



Universidad Austral de Chile
Conocimiento y Naturaleza

TRADUCTOR DIRIGIDO

INFO165: Compiladores

Integrantes:

Claudia Arias Kenigs
Andrés Gutiérrez Salas
Esteban Tejeda Webar

Profesora:

María Eliana de la Maza W.

11 de Diciembre, 2020

Introducción

En esta tarea se desarrolló un traductor dirigido por sintaxis que permite ejecutar programas escritos en lenguaje LMA (Lenguaje Manejador de Arreglos). El traductor acepta una secuencia de instrucciones del lenguaje LMA y efectúa las acciones a medida que se realiza el análisis sintáctico. Para ello se desarrollaron las etapas de análisis léxico y análisis sintáctico del programa, junto con una traducción dirigida por sintaxis.

Recapitulando, se optó por el lenguaje JAVA que resultó ser un medio más cómodo y práctico a la hora de trabajar y desarrollar esta actividad. Por ser este el lenguaje escogido, para una etapa previa se trabajó con JFLEX (erróneamente referido como FLEX en el anterior informe), para el desarrollo del análisis léxico, por tanto, para esta etapa final se trabajó con CUP, pues estas dos librerías son perfectamente compatibles entre sí para el lenguaje elegido.

Tras la selección de tecnología comentada en un informe anterior, se implementaron las propuestas para la gramática en un programa diseñado con las especificaciones ahí comentadas, aunque con ciertos cambios. Las instrucciones proporcionadas fueron:

- **PARTIR:** Primera instrucción de cualquier programa.
- **INICIAR(nomarr,e1,e2,e3,...,e8):** Crea un arreglo llamado *nomarr* con los elementos e1,e2,e3,...,e8 (números enteros positivos). Si el arreglo tiene menos de 8 elementos el resto se completa con ceros.
- **METER(nomarr,x,y):** En *nomarr*, inserta el elemento x en la posición y.
- **SACAR(nomarr,y):** En *nomarr*, elimina el elemento que se encuentra en la posición y.
- **MIRAR(nomarr):** Despliega en pantalla los elementos de *nomarr*.
- **DATO(nomarr,x):** Despliega en pantalla el elemento de *nomarr*, que se encuentra en la posición x.
- **FINALIZAR:** Última instrucción de cualquier programa.

Luego, se presentaron los ajustes a estas instrucciones, definiendo:

Tipo	Definición
ID	Palabras que empiezan con "L" y terminan con al menos un número entero positivo.
NUM	Número enteros positivos.
POS	Un único entero positivo (entre 1 y 8).

Con lo anterior queda la gramática:

PAR → PARTIR INST FINALIZAR

INST → INST INST | INI | MET | SAC | MIR | DAT

INI → INICIAR(ID,NUM,NUM,NUM,NUM,NUM,NUM,NUM,NUM)

MET → METER(ID,NUM,POS)

SAC → SACAR(ID,POS)

MIR → MIRAR(ID)

DAT → DATO(ID,POS)

Instrucciones de ejecución

Requisitos:

- Poseer el archivo comprimido adjunto, llamado Tarea.
- Poseer Winrar o similar para poder descomprimirlo.
- Poseer alguna versión instalada y funcionla de JDK:

<https://www.oracle.com/cl/java/technologies/javase/javase-jdk8-downloads.html>

Acciones:

1. Descomprimir archivo adjunto, aparecerá una carpeta llamada Tarea_COPIADORES_ARIAS_GUTIERREZ_TEJEDA.
2. Entrar Tarea_COPIADORES_ARIAS_GUTIERREZ_TEJEDA/Ejecutable
3. Doble click en el archivo Traductor.JAR.

Ejecución:

1. Escribir la instrucción PARTIR.
2. Escribir cualquiera de las instrucciones INICIAR, METER, SACAR, MIRAR o DATO.
También es posible no poner ninguna y proceder al siguiente punto.
 - Las últimas no funcionarán a menos que se de un ingreso válido de INICIAR.
3. Escribir la instrucción FINALIZAR.



Integración

Durante el desarrollo del programa, el intentar implementar tal gramática requirió modificaciones a la producción INST -> INST INST, añadiendo una nueva producción:

PAR →	PARTIR INST FINALIZAR PARTIR FINALIZAR
INST →	INST INST_F INST_F
INST_F →	INI MET SAC MIR DAT
INI →	INICIAR(ID,NUM,NUM,NUM,NUM,NUM,NUM,NUM,NUM)
MET →	METER(ID,NUM,NUM)
SAC →	SACAR(ID,NUM
MIR →	MIRAR(ID)
DAT →	DATO(ID,NUM)

Se optó además de eliminar al agregado de POS, porque NUM podía emplear la misma función, si lo restringía adecuadamente. El tener las dos definiciones, también ocasionó algunos problemas de compatibilidad entre distintos programas.

Para organizar la información a ingresar, se definieron los conjuntos:

Palabras reservadas	Argumentos	Símbolos -> Nombre referencia
PARTIR	ID	(-> Pa
INICIAR	NUM) -> Pc
METER	POS	, -> COMA
SACAR	-	-
MIRAR	-	-
DATO	-	-
FINALIZAR	-	-

A partir de la información anterior, es importante establecer cuáles serán los terminales y variables de la gramática usará.

Terminales = (PARTIR ; INICIAR ; METER ; SACAR ; MIRAR ; DATO ; FINALIZAR ; NUM ; ID , ERROR ; Pa ; Pc ; COMA)

Variables = (PAR ; INST ; INST_F ; INI ; MET ; SAC ; MIR; DAT)

Gramática integrada, acciones semánticas

Para la biblioteca de CUP, el poder integrar una gramática requiere expresar en una forma más explícita cada producción, reemplazando la gramática original por el conjunto de términos anteriores. En los parámetros de CUP, se posee la siguiente estructura para definir cualquier elemento:

VARIABLE ::= TERMINAL **TERMINAL**:a **VARIABLE** | (*) {:(instrucciones) :} ;

- **VARIABLE**: Cualquier elemento perteneciente al conjunto de variables.
- **::=** : Conector de las variables de la gramática con las definiciones de sus producciones (\leftarrow).
- **TERMINAL**: Cualquier elemento perteneciente al conjunto de terminales. Puede o no ir acompañada de una de las referencias comentadas a continuación.
- **:a** : Nombres de referencia al valor del atributo (int, String o lo que haga falta). Debe ir adjunto a un terminal.
- **|** : Separador entre producciones.
- **(*)**: Cualquier conjunto de variables, terminales, nombres de referencia



- `{: (instrucciones) :};`: Espacio reservado el equivalente en java de las acciones semánticas asociadas a cada producción. De no existir ninguna acción, se puede dejar solo un `;`. Las instrucciones se escriben en lenguaje JAVA, y pueden usar los nombres de referencia de los atributos.

PAR ::= PARTIR INST FINALIZAR | PARTIR FINALIZAR;

- *Instrucción inicial, usa los terminales PARTIR y FINALIZAR para como indica su nombre dar inicio y término al programa.*

INST ::= INST INST_F | INST_F;

- *Instrucción que busca generar a la cantidad de instrucciones que se ajusten a las que cree el usuario.*

INST_F ::= INI | MET | SAC | MIR | DAT;

- *Concretiza que instrucción se aplicará.*

**INI ::= INICIAR Pa ID:id COMA NUM:n1 COMA NUM:n2 COMA NUM:n3 COMA NUM:n4
COMA NUM:n5 COMA NUM:n6 COMA NUM:n7 COMA NUM:n8 Pc**

`{: parser.iniciar(id.toString(), n1.toString(), n2.toString(), n3.toString(), n4.toString(), n5.toString(), n6.toString(), n7.toString(), n8.toString());};`

- *Crea un arreglo de nombre m id, y le pasa los atributos $n1, n2, \dots, n8$*

MET ::= METER Pa ID:id COMA NUM:n COMA NUM:p Pc

`{: parser.mete(id.toString(), n.toString(), p.toString());};`

- *Añade al arreglo id, en la posición p el valor de n, a la vez que de estar en p ya un elemento, mueve ese y aquellos elementos a su derecha.*

SAC ::= SACAR Pa ID:id COMA NUM:p Pc

`{: parser.saca(id.toString(), p.toString());};`

- *Retira del arreglo el atributo en la posición p, moviendo los demás atributos un espacio a la izquierda.*

MIR ::= MIRAR Pa ID:id Pc

```
{: parser.mira(id.toString());};
```

- *Despliega en pantalla los elementos distintos de 0, presentes en id.*

DAT ::= DATO Pa ID:id COMA NUM:p Pc

```
{: parser.mostrarDato(id.toString(),p.toString());};
```

- *Muestra el dato localizado en p del arreglo id.*

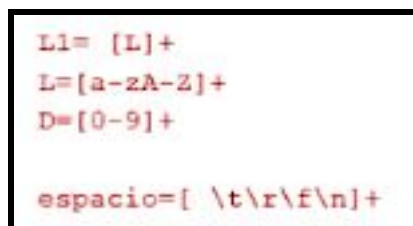
Nota: Las funciones definidas acá en las acciones semánticas, son bastante largas por la cantidad de clases a las que extienden. Para una vista más detallada, ver el archivo Syntax.cup.

Análisis sintáctico

LexerCup.flex:

Dirección: Tarea/src/codigo/LexerCup.flex

Acá se declaran las definiciones para el lenguaje que maneja el programa, es decir, palabras reservadas, identificadores y símbolos de los tokens que se utilizan en la gramática. Estos se definen y se almacenan los nombres de estos en un archivo Token.Java, que se usa para referenciar los nombres de estos.



```
L1= [L]+  
L=[a-zA-Z]+  
D=[0-9]+  
  
espacio=[ \t\r\f\n]+
```

Imagen 1. Declaración de elementos del lenguaje

```
( "PARTIR" )      {return new Symbol(sym.PARTIR, yychar, yyline, yytext());}
( "INICIAR" )     {return new Symbol(sym.INICIAR, yychar, yyline, yytext());}
( "METER" )       {return new Symbol(sym.METER, yychar, yyline, yytext());}
( "SACAR" )       {return new Symbol(sym.SACAR, yychar, yyline, yytext());}
( "MIRAR" )       {return new Symbol(sym.MIRAR, yychar, yyline, yytext());}
( "DATO" )        {return new Symbol(sym.DATO, yychar, yyline, yytext());}
( "FINALIZAR" )   {return new Symbol(sym.FINALIZAR, yychar, yyline, yytext());}
```

Imagen 2. Declaración de elementos esenciales de la gramática.

```
( "(" )           {return new Symbol(sym.Pa, yychar, yyline, yytext());}
( ")" )           {return new Symbol(sym.Pc, yychar, yyline, yytext());}
( ",", " )        {return new Symbol(sym.COMA, yychar, yyline, yytext());}
```

Imagen 3. Declaración de símbolos de la gramática.

```
(espacio) { /*Ignore*/ }
(L1){(L){(D)}* (D) {return new Symbol(sym.ID, yychar, yyline, yytext());}
(D)+ {return new Symbol(sym.NUM, yychar, yyline, yytext());}

.. {return new Symbol(sym.ERROR, yychar, yyline, yytext());}
```

Imagen 4. Declaración de los elementos añadidos a la gramática en el informe anterior.
(Se conservó la definición de POS, pero no se utiliza)

Array.java

Dirección: Tarea/src/codigo/Array.java

Esta clase, crea los objetos "Array", que serán conformados por un String **name**, y un arreglo de Int **arrayNumber**, como el Editor **edit** para desplegarse en pantalla:

```
public class Array{
    String name;
    int[] arrayNumber;
    Editor edit;
```

Imagen 5. Crea el arreglo asignando sus elementos.


```
public Array(String name, int[] arrayNumber){  
    this.name = name;  
    this.arrayNumber = arrayNumber;  
}
```

Imagen 6. Construye el arreglo asignando sus valores con los ingresos *name* y *arrayNumber*..

```
public String getName(){  
    return name;  
}
```

Imagen 7. Añade el número *num* al arreglo en la posición *pos*.

```
public void addNumber(int num, int pos){  
    if(arrayNumber[pos-1] == 0) arrayNumber[pos-1] = num;  
    else(  
        int[] arrayAux = new int[8];  
        for(int i = 0; i<8; i++) arrayAux[i] = arrayNumber[i];  
        arrayNumber[pos-1] = num;  
        for(int i = pos; i < 8; i++) arrayNumber[i] = arrayAux[i-1];  
    )  
}
```

Imagen 8. Añade el número *num* al arreglo en la posición *pos*, luego desplaza los números a la derecha del nuevo, en caso de estar este entremedio.

```
public void delNumber(int pos){  
    int arrayAux[] = {0,0,0,0,0,0,0,0};  
    for(int i = 0; i < 8-pos; i++) arrayAux[i] = arrayNumber[pos+i];  
    arrayNumber[pos-1] = 0;  
    for(int i = 0; i < 8-pos; i++) arrayNumber[i+pos-1] = arrayAux[i];  
    arrayNumber[7] = 0;  
}
```

Imagen 9. Elimina el número en *pos*, lo reemplaza por un 0.

```
public int getNumber(int pos){  
    return arrayNumber[pos-1];  
}
```

Imagen 10. Recibe y entrega la posición *pos*.

```
public void printArray(){  
    for(int i = 0; i < 7; i++){  
        if(arrayNumber[i] != 0) System.out.print(arrayNumber[i]+" ");  
    }  
    if(arrayNumber[7] != 0) System.out.print(arrayNumber[7]);  
}
```

Imagen 11. Imprime los números del arreglo, que no sean 0.

Sintax.cup

Dirección: Tarea/src/codigo/Sintax.cup

Desde acá se realiza la mayor parte del trabajo, pues este archivo posee los diversos métodos que emplea la gramática en las acciones semánticas, como las definiciones anteriores de terminales/no terminales y la gramática que fue definida antes.

Principal.java

Dirección: Tarea/src/codigo/Principal.java

Este archivo se dedica a construir todo archivos en formato .java, como los equivalentes de LexerCup.java, que se ejecuta para vincular este archivo con la programación en JAVA.

Editor.java

Dirección: Tarea/src/codigo/Editor1.java

Este archivo genera la ventana para que el usuario interactúe con el analizador.

Dificultades

El proceso de desarrollo no estuvo exento de problemas, pues existieron muchas instancias donde se pausó el progreso por errores que impedían añadir más cambios al programa.

- **Tecnología** - Pese al razonamiento inicial de qué trabajar con tecnología JAVA resultará más fácil dado el manejo del lenguaje, pero el desarrollo con las herramientas JFLEX y CUP llevó a reiterados problemas en cuanto a validaciones de segmentos de código que no pudo compilar. La mayor expresión de esto, fue al intentar aplicar las acciones semánticas en el código. Este problema fue recurrente, pero se vio disminuir a medida que el código se afinaba, y tomaba consistencia para terminar lo visto antes.
 - *Solución* - Ensayo y error. Seguir probando el código, estudiando y razonando el código hasta dar con la solución y la información que aportó.
- **Teoría/Práctica** - No resultó ser tan sencillo adaptar los conceptos vistos en clases al entorno en que se programó. Se requirió en gran medida de replanteamientos y reescrituras del código, con tal de cumplir lo solicitado.
 - *Solución* - Conseguir documentación extra, analizar material externo hasta que lo plasmado en papel se pudo implementar. La versión final terminó con cambios importantes en la gramática más que nada.
- **Tiempo** - Pese a contar con las facilidades de una semana más flexible posterior a la entrega del trabajo, las diversas circunstancias determinaron que las semanas del desarrollo estuvieran llenas de evaluaciones y compromisos que no se podían postergar. No se logró estimar un rango de tiempo adecuado de cuánto podría demorar el desarrollo completo, cosa que jugó en contra con la fecha de entrega que terminó siendo posterior al plazo dado.
 - *Solución* - Trabajo continuo. Para quién significó una experiencia distinta, más el constante trabajo fué la única forma de terminar.

Ejemplo ejecución

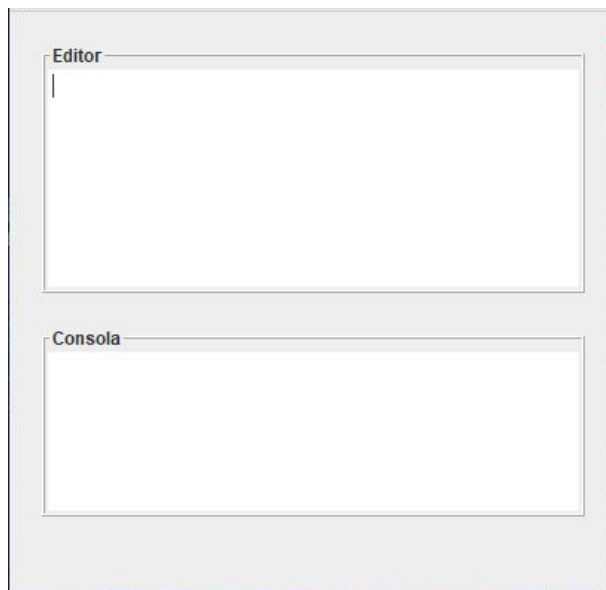


Imagen 12

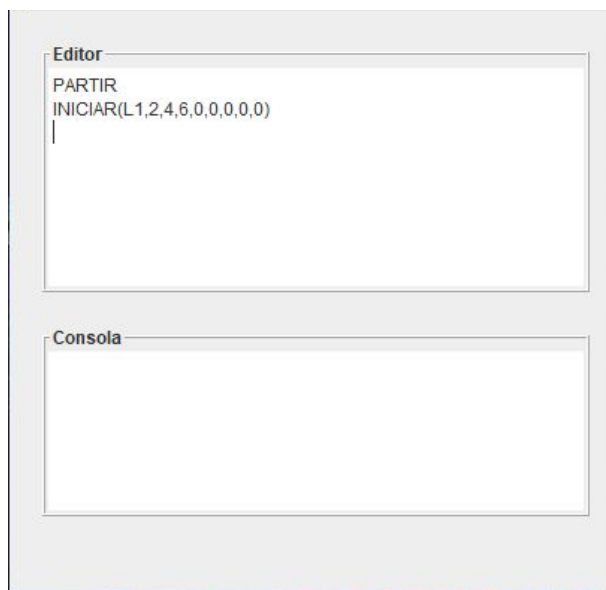


Imagen 13



Imagen 14

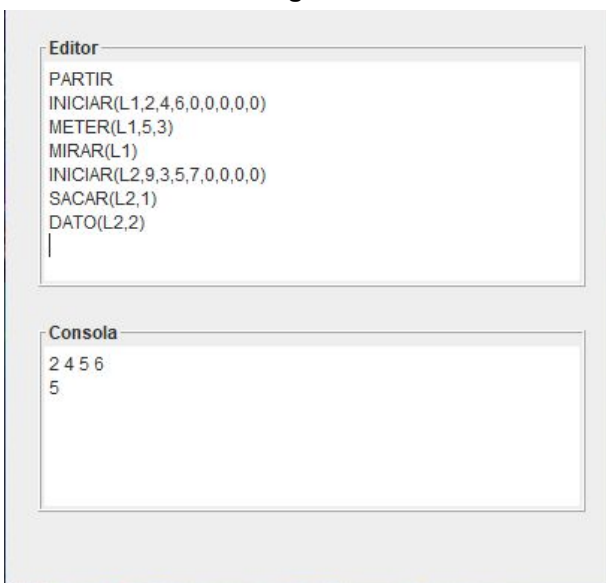


Imagen 15

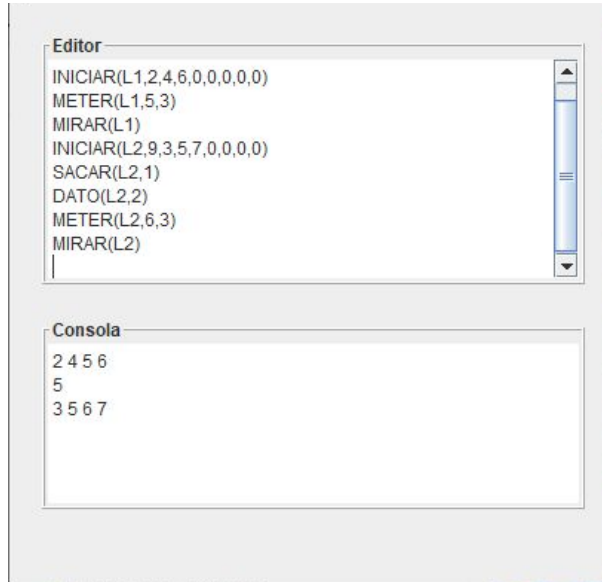


Imagen 16

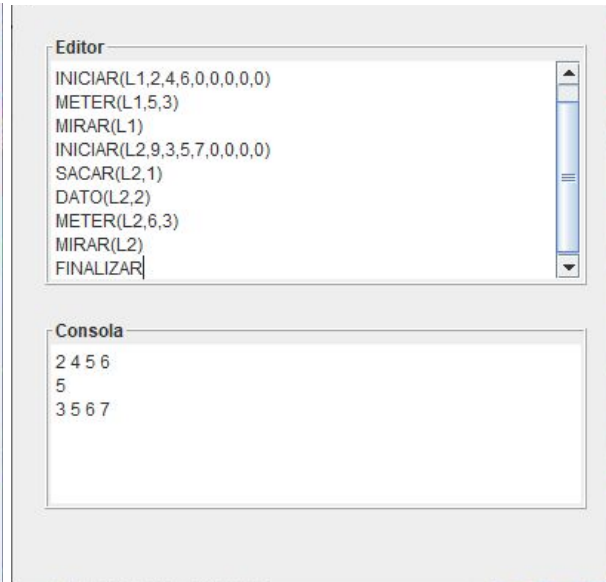


Imagen 17

Conclusiones

“Construir un compilador no es difícil, es laborioso.” - Maria Eliana de la Maza W.

Con este planteamiento inició el curso de compiladores. Si bien esta actividad no busca terminar por desarrollar uno, denota como es el proceso de la elaboración de uno, incluso a base de herramientas ya predefinidas como JFLEX y CUP.

El adecuado uso de estas tecnologías abre paso a la posibilidad de desarrollo y crecimiento en campo laboral y profesional de cualquier estudiante de la carrera, pues más allá de ser el conocimiento en bruto que queda, es la experiencia de trabajar en un proyecto monótono y repetitivo, pero que a cada rato puede ir presentado imprevistos, sorpresas o logros personales cumplidos. En ocasiones simplemente hacer que un par de líneas no tiren error se siente más como un logro que completar el proyecto en sí, pero eso está completamente sujeto a la interpretación de quién programa.

Lo anterior, pese a como se le desee abordar, no cambia un hecho: El programa es completamente mejorable. En un proyecto real es de imaginar que quienes usen el producto final procuren que no tenga errores en su entrega, pero este código se hizo con el razonamiento de que ciertas pruebas no se efectuaron, tales como intentar adrede probar un ingreso erróneo (poner 9 números al inicializar un arreglo), o incluso ingresos válidos pero que podrían agregar problemas con posteriores instrucciones, como agregar a un arreglo con 3 números guardados, un nuevo número en la casilla 6. No es que no se pueda programar, si no que se optó por ignorar posibles problemas, al tener conocimiento del tipo de pruebas por las que será evaluado, no aspirando a que el entregable final sea el producto más óptimo si algún externo lo usa. Esto evidenció lo detallado que es un proceso de desarrollo, más allá de la teoría, que en retrospectiva se fue estudiando a lo largo de los cuatro cursos, para concluir en el desarrollo de esta actividad.



Bibliografía

Apuntes:

- Curso “Compiladores INFO165”
- Repaso de gramáticas, clase 14

Referencias Web:

- https://www.u-cursos.cl/ingenieria/2009/2/CC3102/1/material_docente/bajar?id=260381
- http://www2.cs.tum.edu/projects/cup/docs.php#action_part
- <http://www.cc.uah.es/ie/docencia/ProcesadoresDeLenguaje/CUP.pdf>