DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

2 de Julio de 2014 — Teoría de lenguajes

Integrante	LU	Correo electrónico
Axel Iglesias	79/10	axeligl@gmail.com
Agustin Martínez Suñé	630/11	agusmartinez.92@gmail.com
Nahuel Lascano	476/11	laski.nahuel@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

El objetivo del Trabajo Práctico es crear un lenguaje de programación imperativo que permita definir funciones que serán evaluadas y nos permitirán graficar curvas paramétricas en el plano.

Para esto, tuvimos que decidir e implementar los siguientes puntos:

- La gramática que define el lenguaje especificado por la cátedra.
- El analizador léxico que recibe el código fuente y lo tokeniza.
- El parser que toma los tokens, verifica que coincida con la gramática y genera el Abstract Syntax Tree.
- El Abstract Syntax Tree que permite representar las funciones del código fuente y ejecutarlas.

Utilizamos Flex y BISON para implementar el lexer y el parser respectivamente. El lenguaje de programación utilizado fue C++.

2. Decisiones

2.1. BISON

Como se nombró en la introducción, para la implementación del parser decidimos usar BISON. Esto se debe a la documentación que encontramos para desarrollar el parser y a la aparente sencillez para usarlo.

2.2. Comentarios

Con respecto al uso de comentarios en el código fuente, para la primer entrega decidimos usar el comando *sed* para eliminarlos. Esta decisión fue mayormente motivada por ignorancia de cómo implementarlo en Flex o BISON.

Para esta segunda entrega, el lexer se encarga de detectar los comentarios. Es decir, es el lexer el que, al tokenizar, descarta los bloques de texto que son comentarios, con lo cual el parser ni siquiera debe tenerlos en cuenta.

2.3. Redondeo

Este fue un detalle el cual, en principio, no pudimos confirmar su procedencia: el error de redondeo que causaba nuestra implementación y provocaba leves diferencias en los gráficos.

En un comienzo, creímos que se debía al uso de Int's en vez de Float's, pero rápidamente confirmamos que no era esa la razón. Tampoco lo fueron los dígitos de Pi

Finalmente, usando la precisión estándar de *cout* se corrigieron los errores de redondeo.

2.4. Tipos de datos

Aunque no estaba especificado en el enunciado, decidimos que todas las funciones retornan *double*s, que a su vez, por simplicidad, es nuestro único tipo de datos para variables y operaciones aritméticas.

2.5. Ifs anidados

Para solucionar el conocido problema de ambigüedad de los If's anidados, decidimos verificar cuál es el comportamiento estándar de C++ e imitarlo.

Más detalles en la sección de Resultados.

3. Resultados

Para probar la funcionalidad de nuestro parser primogénito creamos los siguientes tests que abarcan las funcionalidades provistas por el lenguaje.

3.1. Test de funcionalidades básicas

El código se encuentra en testsPropios/testBueno1.cod

```
function id(x){
        return x
}

function fact(x){
        if x <= 0 then {
                return 1
        } else {
                res = 1
                while x!=0 {
                        res = x*res
                        x = x-1
                }
                return res
        }
}

plot (id(x), fact(x)) for x=1..1..6

/* Test de funcionalidades basicas:
        Definicion de funciones.
        Ciclos while.
        Condiciones ifthenelse.
        Variables locales.
        Operaciones aritmeticas basicas.
        Comparacion de expresiones.
        Varios puntos de retorno.
*/
```

Salida:

```
1 1
2 2
3 6
4 24
5 120
6 720
```

Como sus comentarios lo indican, este test permite probar algunas de las funcionalidades básicas computando la función de Fibonacci.

3.2. Test de más funcionalidades

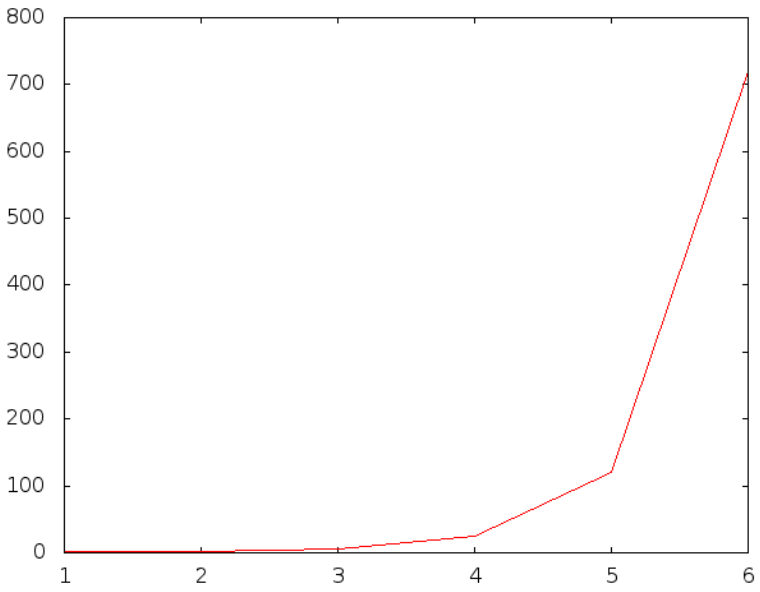
El código se encuentra en testsPropios/testBueno2.cod

```
function id(x){
        return x
}

function fact(x){
        if x <= 0 then {
                return 1
        } else {
                return id(x)*fact(x-1)
        }
}

plot (id(x), fact(x)) for x=1..1..6

/* Test de mas funcionalidades:
        Recursion.
        Llamados a funciones.
        Expresiones anidadas.
*/
```

Salida:

```
1 1
2 2
3 6
4 24
5 120
6 720
```

Ambos tests dan el siguiente gráfico, como era esperado:



3.3. Tests de desambiguación

El código se encuentra en testsPropios/testBueno3.cod

```
function dos(x){
        /* esta funci n deberia devolver siempre 2 */
        if 0 != 0 then
                if 1 == 1 then
                        return 0
                else    /* este else deberia asociarse (como indica
                        la indentacion) con el ultimo if, no con el
                        primero */
                        return 1
        return 2
}

function tres(x){
        /* esta funcion deberia devolver siempre 3 */
        return 35+dos(x)*-4**2
}


plot (dos(x), tres(x)) for x=1..1..6

/* Tests de desambiguacion:
         De precedencia de operaciones aritmeticas.
         De ifthenelse: resolucion est ndar al conocido conflicto shift/reduce.
   y correcto funcionamiento de la potencia (**).
*/
```

Salida:

```
2 3
2 3
2 3
2 3
2 3
2 3
```

En este ejemplo vemos cómo resolvimos el problema de la desambiguación de los ifthenelse anidados. Nuestro enfoque es el mismo que el útilizado en lenguajes de programación conocidos como C o C++. Podemos ver un ejemplo con el siguiente código:

```
#include <iostream>
using namespace std;

int test(){
        /* devuelve siempre 0... */
        if (0)
                if (1)
                        return 1;
                else    /* ...porque este else se asocia con el ultimo if */
                        return 2;
        return 0;
}

int main(){
        cout << test() << endl;
```

```
        return 0;
}
```

Para garantizar dicho funcionamiento no hizo falta modificar nada. Ante el conflicto Shift-Reduce BISON elige siempre Shift por defecto, con lo cual el resultado es el esperado.

Nota: El conflicto redunda en un Shift-Reduce porque nuestra gramática presenta una ambigüedad: al encontrar el segundo *else* como lookahead BISON no sabe si aplicar un Reduce (reducir el código del if más interno a un símbolo $IfThen$ y "asociar" luego el *else* al if externo) o un Shift (consumir el *else* con su bloque de código para después reducir el if lo más interno a un símbolo $IfThenElse$ y el más externo a un $IfThen$).

4. Gramática

El esquelto para hacer la gramática[1] fue tomado de la página citada en la presentación[2]. Sobre este esqueleto armamos nuestra gramática, que tiene varios puntos de diferencia con la que muestran de ejemplo en ese sitio.

Nuestra gramática parte de un símbolo incial *programa* que a su vez se divide en una lista de $funciones$ y una instrucción de *ploteo*.

Una función consiste en la palabra 'function' (representada por el token terminal TFUNC), un *nombre* (que no es más que un terminal que matchea con strings no empezados por números), una lista de $argumentos$ (lista de *nombre*s separadas por comas) entre paréntesis y un *bloque* de código.

Un *bloque* es una lista (quizás vacía) de sentencias encerradas entre llaves o una única sentencia.

Una *sentencia* puede ser una *asignacion* (un *nombre*, un símbolo de = y una *expresin*), una sentencia $ifthenelse$, un *while* (que a su vez tiene una *condicion* y otro *bloque*) o un $return$ (que lleva a su lado una *expresion*).

El caso del $ifthenelse$ requiere un análisis específico. Un $ifthenelse$ puede tener o no tener 'else', lo cual nos trae una ambiguedad a la hora de parsear cadenas del estilo (intencionalmente sin indentar):

```
if cond1 then
if cond2 then
codigo1
else
codigo2
```

que se refleja en el parser en un conflicto shift/reduce, como ya explicamos anteriormente (junto con la forma de resolución que elegimos). Es decir: nuestra gramática es ambigua.

Una *expresion* puede ser un *numero* (double o π), una $llamada_funcion$ (el nombre de una función con más *expresiones* entre paréntesis como argumentos) u operaciones aritméticas entre expresiones. Para salvar las ambiguedades propias de esta parte de la gramática le indicamos a BISON la precedencia de los operadores[3].

Una *condicion* puede ser una comparación entre dos *expresion*es o varias condiciones unidas con operadores lógicos.

Por último, la instrucción *ploteo* consiste en dos $llamada\_funcion$ con un *nombre* de variable y tres *expresion*es que reflejan el inicio, el paso y el final.

[1] implementada en parser.y

[2] http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/

[3] basándonos en el código de ejemplo de http://www-h.eng.cam.ac.uk/help/tpl/languages/flexbison/

La gramática se complementa con un archivo[4] con directivas para el tokenizador, que le permite convertir símbolos terminales en los tokens que usamos, ignora saltos de línea y maneja números enteros, floats y cadenas arbitrarias (como los nombres de variables).

Entre los problemas que encontramos hubo uno que nos presento dificultades: lograr armar la expresión regular para que el tokenizador ignore los comentarios multilínea. Finalmente gracias a los comentarios del corrector pudimos dar con la expresión regular acertada.

5. Respuestas

5.1. 1er Pregunta

Para verificar estáticamente que las condiciones de los if y los while sean booleanos se puede utilizar, en la gramática, un no terminal que represente únicamente operaciones booleanas (por ej comparación). De esta forma la gramática se asegura de que las condiciones sean booleanas y al momento de parsear el código se verifica el cumplimiento de ese requisito. De hecho, es así como lo implementamos nosotros.

En los lenguajes de programación tipados probablemente la gramática permita que las condiciones de los if no sean únicamente booleanos pero se verifica a la hora del chequeo de tipos (ya sea estático o dinámico).

5.2. 2da Pregunta

El uso del ; en la gramática de C permite desambiguar código. Veámoslo con un ejemplo:

```
int f(){
        return 42;
}

int main(){
        int x = 0;
        -f(); /* sin ';', esto se interpretaria como int x = 0-f() */
        return 0;
}
```

Este es el código de un ejemplo básico de C. Lo interesante es que, a diferencia de nuestro lenguaje, nos permite poner expresiones como si fueran instrucciones. Las llamadas a funciones en C/C++ son sentencias, a diferencia de nuestro lenguaje donde ninguna funcion tienen efectos secundarios.

Es por esto que si quitamos los ; del código de ejemplo nos queda una ambigüedad.

5.3. 3ra pregunta

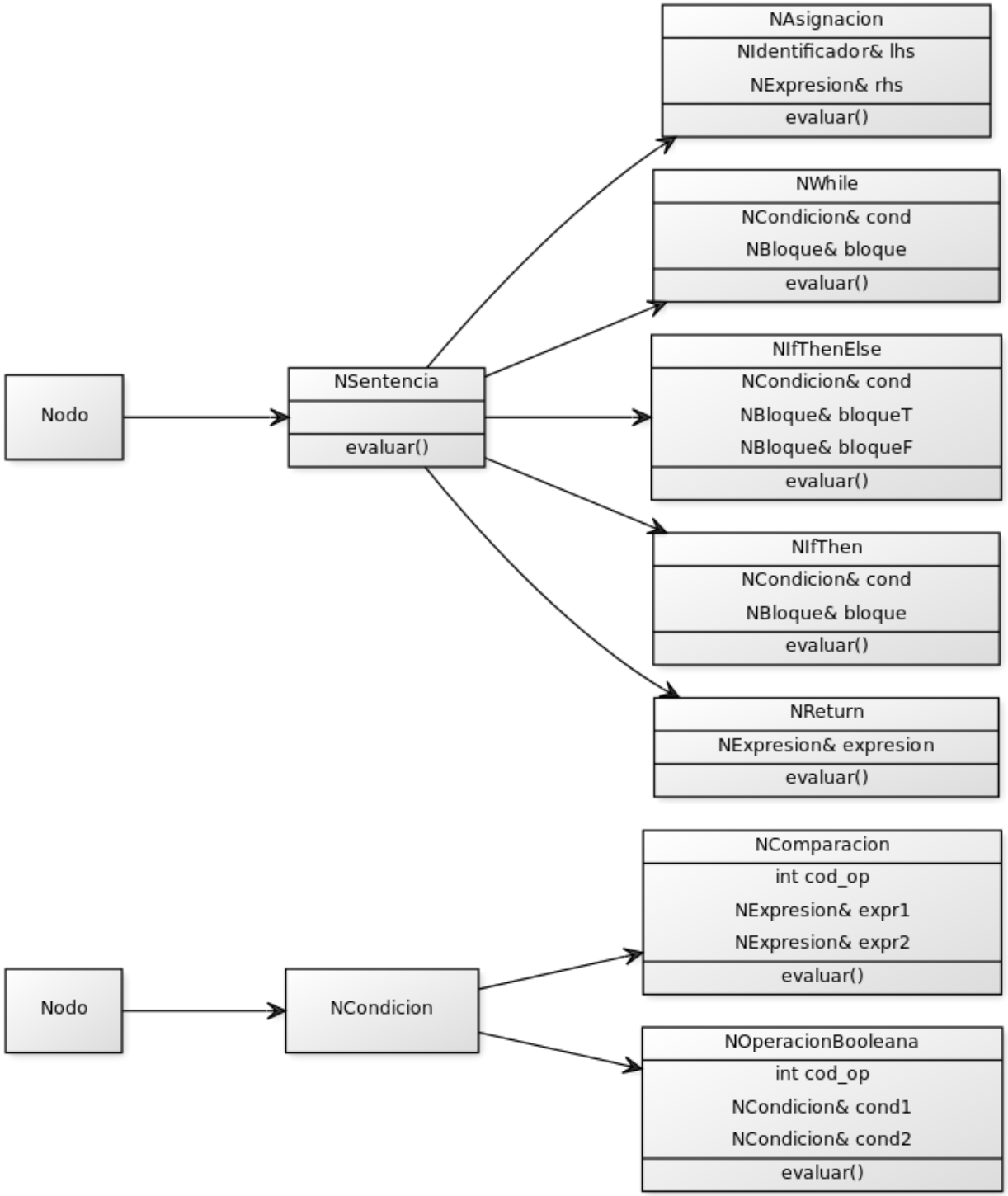
El orden de definición de las funciones solo importa si queremos verificar, en tiempo de parseo, que las llamadas a función invoquen a una función. Esto se debe a que, al encontrarnos con una llamada a función, solo conoceríamos las funciones que ya parseamos hasta ese momento. Una posible solución es hacer el parsing en dos pasadas, una para reconocer (y guardar en memoria) todos los nombres de función y otra para generar el arbol sintáctico. En nuestro caso, para resolver el problema, no hizo falta este enfoque, ya que verificamos en tiempo de ejecución que las llamadas a función invoquen a funciones válidas. Por otro lado, para soportar recursión se necesita tener un stack de variables locales por cada llamada a función, lo que permite que dos llamadas a la misma función incluso aunque estén anidadas sean independientes y tengan el comportamiento esperado. Es así como nosotros lo implementamos y por eso nuestro lenguaje soporta recursión.

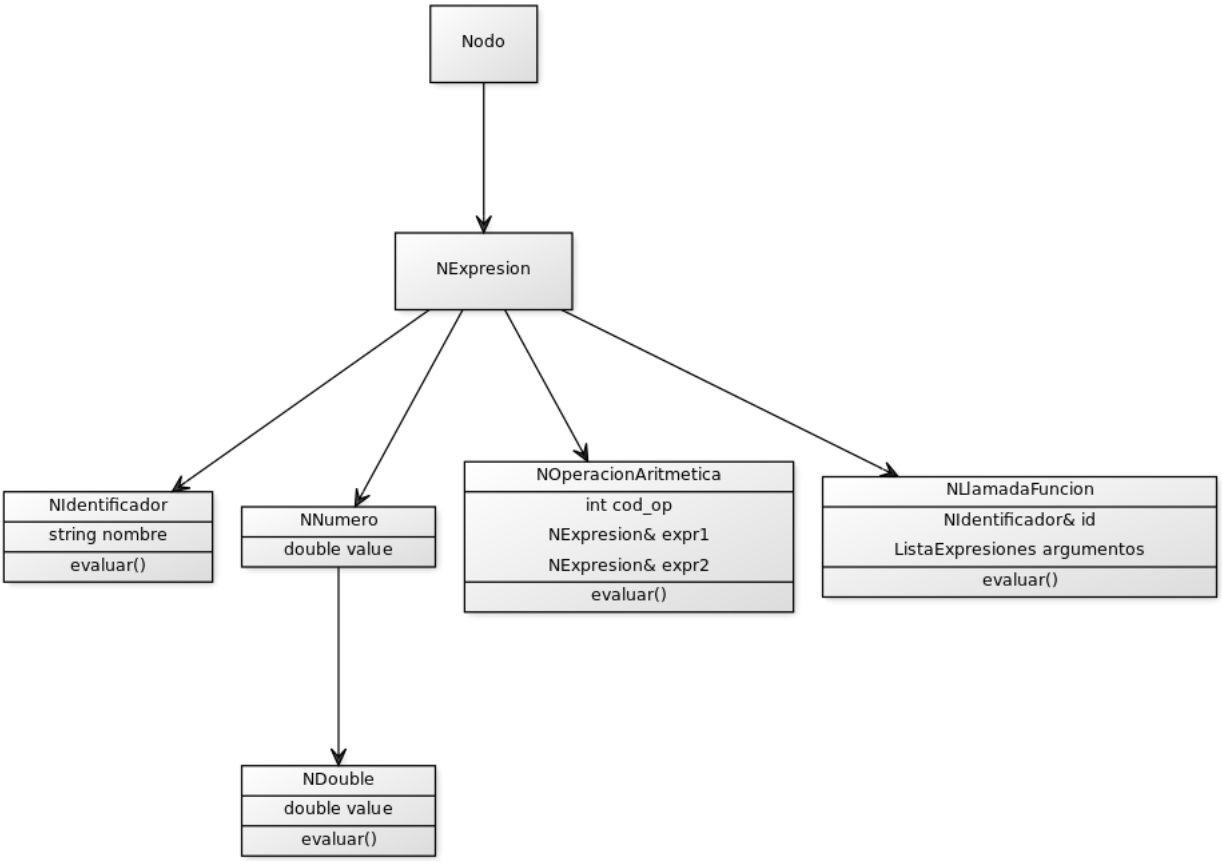
[4] tokens.l

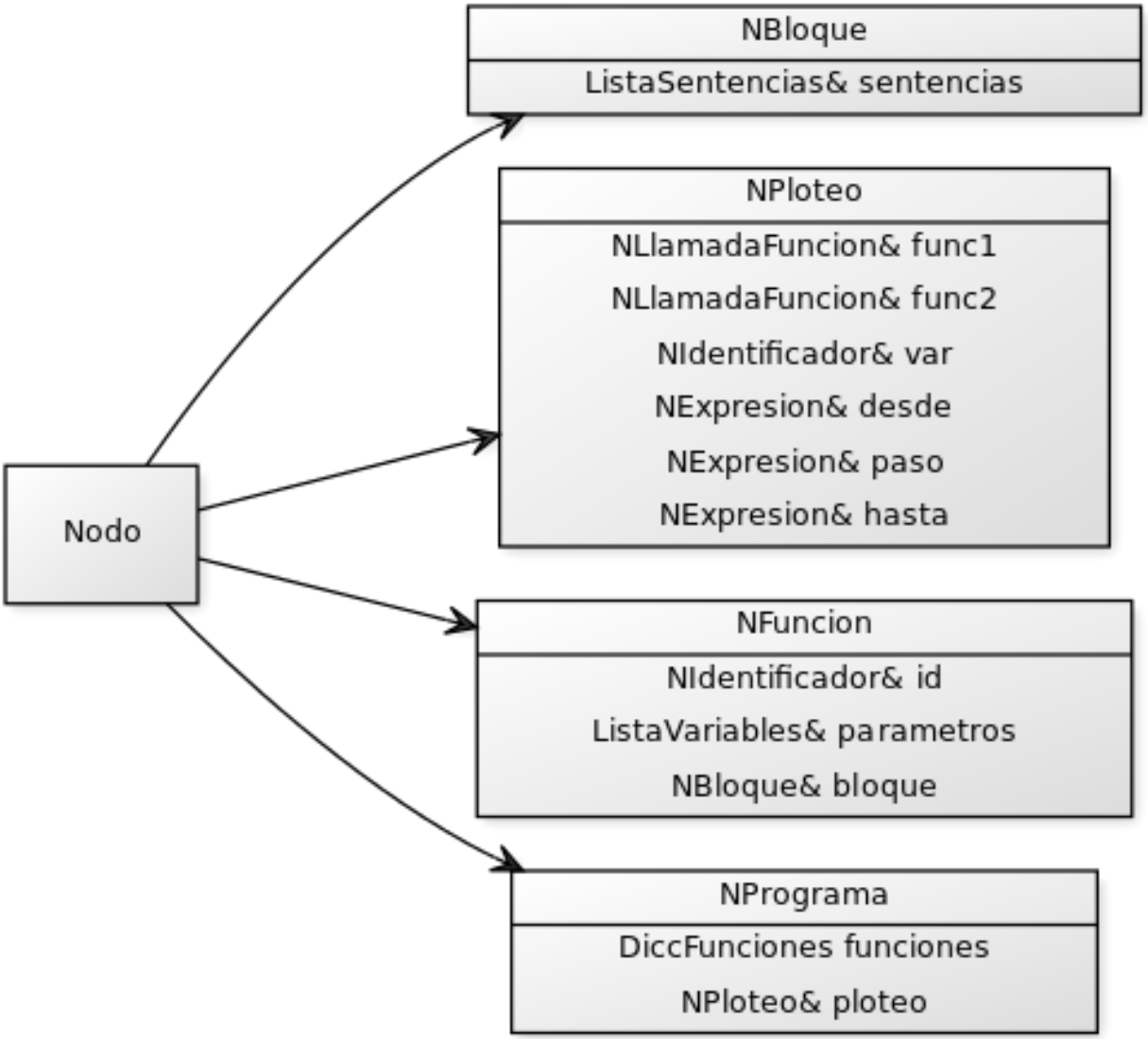
6. Manual

6.1. Diagrama de clases

Se muestra el diagrama de clases que compone al arbol sintáctico, cada clase se corresponde con un componente del lenguaje. Se repite la clase Nodo para facilitar la visualización.

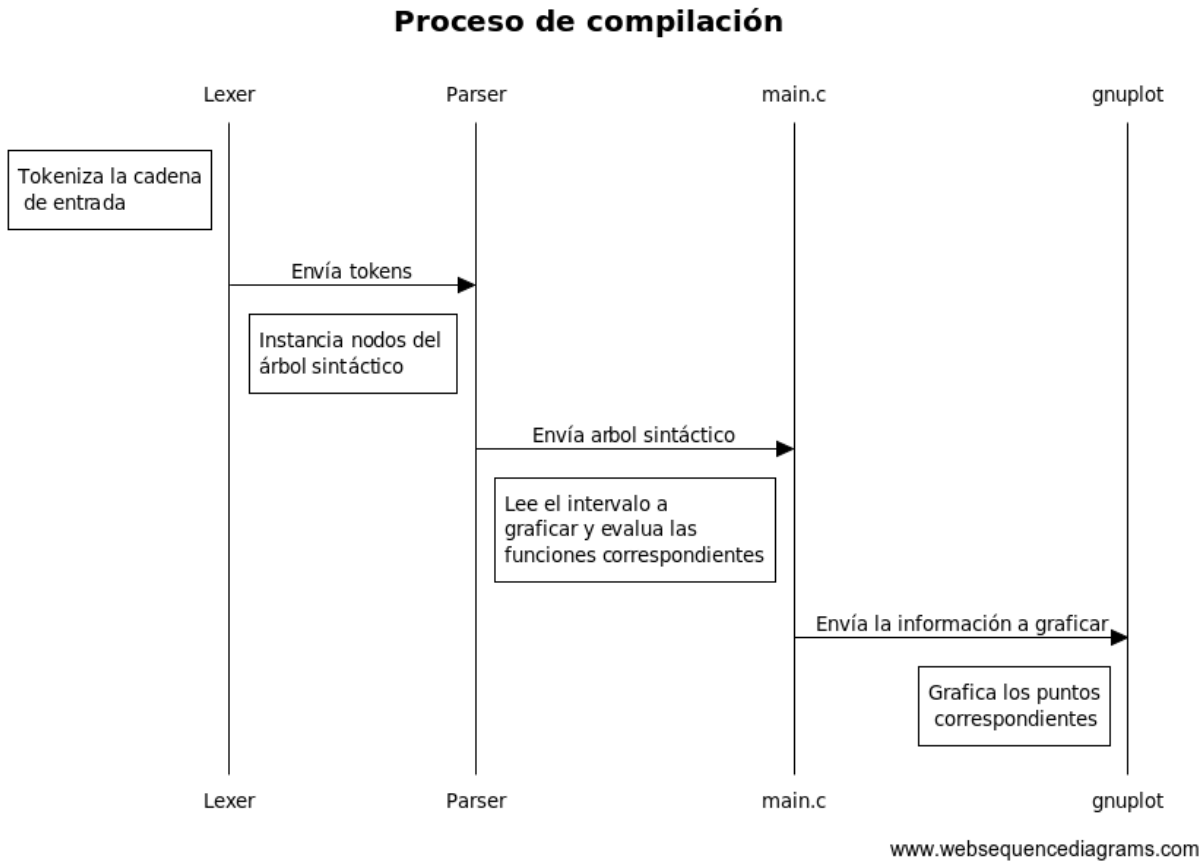



Nodo
NExpresion
NIdentificador
string nombre
evaluar()
NNumero
double value
NOperacionAritmetica
int cod_op
NExpresion& expr1
NExpresion& expr2
evaluar()
NLlamadaFuncion
NIdentificador& id
ListaExpresiones argumentos
evaluar()
NDouble
double value
evaluar()


NBloque
ListaSentencias& sentencias
NPloteo
NLlamadaFuncion& func1
NLlamadaFuncion& func2
NIdentificador& var
NExpresion& desde
NExpresion& paso
NExpresion& hasta
Nodo
NFuncion
NIdentificador& id
ListaVariables& parametros
NBloque& bloque
NPrograma
DiccFunciones funciones
NPloteo& ploteo

6.2. Diagrama de secuencia

Este diagrama muestra una ejecución típica de nuestro intérprete/graficador.



Modo de uso:

```
make clean && make
cat archivodetest.cod | ./parser | ./graficador.sh
```

7. Conclusiones

Como primer conclusión general podemos afirmar que el lenguaje propuesto por la cátedra es LR1, ya que pudimos construir un parser LR1 que lo reconoce. Además podemos decir que, a pesar de su simpleza, es un lenguaje potente que no solo permite el uso de las clásicas estructuras de control de flujo sino que también, como hemos visto, permite definir funciones de manera recursiva.

Por otro lado el desarrollo de la implementación de este lenguaje de programación nos permitió terminar de comprender las diferentes etapas que son inherentes a cualquier compilador o intérprete, desde el reconocimiento de la cadena de entrada y la generación del arbol sintáctico, hasta dar semántica a este.

En ese sentido un punto interesante a remarcar es la relativa independencia de estas etapas. Por ejemplo, si en lugar de interpretar el codigo, simularlo y generar una salida apta para graficar quisieramos traducir las funciones a codigo máquina para generar un binario ejecutable sólo deberíamos modificar, en principio, el código que se encarga de manejar el árbol sintáctico una vez construido.

Por último, también descubrimos y pudimos valorar la inmensa utilidad de tener otros lenguajes de programación (en nuestro caso C++) para poder parsear nuevos. Nos compadecemos de quien haya tenido que escribir el primer compilador.