maybe(e)$	$maybe(e)$
& e	and_p(e)	and_p(e)
!e	!e	!e
's'	a_char('s')	a_char('s')
$\boxed{s}$	in("s")	in("s")
$\boxed{a-s}$	in ('a', 's')	in ('a', 's')
" s"	a_word(" s")	a_word(" s")
.	some_char()	some_char()

Cuadro 4: Comparación expresiones tanto de entrada como de salida de EgParserGen.

Cabe destacar que otras herramientas basadas en PEGs carecen mecanismos de informe y recuperación de errores.

4. Herramienta EgParserGen

EgParserGen es una herramienta de generación de parsers descendentes recursivos que procesa un lenguaje descrito en términos de Parsing Expression Grammars (PEGs) y genera un nuevo parser que reconoce las sentencias permitidas por dicha gramática.

Esta herramienta procesa las PEGs escritas con la sintaxis original descrita por Ford [2] y la reescribe en un parser en términos de EgLib, para que esta pueda procesarla. Para una mejor visualización de la relación entre EgParserGen y EgLib, se puede ver el cuadro 4, que contiene una comparativa de las expresiones de entrada y salida de EgParserGen y EgLib.

Debido a las propiedades de las PEGs, EgParserGen soporta backtracking ilimitado, combina el análisis sintáctico y semántico en una sola actividad y provee alternativas con prioridad como una manera de desambigüedad.

Al igual que EgLib, esta herramienta está implementada en el lenguaje de programación C++.

4.1. ¿Cómo funciona EgParserGen?

EgParserGen recibe como entrada una gramática PEG, la cual debe estar bien definida, es decir, debe estar escrita respetando la sintaxis original de las PEGs. Los datos de entrada se procesan a partir de un parser escrito en términos de EgLib, que describe la sintaxis completa de las PEGs incluyendo todas las características léxicas. En el cuadro 15 del apéndice A.7 se puede apreciar la gramática en términos de EgLib que conforma dicho parser.

Si el parsing es exitoso, EgParserGen toma el AST generado y al recorrerlo genera como salida un parser que representa a la gramática pasada como entrada, pero escrita en términos de EgLib.

Esta herramienta implementa un DSL externo ya que se debe escribir un parser para que procese una sintaxis específica.

En el apéndice A.5 se observa un ejemplo de cómo se utiliza esta herramienta.

4.2. Diseño e implementación

El diseño esta basado en dos clases, la primera es la clase donde se encuentra el parser escrito en términos de EgLib, que reconoce la sintaxis completa de las PEGs. La gramática de entrada es procesada por este parser y si resulta exitoso se genera el AST correspondiente, el cual representa la Expression Grammar especificada.

Luego la siguiente clase toma el AST generado, y mediante la invocación de una acción semántica se ejecutan distintas operaciones sobre este árbol para generar el nuevo parser de salida.

De esta forma EgParserGen funciona como un Visitor, ya que realiza un recorrido por una estructura de objetos, en este caso un AST, y cada vez que se visita un nodo, se invoca al método de visita definido, el cual genera el código correspondiente.

Por lo tanto es posible realizar distintas operaciones sobre los elementos de una estructura de objetos sin cambiar las clases de los elementos sobre los que se opera.

5. Comparativa con otras herramientas

En esta sección se compara EgLib con dos herramientas generadoras de parsers más: Spirit[5] y Rats[6].

En las tablas 5 y 6 vemos una comparación cualitativa de EgLib con Spirit y Rats respectivamente. Los parámetros de comparación son:

Tipo de Software: Biblioteca o Herramienta .

Funcionalidad: Cuál es el objetivo de cada uno de los software comparados.

Tipo de parser: Qué tipo de parser implementa la herramienta o biblioteca: LL, LR, descendente recursivo, etc.

Lenguaje de Programación: En qué lenguaje de programación esta implementado el software.

Gramática: Que tipo de gramática acepta el software en su especificación.

Scanner less: Si la especificación de la gramática es léxico-sintáctica.

Generación de AST: Provee la herramienta algún mecanismo de generación de AST.

Informe y Recuperación: Indica si provee algún mecanismo informe y recuperación de errores.

Acciones Semánticas: Indica si es posible la definición de acciones semánticas.

Módulos extra: Indica si posee módulos extras.

Memorización de Reglas: Indica se utiliza la técnica de memorización de re-

	Rats	**EgLib**
Tipo de Software	Herramienta	Biblioteca
Funcionalidad	Generación de parsers	Generación de parsers
Tipo de parser	Descendente Recursivo	Descendente Recursivo
L. de Programación.	Java	C++
Gramáticas	PEGs	PEGs
Scanner Less	Sí	Sí
Generación AST	Sí	Sí
Manejo de errores	No	Sí
Acciones Semánticas	Sí	Sí
Memoization de reglas	Sí	No

Cuadro 5: Comparación entre Rats y EgLib.

	Spirit Classic	**EgLib**
Tipo de Software	Biblioteca	Biblioteca
Funcionalidad	Generación de parsers	Generación de parsers
Tipo de parser	Descendente Recursivo	Descendente Recursivo
L. de Prog.	C++	C++
Gramáticas	EBNF	PEGs
Scaner less	No(Módulos Extra sí)	Sí
Generación de AST	No(Módulos Extra sí)	Sí
Manejo de errores	No	Sí
Acciones Semánticas	Sí	Sí
Módulos extras	Sí	No
Memorización de Reglas	No	No

Cuadro 6: Comparación entre Spirit y EgLib.

glas.

5.1. Perfomance

EgLib vs Spirit: Esta sección muestra una prueba de comparación de tiempos de ejecución, realizados a EgLib y Spirit. La gramática de prueba es la de expresiones aritméticas. Mediante el comando time de UNIX, pudimos obtener el tiempo real, de usuario y del sistema de cada generador de parser, como se muestra en el cuadro 7.

6. Conclusiones y Trabajos Futuros

6.1. Trabajos Futuros

Utilizar esta técnica para optimizaciones con respecto a la eficiencia.
Memoization una estrategia de parsing que permite solucionar el problema de

	Reglas	**Input**	**AST**	**T. Real**	**T. Usuario.**	**T.Sistema**;
EgLib	5	2.5 KB	No	0m0.024s	0m0.020s	0m0.000s
Spirit	5	2.5 KB	No	0m0.037s	0m0.016s	0m0.012s
EgLib	5	2.5 KB	Si	0m0.167s	0m0.040s	0m0.000s
Spirit	5	2.5 KB	Si	0m1.923s	0m1.904s	0m0.020s

Cuadro 7: Comparación de EgLib con Spirit- Gramática de expresiones aritméticas

tiempo de complejidad exponencial que poseen los parsers descendentes recursivos con backtracking. La idea básica de esta técnica es que cuando un parser es aplicado al *input*, el resultado es almacenado en una *memotabla* para reusarlo si el mismo parser es reaplicado al mismo input en el futuro.

Permitir al usuario la declaración de gramáticas recursivas a izquierda. Si el parser detecta que la gramática es recursiva a izquierda podría transformarla automáticamente para obtener una que sea recursiva a derecha.

Modificar la herramienta de tal forma que se puedan especificar en la sintaxis de la PEG de entrada, acciones semánticas y reglas de error.

Que el parser generado por EgParserGen no se restrinja a un único lenguaje (en este caso C++) sino que también sea posible generar código para distintos lenguajes (Java, C#, etc.).

6.2. Conclusiones

Dado que la biblioteca y la herramienta están escritas en términos de PEGs, nos ofrecen todas las ventajas propias de este tipo de gramáticas, entre las que se podrían destacar: las alternativas con prioridad que permiten la especificación de gramáticas libres de ambigüedades; los predicados permiten especificaciones más compactas y los parsers aceptan una clase de lenguajes más amplias que los parsers clásicos determinísiticos;la especificación de las gramáticas no requiere dividir la parte léxica de la sintáctica.

Si bien la biblioteca es mucho más simple que otras existentes (como por ejemplo Boost-Spirit), ofrece las funcionalidades más comúnmente usadas por los usuarios de generadores de parsers como ser: construcción del AST, informe y recuperación de errores y acciones semánticas.

A su vez, son pocas las herramientas que ofrecen el mecanismo de informe y recuperación de errores.

Por último, los resultados obtenidos con respecto a la performance de la biblioteca fueron los deseados.

Con respecto a EgParserGen, se destaca por la trasparencia de la herramienta con el usuario, ya que para generar un parser simplemente se debe escribir en un texto plano una Parsing Expression Grammar.

Appendices

A. Ejemplos de EgLib

A.1. EgLib - Ejemplo

La siguiente PEG que reconoce expresiones aritméticas :
Arit_Expr ← **Expr EOFile**
Expr ← **term ('+' Expr)?**
term ← **factor ('*' term)?**
factor ← **number**
/'(' Expr ')'
number ← **('-')? [0123456789]**
EOFile ← **!.**
puede ser expresada en C++ utilizando la sintaxis de EgLib como se vé en el cuandro 8 Para las distintas cadenas de entrada el parser devolverá éxito o falla dependiendo del input.

```
//Rule declarations
    rule Grammar, Expr, term, factor, number, EoFile ;
//Grammar body
    Grammar = Expr >> EOFile;
    Expr   = term >>  maybe( a_char('+') >>  Expr) ;
    term   = factor >> maybe(a_char('*') >>  term);
    factor = (number)
           /( a_char('(') >> Expr >> a_char(')'));
    number = maybe( a_char('-') ) >> +(in("0123456789"));
    EoFile  = !some_char();
```

Cuadro 8: Código C++: Definición de la gramática de expresiones aritméticas .

A.2. Generación de AST - Ejemplo

En el código C++ del cuadro 9 se muestra la definición de la gramática que reconoce expresiones aritméticas. Para la cadena de entrada : **1 + -24 * (3 + 45) #** , se obtiene el árbol de derivación de la figura 2.

A.3. Acciones Semánticas - Ejemplo

Se desea generar un parser para la gramática que acepta las cadenas : $a^n b^n c^n$ con n$\geq$ 1, y además calcular el valor de n mediante acciones semánticas.

Los cuadros 10 y 11 contienen respectivamente la definición de la gramática en C++ y la clase concreta counter que define en el método eval el algoritmo que calculará el valor de n. Por ejemplo para la cadena de entrada : **aaaaabbbbbccccc** la variable **cant** será igual a 5.

```
//Rule declarations
    rule Grammar, Expr, term, factor, number, EoFile ;
//Grammar body
    Grammar = Expr >> EOFile;
    Expr    = term >>  maybe( a_char('+') >>  Expr) ;
    term    = factor >> maybe(a_char('*') >>  term);
    factor = (number)
            /( a_char('(') >> Expr >> a_char(')'));
    number = maybe( a_char('-') ) >> +(in("0123456789"));
    EoFile  = a_char('#');
```

Cuadro 9: Código C++: Definición de una gramática.

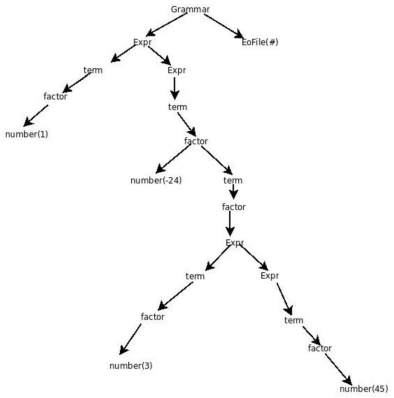

Figura 2: Árbol Abstracto Sintáctico

```
//Rule declarations
    rule S, A, B;
    counter cant_n;
//Grammar body
    S = S [&cant_n];
    S =  conjuntion(A >> a_char('c')) >> (+ a_char('a'))
         >> B >> (!(a_char('a')/a_char('b')/a_char('c')))
         >> !some_char();
    A = a_char('a') >> maybe(A) >> a_char('b');

    B = a_char('b') >> maybe(B) >> a_char('c');
```

Cuadro 10: Código C++: Definición de la de las gramática PEGs.

```
// Concrete class Example_Action

class counter : public action{
public:
    action(){}
    //concrete function eval
    void eval(stringstream input, ast *){
        cant = 0;
        for (list<ast *>::iterator it = t->get_begin_childs() ;
         it != t->get_end_childs();
         it++) {
            eval(input,*it);
            cant++;
        }
    }
    int cant;
}
```

Cuadro 11: Código C++: Definición de una acción.

A.4. Informe y Recuperación de errores - Ejemplo

En el cuadro 12 se especifica una gramática muy simple que acepta cadenas de entrada compuesta por palabras con letras de 'a' a la 'z', y permitiendo espacios de línea entre una o más palabras.
También se observa la regla error la cual toma como punto de recuperación a la regla NewLine, ya que si se encuentra algún error sintáctico este ejemplo ignorará todos los caracteres hasta el fin de línea y continuará con la siguiente línea; y como look-ahead toma la regla Punto, para darle a conocer al parser que en caso de que se produzca un error, el mismo deberá ocurrir antes que la reglaPunto sea correspondida con algún caracter.

```
//Rule declarations
    rule Sentence, Words, Literal, NewLine, Punto \\
//Grammar body
    Sentence = +( (Words >> Punto >> NewLine)
                / error(Punto,NewLine)) ;
    Words   = +(Literal);
    Literal = a_char_range ('a,z');
    NewLine = a_char('\n');
    Punto   = a_char('.') ;
```

Cuadro 12: Ejemplo de gramática 2

Por ejemplo para la siguiente cadena de entrada:

El experimentador1
que no sabe
lo que está buscando4
no comprenderá
lo que encuentra.

Se observa en color rojo los errores de sintaxis de la cadena de entrada, los cuales producirán los mensajes de error que se ve en el cuadro 13 al correr el parser.

line 1 error in substring : "**El experimentador1**" in rule Sentence fail trying parsing Words

line 3 error in substring :"**lo que está buscando4**" in rule Sentence fail trying parsing Words

Cuadro 13: Ejemplo de cadena de entrada

A.5. EgParserGen - Ejemplo

EgParserGen recibe un archivo como entrada que contiene la gramática a analizar. La PEG del ejemplo A.1, que reconoce expresiones aritméticas, es pasada como cadena de entrada en éste archivo, como se vé en el cuandro 14 .

Como salida, la herramienta genera dos archivos con el mismo nombre: un archivo de cabecera (.h) con la declaracion de métodos y variables, y el archivo de implementación (.cc) que contiene el parser que describe la misma gramática de entrada, pero ahora utilizando la sintaxis de EgLib, como se observó en el cuandro 8. Éste archivo luego será *linkeado* con la librería devolviendo éxito o falla dependiendo del input.

El usuario puede elegir el nombre de los archivos de salida pasándolo como una opción de compilación. Si el nombre se omite, por defecto los archivos de salida se llamarán *eg_parser_file*.

```
expression <- expr eol
eol        <- '#'
expr       <- term ('+' expr )?
term       <- factor ( '*' term)?
factor     <- number / '(' expr ')'
number     <- ('-')? [1234567890]+
```

Cuadro 14: Gramática de entrada de EgParserGen en texto plano.

A.6. Representación de las PEGs en un lenguaje de programación

En esta sección se presentan todos los operadores de las PEGs y su representación correspondiente en pseudocódigo, como muestran los algoritmos 1,2,3,4,5,6,7,8,9,10,11 y 12. Previamente se describen las funciones auxiliares utilizadas y su funcionalidad.

next() Esta función se encarga de tomar el próximo caracter a analizar en la cadena de entrada.

next(p) Esta función se encarga de tomar los p próximos caracteres a analizar en la cadena de entrada.

back(p) Esta función se posiciona al caracter p como caracter corriente.

Algorithm 1 Representación del operador 'c' en pseudocódigo.

```
bool parse () {
if 'c' == input.next()  then
   pos++;
   return  true;
else
   return  false;
end if}
```

Algorithm 2 Representación del operador "string" en pseudocódigo

```
bool parse () {
string = "string";
length = string.length();
if string == input.next(length) then
   pos + length;
   return  true;
else
   return  false;
end if}
```

Algorithm 3 Representación del operador $[s]$ en pseudocódigo

```
bool parse () {
if pos≥endpos then
   return  true;
end if
if not (exist (input.next(), "string") then
   return  false;
end if
pos++;
return  true; }
```

A.7. Especificación de las PEGs en EgLib.

En el cuadro 15 se muestra el parser escrito en términos de EgLib que reconoce la sitaxis completa de las PEGs.

Algorithm 4 Representación del operador $\left[c_1 - c_2\right]$ en pseudocódigo

```
bool parse () {
if not ( c1 ≤input.next()≤ c2) then
  return  false;
end if
pos++;
return  true; }
```

Algorithm 5 Representación del operador . en pseudocódigo

```
bool parse () {
pos++;
return  true; }
```

Algorithm 6 Representación del operador !E en pseudocódigo

```
bool parse () {
int p = pos;
if E then
  return  false;
end if
return  back(p) }
```

Algorithm 7 Representación del operador &E en pseudocódigo

```
bool parse () {
int p = pos;
if not ( Grammar) then
  return  false;
end if
return  back(p) }
```

Algorithm 8 Representación del operador E? en pseudocódigo

```
bool parse () {
E;
return  true; }
```

Algorithm 9 Representación del operador E* en pseudocódigo

```
bool parse () {
while E() do

end while
return  true; }
```

Algorithm 10 Representación del operador E^+ en pseudocódigo

```
bool parse () {
if not (E()) then
  return false;
end if
while E() do

end while
return true; }
```

Algorithm 11 Representación del operador $E_1/E_2/.../E_n$ en pseudocódigo

```
bool parse () {
if E1() then
  return true;
end if
if E2() then
  return true;
end if
.
.
.
if En() then
  return true;
end if
return false; }
```

Algorithm 12 Representación del operador $E_1E_2...E_n$ en pseudocódigo

```
bool parse () {
int p =pos;
if !E1() then
  return false;
end if
if !E2() then
  return back(p);
end if
.
.
.
if !En() then
  return back(p);
end if
return true; }
```

```
//Rule declarations
    rule Grammar, factor, term, number, Rule, Id, IdStart, IdCont,
    EOFile, LeftArrow, Expr, EOLine, Slash, Seq, Prefix, Suffix, And,
    Not, Quest, Star, Plus, Open, Close, Basic, Dot, Class, Range,
    Char, Literal, Spacing, Comment, Space, Line, OpenC, CloseC, Coma

// Grammar body
    Grammar = Grammar [&example_action];
    Grammar = Spacing >> +((Rule) / error(a_char('\n'), EOFile))  >>
              *(a_char('\n')) >> EOFile ;
    Rule    = Id >> LeftArrow  >> Expr;
    Expr    = (Seq >> (*(Slash >> Seq))) ;
    Seq     = +(Prefix);
    Prefix  = maybe(And / Not) >> Suffix;
    Suffix  = Basic >> (*(Star / Plus /Quest));
    Basic   = (Id >> !(LeftArrow))
            / (Open >> Expr >> Close)
            / (Literal)
            / (Class )
            / (Dot);
    Literal = (a_char('\'') >> *((!(a_char('\''))) >> Char) >>
               a_char('\'') >> Spacing)
            / (a_char('\"') >> *((!(a_char('\"'))) >> Char) >>
               a_char('\"') >> Spacing);
    Class   = a_char('[') >> +(!(a_char(']')) >> (Range) ) >>
              a_char(']') >> Spacing;
    Range   = ((Char >> Line >> Char))
            / ((Char));
    Char    = (a_char('\\') >>  in("n r t \" \' \\ [ ]"))
            / (a_char('\\') >> in('0','2') >> in('0','7') >>
               in('0','7'))
            / (a_char('\\') >> in('0','7') >> maybe(in('0','7')))
            / (!(a_char('\\')) >> some_char());
    Id  = IdStart >> *(IdCont)  >> Spacing;
    IdStart = in('a','z') / in('A','Z') / a_char('_');
    IdCont  = IdStart / in('0','9');
    Dot = a_char('.') >> Spacing;
    Slash   = a_char('/') >> Spacing;
    LeftArrow = a_word("<-") >> Spacing;
    Not     = a_char('!') >> Spacing;
    And     = a_char('&') >> Spacing;
    Quest   = a_char('?') >> Spacing;
    Star    = a_char('*') >> Spacing;
    Plus    = a_word("+") >> Spacing;
    Open    = a_char('(') >> Spacing;
    Close   = a_char(')') >> Spacing;
    Line    = a_char('-');
    OpenC   = a_char('[');
    CloseC  = a_char(']') ;
    Spacing =  *(Space / Comment);
    Comment = a_char('#') >> *(!(EOLine) >>some_char()) >>  EOLine;
    Space   = a_char(' ') / a_char('\t') / EOLine;
    EOLine  = a_word("\r\n") / a_char('\n') / a_char('\r');
    EOFile  = !some_char();
```

Cuadro 15: Código C++: Definición de la gramática que describe la sintaxis de las PEGs .

Referencias

[1] Brian Ford. *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation* POPL '04, ACM, January 2004. Massachusetts Institute of Technology Cambridge, MA

[2] Bryan Ford. *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking.* Master's Thesis. Massachusetts Institute of Technology Cambridge, MA.

[3] Roman R. Redziejowski *Parsing Expression Grammars as a Primitive Recursive-Descendent Parser with Backtracking* Fundamenta Informaticae 79, 3-4 (Sept 2007) 513–524.

[4] Betina Peredo *Parsing Expression Grammars. Desarrollo de EgLib.* Proyecto Final de la carrera Licenciatura en Ciencias de la Computación. Universidad Nacional de Río Cuarto (Abril 2010).

[5] Spirit. *Oficcial Spirit web site* `http://boost-spirit.com/`

[6] Rats. *Oficcial Rats web site* `http://cs.nyu.edu/rgrimm/xtc/rats.html`