# Autómatas, Teoría de Lenguajes y Compiladores

# Trabajo Práctico Especial 1C 2016

## Muffin Interpreter



**Integrantes**

* Lucas Casagrande 55302
* Nicolás Kuyumciyan 55165
* Magdalena Vega 55206
* Nicolás Castaño 53384

# índice

Consideraciones 3

Características del intérprete Muffin: 4

Descripción del desarrollo del TP 6

Descripción de la gramática. 7

Dificultades encontradas en el desarrollo del TP 8

Futuras extensiones, con una breve descripción de la complejida de cada una 9

Referencias. 11

# Consideraciones

* Es importante mencionar que más allá de que la consigna del trabajo practico indicara la realización de un compilador, se decidió implementar un intérprete, es decir un programa capaz de analizar y ejecutar otros programas, similar a un compilador, pero con la diferencia que no genera archivos adicionales (no se guarda el resultado de la traducción al código máquina). Consideramos que la esencia del compilador sigue presente, esto es la traducción de un lenguaje a otro que la computadora pueda entender, mediante la creación de una gramática definida y la adición de atributos que permiten a esa gramática tener comportamientos.
* Dicho intérprete recibe el nombre de “Muffin”. La decisión del mismo se basó en que resulta un nombre divertido, si se quiere atractivo desde el punto de vista comercial y, además, se le atribuye a uno de los integrantes del equipo: Magdalena (seria la traducción al castellano del nombre).
* Es de suma importancia mencionar que la función principal donde comienza la ejecución en todo programa no se llama main, sino que recibe el nombre el intérprete. Dicha función no recibe parámetros

int muffin(void) {

/\*Codigo\*/

return 0;

}

# Características del intérprete Muffin:

* El lenguaje cuenta con 4 tipos de datos:
  + Booleans, que se los determina bool en el lenguaje
  + Integer , que se denomina int
  + Double, que se denomina doublé
  + String, que se los puede denominar como str, Str, String o string.

Fue una decisión nuestra evitar el uso de built-in que tiene java, para mantener lo más simple posible el entendimiento del lenguaje.

* Cuando una variable de uno de estos tipos de datos se crea es posible asignarles el valor que se desee en la misma línea donde se declara: ( int i = 0; )
* Para mantener el código prolijo se restringió la posibilidad de poder declarar variables en cualquier lugar del código, sino que tiene que hacerse al principio de la función (ANSI standard).
* Permite crear Arrays de los tipos de datos previamente mencionados. A diferencia de las variables no se les puede asignar un valor en la línea de declaración. Para distinguirnos de otros lenguajes, se buscó que la declaración de los Arrays no resultara tan “engorrosa” (ejemplo en c: int numeros[ 10 ]). Se pensó lo siguiente: “Si tengo que declarar un arreglo de números enteros, lo más intuitivo seria pensar en una variable que tenga muchos valores de tipo int. Entonces, qué bueno sería que los arreglos se declaren como una variable, agregándole una ‘s’ al final para indicar que tiene muchos elementos.”. Es decir, los arreglos soportados por Muffin son: ints,doubles, bools, strings y sus variantes(Strings, strs, Strs, strings). De manera que no importa la dimensión del mismo, así como tampoco interesa en qué posición se asigna un elemento, ya que estos siempre se agregan al final. La asignación del mismo se realiza de la siguiente manera: *nombreArray* <- elemento (ejemplo: weapons <- “piedra” ;). Para consultar un valor en una posición del Array se utiliza la notación Nombre[Posición] como en la mayoría de los lenguajes.
* Cada línea de código debe finalizar con un ‘;’.
* Para imprimir una variable o texto se sustituyo el típico ‘print’ o ‘System.out.println’ por un simple “>>”.
* Para poder leer algo que escribe el usuario, en vez de usar un getChar() se reemplazó por un “<<”ß
* Se puede sumar un String con cualquier otro tipo de variable, donde el resultado es la concatenación de el String y la representación en String de la otra variable. Esto permite que se puedan hacer prints complejos.
* Operar entre valores enteros y punto flotante: suma, resta, multiplicación, división, calcular el resto y la potencia (mediante el operador \*\*).
* Se pueden utilizar valores negativos.
* Comparar valores enteros o punto flotante (inclusive entre ellos)
* Comparar valores booleanos: AND (o and o && ), OR (o || u or ), NOT (o not o !=). Se puede hacer de diferentes formas para aquellos que vienen de otros lenguajes, para que les resulte más fácil adaptarse y escribir código.
* Se puede comparar entre String. Un String se considera más chico que otro si en caso de ser ordenado alfabéticamente el menor se encuentra primero.
* Obtener el tamaño de un arreglo: $nombreArray
* Crear funciones:
  + Es importante mencionar que todas las funciones deben retornar un valor, y la directiva de return tiene que estar al final de la función.
  + Una función puede no recibir un parámetro, en caso de ser así es necesario poner void entre los paréntesis en la declaración y a la hora de llamarla.
* Generar bloques de repetición tipo while(condición){bloque}.
* Generar bloques de decisión tipo if(condición){bloque}else{bloque} o if(condición){bloque}
* Incrementar una variable entera (ejemplo: muffin++;).
* Decrementar una variable entera (ejemplo: muffin--;)
* Se pueden insertar comentarios en el código rodeándolos de los símbolos /\* y \*/

# Descripción del desarrollo del TP

Como primer paso se determinó cuales iban a ser los objetivos del lenguaje que desarrolláramos. La idea principal fue hacer un lenguaje sencillo que permita realizar cosas simples de una forma rápida sin preocuparse mucho sobre la sintáxis específica del lenguaje. Para lograr esto se decidió hacerlo lo más conciso posible, permitiendo, por ejemplo, imprimir en pantalla simplemente con el operador >> o leer del input con el operador <<. Esto también tiene la desventaja de que no es un lenguaje sumamente potente, pero no es su objetivo. En caso de querer hacer algo súper complejo, ya existen lenguajes que te brindan muchas herramientas para poder hacerlo. Pero por ejemplo para hacer un programa que lea un input y calcule si es primo o factorial en java es necesario varias líneas que en nuestro lenguaje se omiten (más de esto en la sección comparaciones).

Antes de comenzar la etapa de implementación del lenguaje, se tuvo que definir la gramática del mismo. Esto es, considerar todas las reglas que va tener que contemplar el intérprete a la hora de traducir el código. Dichas reglas serán definidas en la sección *Descripción de la gramática.*

Una vez definida la gramática, se establecieron los atributos, los cuales fueron especificados en el archivo Parser.cup. Allí también se especificaron nodos terminales y no terminales. De manera que cada regla del lenguaje se corresponde con una forma de construcción, la cual llama utiliza métodos correspondientes a clases de java previamente construidas.

# Descripción de la gramática.

La gramática se divide en 2 partes, la definición de dicha gramática y los atributos que le permiten no solo aceptar una cadena sino realizar operaciones

Primera Parte: Descripción de la gramática

Startp -> program

Program -> function program | 

Function -> type VAR ( params ) { body RETURN expr ; }

| type VAR ( void ) { body RETURN expr; }

params -> type VAR , params | type VAR

body -> varlist stmtList

varlist -> varlist dec | varlist arrayDec | 

dec -> type VAR = expr ;

arrayDec -> type VAR ;

stmtList -> stmtList statements | 

statement -> ; | VAR = expr; | VAR [ expr ] = expr ; | VAR <- expr; | VAR ++; | VAR --;

| << VAR; | >> VAR; | IF ( expr) { stmtList } | IF ( expr) {stmtList} ELSE {stmtList}

| WHILE ( expr) {stmtList } | VAR ( expr\_list) | VAR ( void)

expr\_list -> expr | expr , expr\_list

expr -> NUMBER | DOUBLE | BOOLEAN | VAR [ expr ]| VAR | - expr | expr op expr

| ( expr ) | $ expr | LIT | RAND ( expr ) | TRUE | FALSE | expr compare expr

| expr logic expr | NOT expr | VAR ( expr\_list ) | VAR ( VOID )

logic -> AND | OR

compare -> < | > | <= | >= | == | !=

type -> INT | STR | BOOL | DOUBLE | AINT | ASTR | ABOOL | ADOUB

op -> + | - | \* | % | \*\* | /

2da Parte: Agregándole los atributos:

Una vez definida la gramática es necesario darle atributos para que sea capaz de correr el código. La idea atrás de esto es que todos los no terminales son un objeto.

Idea general:

•Expr devuelve un objeto que al ejecutarlo devuelve el resultado de realizar la operación correspondiente a cada una de las posibles expresiones.

•Un Statement también es un objeto que se ejecuta, que puede contener diferentes expresiones como por ejemplo una asignación de una variable.

•Un body contiene una lista de Statements y variables. Al momento de ejecutar un body (por ejemplo, porque se entró en un if que adentro tiene instrucciones) lo que hace es ir ejecutando cada uno de los statements que tiene contenido.

•Function es un objeto que contiene un body que va a ejecutar, además de esto también tiene la lista de variables que le pasan cuando se llama a una función.

•El program tiene una lista con todas las funciones que contiene el programa y al momento de ejecutarse busca la función muffin para empezar a ejecutar el programa.

Dificultades encontradas en el desarrollo del TP

La única dificultad que fue encontrada fue que en varios casos, lo que noostors queríamos agregar a la gramática generaba conflictos. Y había que reordenar las sentencias para que funcionara.

# Futuras extensiones, con una breve descripción de la complejida de cada una

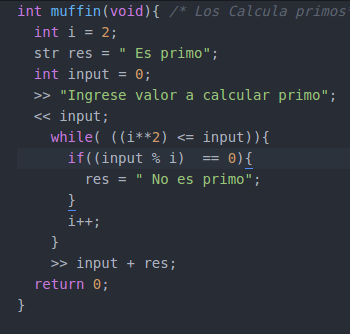
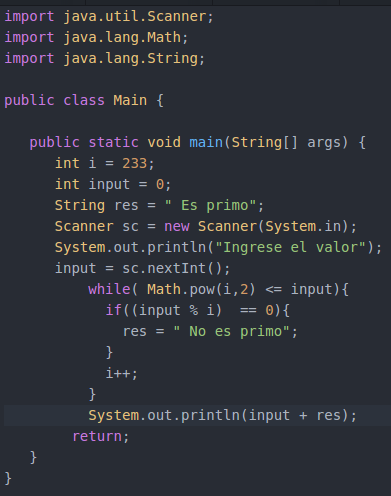
En un futuro se podría agregar al lenguaje:

* Generar bloques de repetición del tipo for, do-while.
* Generar bloques de decisión del tipo if () {} else if () {}…. Else {} , o switch(case).
* Agregar otro tipo de tipo de datos, ya sea float, char, etc. .
* Poder tener arreglos de arreglos. Ejemplo stringss sería una matriz de strings.
* Declarar variables al comienzo de los bloques if y while.
* Estructuras.

Estas ideas son cosas se que podrían haber llevado a cabo pero por una cuestión de limitación en los plazos de entrega no se nos hizo posible implementarlo.

Comparación con otro lenguaje

Como describimos arriba el objetivo del lenguaje era poder minimizar la cantidad de código que el programador tiene que escribir cuando lo que él desea es hacer algo sencillo. Esto se pone de manifiesto en este sencillo programa que calcula si un número es primo, donde se puede ver que el código en java no solo es más largo, sino que las instrucciones son más verborragicas.



# Referencias.

* http://www2.cs.tum.edu/projects/cup/examples.php