

ProgramAR

72.39 - Autómatas, Teoría de Lenguajes y Compiladores Segundo Cuatrimestre 2021

Integrantes:

- Julián Francisco **Arce**, 60509
 - Gastón **De Shant**, 60755
 - Paula Andrea **Domingues**, 60148
 - Gian Luca **Pecile**, 59235
-

Índice

Idea subyacente y objetivo del lenguaje	2
Consideraciones realizadas	2
Benchmarking	2
Desarrollo	3
Gramática	4
Tipos de Datos	7
Delimitadores	7
Operadores aritméticos	7
Operadores relacionales	8
Operadores lógicos	8
Operadores de asignación	8
Operadores booleanos	8
Bloque Condicional	8
Bloque Do-While	9
Entrada estándar	9
Salida estándar	9
Comentarios	9
Constantes	9
Separadores de contexto	9
Dificultades encontradas	9
Posibles extensiones	10
Tests	11
test1.ar	11
test2.ar	11
test3.ar	11
test4.ar	11
test5.ar y test5.c	12
test6.ar	12
test7.ar	12
Bibliografía y referencias	12

Idea subyacente y objetivo del lenguaje

La idea subyacente es crear un lenguaje que sea en español y, a su vez, tenga como objetivo ser lo más didáctico posible. El enfoque está en que sea sencillo de usar por chicos de escuelas primarias/secundarias al igual que gente sin conocimiento previo sobre programación de cualquier edad, donde no sea necesario lidiar con el proceso de aprender un lenguaje en específico, sino aprender a resolver problemas de programación y lógica. Para ello, proponemos el lenguaje "ProgramAR".

En primer instancia esa fue la idea detrás del lenguaje y al consultar con la cátedra nos encontramos que otro grupo había elegido una idea similar y decidimos proponer una estructura para "forzar" ciertas buenas prácticas en cuanto a estilo de código, manteniendo el fin didáctico y el proponer que los programas escritos en nuestro lenguaje posean buenas prácticas; en el sentido de siempre definir primero variables, luego trabajar y ejercitar esas variables, por último, mostrar adecuadamente los resultados. Dicha estructura se asemeja a la que posee un test unitario, donde primero se setean las precondiciones, luego se ejercita el método a testear y luego se validan resultados; así se puede generar una organización del código bien diferenciada y seccionada para que facilite la lectura del mismo.

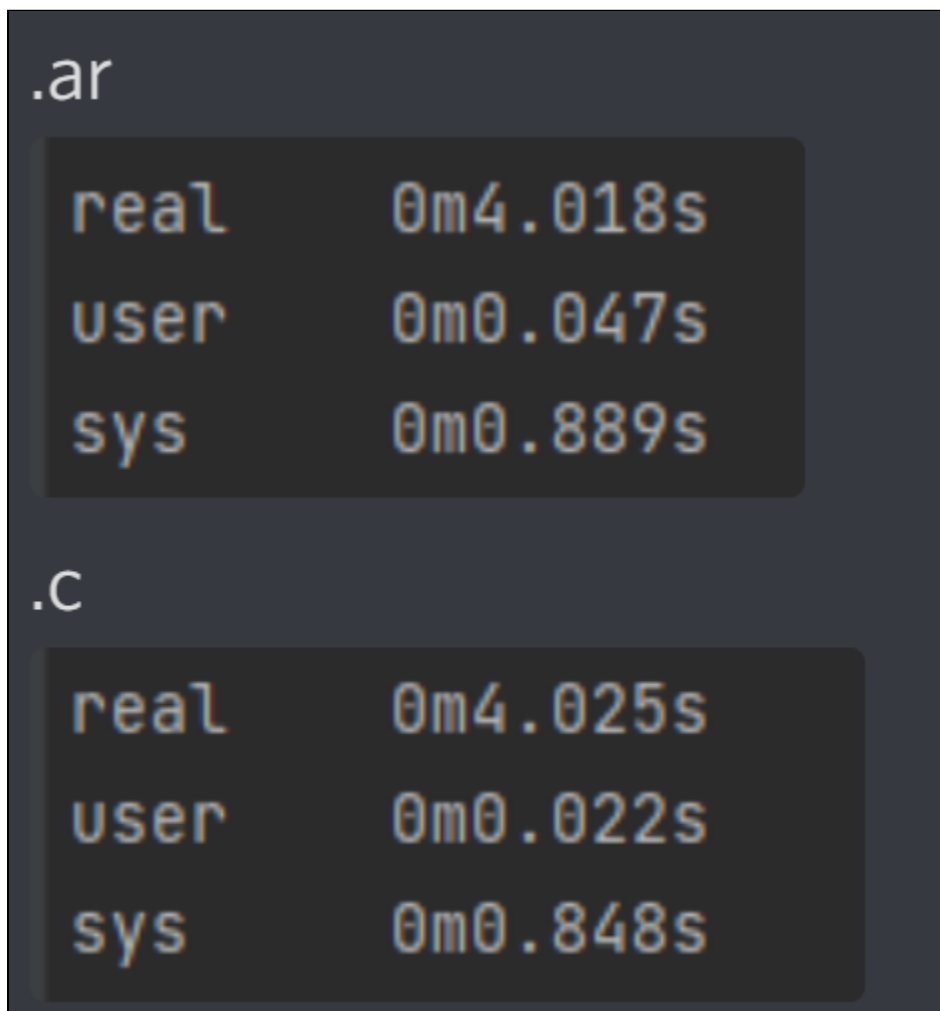
Consideraciones realizadas

El lenguaje ProgramAR cumple con lo pedido por el enunciado dado por la cátedra y fue probado de manera extensiva localmente al igual que en pampero para validar el funcionamiento correcto.

Benchmarking

Se agrega como consideración el benchmarking de un test realizado tanto en C como con ProgramAR en el cual se hace uso de syscalls las cuales generan

demoras significativas en C (refiere el *test5*). Los resultados utilizando el comando *time* es el siguiente:



The image shows a terminal window with two sections of benchmarking results. The first section is for a file named '.ar' and the second is for a file named '.C'. Each section displays three lines of output: 'real', 'user', and 'sys', followed by their respective time values in seconds.

.ar	
real	0m4.018s
user	0m0.047s
sys	0m0.889s

.C	
real	0m4.025s
user	0m0.022s
sys	0m0.848s

Imagen 1: Benchmarking realizado para el mismo programa en el lenguaje C y como programa realizado con el lenguaje ProgramAR.

Desarrollo

Para el desarrollo del lenguaje ProgramAR se hizo uso de Yacc y Lex, herramientas vistas en clase y provistas por la cátedra con más información sobre la documentación y referencias utilizadas en dicha [sección](#).

En primer lugar se definieron las palabras propias del lenguaje y se creó el archivo *lex.l*. Luego se creó el archivo de sintaxis llamado *syntax.y*, donde se

encuentran definidos los diferentes terminales finales y no finales, al igual que generar el analizador sintáctico para nuestro lenguaje. Para almacenar dichas variables usadas al igual que sus nombres, se hace uso de una implementación de una estructura de datos, en específico una lista encadenada, dónde se guarda el nombre de cada variable y su tipo (entero o texto) además del puntero al siguiente. La salida es en lenguaje C, compilado con gcc. Se puede notar que la extensión de los archivos propios del lenguaje ProgramAR poseen la extensión `.ar` debido al énfasis en el lenguaje en español que usa jerga proveniente Argentina. Para ilustrar mejor la compilación se presenta el siguiente diagrama de flujo:

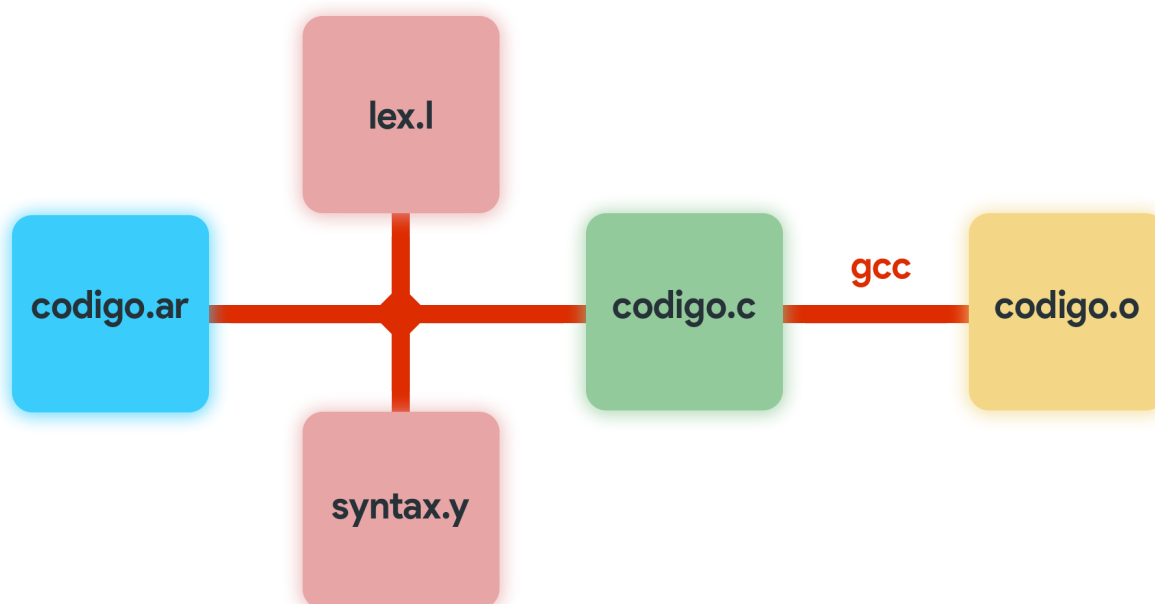


Figura 1: flujo compilación de archivos realizados con el lenguaje ProgramAR.

Todo el desarrollo se realizó mediante el uso de git, a través del [repositorio localizado en github](#) donde se encuentra este informe.

Gramática

Definimos nuestra gramática G acorde a lo visto durante la cursada como:

$$G = \langle NT, T, S, P \rangle$$

Dónde se tiene que **NT** representa el conjunto:

```
{ inicio, final, declaraciones, declarar, decl, rutina, rutr,
instruccion, impr, declaracion, declaracion_nombre_string, nombre_st,
sentencia_booleana, sentencia_not, sentencia_logica, operacion,
parentesis_st_abre, parentesis_st_cierra, boolean,
sentencia_comparativa, comparador, operador, valor, control_logico,
super_si, super_si_sino, si_st, entonces, fin_si, si_no_st,
super_hacer, hacer_st, fin_hacer, mientras_st, fin_mientras,
asignacion, asignacion_numero, asignacion_texto, asignacion_st,
texto_st, declaracion_y_asignacion, print, imprimir_pabr, comentario,
nombre_const_st, read }
```

Dónde **T** representa el conjunto:

```
{ FIN_LINEA, VAR_NUMERO, VAR_STRING, CONST, MAS, MENOS, POR,
DIVIDIDO, MOD, VERDADERO, FALSO, MENOR, MAYOR, MENOR_IGUAL,
MAYOR_IGUAL, IGUAL, DISTINTO, Y, O, NO, ASIGNACION, PARENTESIS_ABRE,
PARENTESIS_CIERRA, LLAVE_ABRE, LLAVE_CIERRA, COMILLA, SI, SI_NO ,
HACER ,MIENTRAS,LEER ,IMPRIMIR ,TEXTO ,NUMERO ,NOMBRE, CODIGO,
COMENTARIO }
```

Por último, para **P** se tiene el siguiente conjunto:

```
{
S → define const inicio declaraciones rutina final
inicio → CODIGO
final →
declaraciones → , declarar
declarar → decl FIN_LINEA, decl FIN_LINEA declarar
decl → declaracion {}, declaracion_y_asignacion {}
rutina → , rutr
rutr → instruccion FIN_LINEA, instruccion FIN_LINEA rutr,
comentario, comentario rutr, control_logico, control_logico rutr,
read, read rutr, impr, impr rutr
instruccion → asignacion {}
```

```
impr → print FIN_LINEA
declaracion → declaracion_nombre_string
declaracion_nombre_string → VAR_NUMERO NOMBRE, VAR_STRING NOMBRE
nombre_st → NOMBRE
sentencia_booleana → boolean, boolean sentencia_logica boolean,
parentesis_st_abre sentencia_booleana parentesis_st_cierra
sentencia_logica sentencia_booleana, boolean sentencia_logica
parentesis_st_abre sentencia_booleana parentesis_st_cierra,
sentencia_not parentesis_st_abre sentencia_booleana
parentesis_st_cierra, sentencia_not boolean. parentesis_st_abre
sentencia_booleana parentesis_st_cierra. sentencia_comparativa
sentencia_not → NO, NO sentencia_not
sentencia_logica → Y, O
operacion → valor operador valor{}
parentesis_st_abre → PARENTESIS_ABRE
parentesis_st_cierra → PARENTESIS_CIERRA
boolean → VERDADERO, FALSO
sentencia_comparativa → valor comparador valor
comparador → MENOR, MAYOR, MAYOR_IGUAL, MENOR_IGUAL, IGUAL, DISTINTO
operador → MAS, MENOS, POR, DIVIDIDO, MOD
valor → nombre_st, NUMERO, parentesis_st_abre operacion
parentesis_st_cierra, nombre_const_st
control_logico → super_si, super_si_sino, super_hacer
super_si → si_st sentencia_booleana entonces rutina fin_si
super_si_sino → si_st sentencia_booleana entonces rutina si_no_st
rutina fin_si
si_st → SI PARENTESIS_ABRE
entonces → PARENTESIS_CIERRA LLAVE_ABRE
fin_si → LLAVE_CIERRA
si_no_st → LLAVE_CIERRA SI_NO LLAVE_ABRE
super_hacer → hacer_st rutina fin_hacer mientras_st
sentencia_booleana fin_mientras
hacer_st → HACER LLAVE_ABRE
fin_hacer → LLAVE_CIERRA
mientras_st → MIENTRAS PARENTESIS_ABRE
fin_mientras → parentesis_st_cierra FIN_LINEA
asignacion → nombre_st asignacion_numero, nombre_st asignacion_texto
asignacion_numero → asignacion_st valor
asignacion_texto → asignacion_st texto_st
```

```
asignacion_st → ASIGNACION
nombre_const_st → CONST NOMBRE
texto_st → TEXTO
declaracion_y_asignacion → declaracion_nombre_string
asignacion_texto | declaracion_nombre_string asignacion_numero
print → imprimir_pabr TEXTO PARENTESIS_CIERRA, imprimir_pabr NOMBRE
PARENTESIS_CIERRA, imprimir_pabr CONST NOMBRE PARENTESIS_CIERRA
imprimir_pabr → IMPRIMIR PARENTESIS_ABRE
comentario → COMENTARIO
read → LEER PARENTESIS_ABRE NOMBRE PARENTESIS_CIERRA FIN_LINEA
}
```

La gramática detrás del lenguaje ProgramAR se detalla de la siguiente manera:

Tipos de Datos

- texto
 - Representa un string que se usa en lenguajes como Java.
- letra
 - Representa el tipo de dato char en lenguaje C.
- numero
 - Representa un entero.

Delimitadores

- ;
 - Actúa como indicador de fin de línea.
- ()
- { }
- " "

Operadores aritméticos

- +

- -
- *
- /
- modulo

Operadores relacionales

- vale menos que
- vale mas que
- es igual o vale menos que
- es igual o vale mas que
- es igual a
- es distinto de

Operadores lógicos

- y
- o
- opuesto de

Operadores de asignación

- vale

Operadores booleanos

- verdadero
- falso

Bloque Condicional

- si se cumple(condición) { // código }
- } si no { // código }

Bloque Do-While

- `hacer { // código } mientras(condición);`

Entrada estándar

- `leer(variable);`

Salida estándar

- `imprimir(variable);`

Comentarios

- `#comentario#`

Constantes

- Se declaran previo al separador de contexto código.

Separadores de contexto

- `constantes { // declaraciones }`
 - Se declaran las constantes.
- `codigo { // código }`
 - Se realizan operaciones lógicas, aritméticas, entre otras y se imprimen valores.

Dificultades encontradas

La principal dificultad encontrada fue la falta de conocimiento al respecto de tanto Lex como Yacc. Se recurrió a las clases dadas por la cátedra al igual que la bibliografía y los manuales disponibles online que se encuentran en la sección de [bibliografía y referencias](#).

Los operadores lógicos funcionan de manera correcta dependiendo de su sintaxis. Por ejemplo:

```
( 1 == 2 ) || true
```

✓ Funciona correctamente, mientras que:

```
1 == 2 || (2 ==1)
```

✗ No funciona correctamente debido a que no posee paréntesis para el primer término.

A medida del avance del desarrollo del lenguaje se vieron conflictos con *shift/reduce* y *reduce/reduce*, los mismos pudieron ser resueltos con las herramientas vistas en clase y lo encontrado en la sección de referencias.

Posibles extensiones

Una posible extensión para generar al lenguaje es la inclusión de objetos, en esta iteración se consideró que sería afrontar mucho contenido para una primer versión del lenguaje ya que el paradigma orientado a objetos no suele ser introductorio a un lenguaje de programación y ya se abarca contenido de programación imperativa al igual que la estructura subyacente detrás de todo testeo unitario en la iteración actual del lenguaje programAR.

Recibir argumentos al momento de ejecutar el programa similar a cómo funciona *argv* en C.

El agregar tipos de datos como *float*, *double*, *long*, entre otros y la validación en compilación de los mismos es una funcionalidad a agregar a futuro que sería de utilidad para mayores casos de uso del lenguaje.

Tests

Como requisito del enunciado se pide la realización de ejemplos de uso del lenguaje ProgramAR. En esta sección se detalla el fin de cada ejemplo.

test1.ar

Se le requiere al usuario por entrada estándar un número, al mismo suma los dígitos de un número entre sí y lo imprime como salida.

Se valida el funcionamiento correcto de los separadores de contexto, la asignación, la salida estándar, lectura por entrada estándar, la validación por tipo a través de condicionales y se hace uso de un bloque *do-while*.

test2.ar

Pide por entrada estándar dos números positivos y devuelve si algún número ingresado es negativo.

Se valida el funcionamiento correcto de los separadores de contexto, la asignación, la salida estándar, lectura por entrada estándar de más de un elemento y el uso de condicionales *if-else*.

test3.ar

Lee por entrada estándar dos números ingresados y calcula el mínimo común divisor (MCD) entre ambos números como salida esperada.

Se valida el funcionamiento correcto de los separadores de contexto, la asignación, la salida estándar, lectura por entrada estándar de más de un elemento, el uso de operadores aritméticos y el uso de un ciclo *do-while*.

test4.ar

Por entrada estándar se lee un número a ser ingresado y se obtiene como salida el factorial de dicho número.

Se valida el funcionamiento correcto de los separadores de contexto, la asignación, la salida estándar, lectura por entrada estándar, el uso de operadores aritméticos y el uso de un ciclo *do-while*.

test5.ar y test5.c

Realiza la impresión de 1 a 100000 impresiones por salida estándar a modo de benchmarking entre el lenguaje C y ProgramAR.

Se valida el funcionamiento correcto de los separadores de contexto, la salida estándar y la asignación de variables a modo de testeo de performance.

test6.ar

Se requiere por entrada estándar un número y por salida estándar se recibe si el mismo es par o impar.

Se valida el funcionamiento correcto de los separadores de contexto, la asignación, la salida estándar, lectura por entrada estándar y el uso de condicionales *if-else*.


test7.ar

Realiza un define, crea una variable la cual le asigna el valor de esa constante y se imprime la misma por salida estándar.

Se valida el funcionamiento correcto de los separadores de contexto, la asignación de variables, salida estándar y el uso de constantes.

Bibliografía y referencias

- Clases y bibliografía aportada por la cátedra:
 - [Intro a los Compiladores \(Presentación sobre Compiladores\)](#)
 - [Lex \(Presentación sobre Lex\)](#)
 - [Yacc \(Presentación sobre Yacc\)](#)

-
- [Ejemplos Yacc/Lex](#)
 - [Arquitectura de Compiladores](#)
 - [faturita/YetAnotherCompilerClass: Lex and Yacc samples and tools repository for Languages and Compiler class.](#)
 - Lex & Yacc - Doug Brown, John R. Levine, and Tony Mason.
 - [Ubuntu Manpage: flex - generador de analizadores léxicos rápidos.](#)
 - [Condiciones de arranque para separadores de contexto.](#)
 - [Introducción a yacc.](#)
 -  Part 02: Tutorial on lex/yacc. .
 - [Implementación de Linked List.](#)