Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Organización de Lenguajes y Compiladores 1
Vacaciones de Junio 2024
Catedrático:
Ing. Mario Bautista
Tutor académico:
Fabian Reyna

JavaCraft

Tabla de Contenido

1. Objetivo General

Aplicar los conocimientos sobre la fase de análisis léxico, sintáctico y semántico de un compilador para la realización de un intérprete sencillo, con las funcionalidades principales para que sea funcional.

2. Objetivos específicos

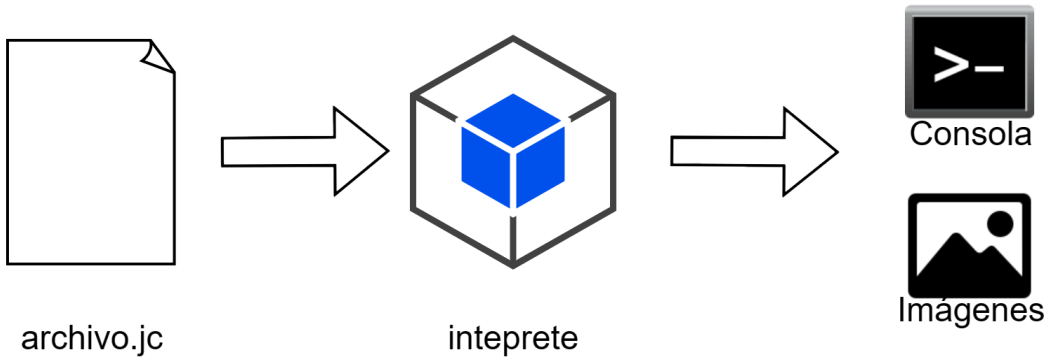
- Reforzar los conocimientos de análisis léxico, sintáctico y semántico para la creación de un lenguaje de programación.
- Aplicar los conceptos de compiladores para implementar un proceso de interpretación de código de alto nivel.
- Construir un árbol de sintaxis abstracta (AST) para la ejecución de código de alto nivel.
- Detectar y reportar errores léxicos, sintácticos y semánticos.

3. Descripción General

El curso de Organización de Lenguajes y Compiladores 1, ha puesto en marcha un nuevo proyecto que consiste en crear un lenguaje de programación para poder utilizar en los proyectos de los cursos de Introducción a la Programación y Computación 1 y 2. Este lenguaje está fundamentado en los principales conceptos de programación, con un enfoque particular en el uso de estructuras de datos.

Por tanto, a usted, que es estudiante del curso de Compiladores 1, se le encomienda realizar el proyecto llamado JavaCraft. dado sus altos conocimientos en temas de análisis léxico, sintáctico y semántico.

Flujo de la aplicación



4. Entorno de Trabajo

4.1 Editor

La función principal del editor será el ingreso de código fuente que será analizado. Queda a discreción del estudiante el diseño.

4.2 Funcionalidades

- **Nuevo archivo:** El editor debe tener la capacidad de crear archivos en blanco.
- **Abrir archivos:** El editor deberá abrir archivos .jc, cuyo contenido se mostrará en el área de entrada en una nueva pestaña con el nombre del archivo
- **Guardar archivo:** El editor deberá guardar el estado del archivo en el que se está trabajando.
- **Eliminar pestaña:** Cada pestaña puede ser cerrada en cualquier momento. Si los cambios no se han guardado, se descartan.

4.3 Características

- **Múltiples pestañas**: se podrán crear nuevas pestañas con la finalidad de ver, abrir y editar los archivos de entrada de la aplicación. Para cada pestaña, corresponde un archivo.

4.4 Herramientas

- **Ejecutar**: hará el llamado al intérprete, el cual se hará cargo de realizar los análisis léxico, sintáctico y semántico, además de ejecutar todas las sentencias.

4.5 Reportes

- **Reporte de Errores:** se mostrarán todos los errores encontrados al realizar el análisis léxico, sintáctico y semántico.
- **Generar AST**: se debe generar el ast del último archivo analizado y mostrarlo desde la interfaz.
- **Reporte de Tabla de Símbolos**: se mostrarán todas las variables, métodos y funciones que han sido declaradas dentro del flujo del programa.

4.6 Área de Consola

En esta área se mostrarán los resultados, mensajes y todo lo que sea indicado dentro del lenguaje.

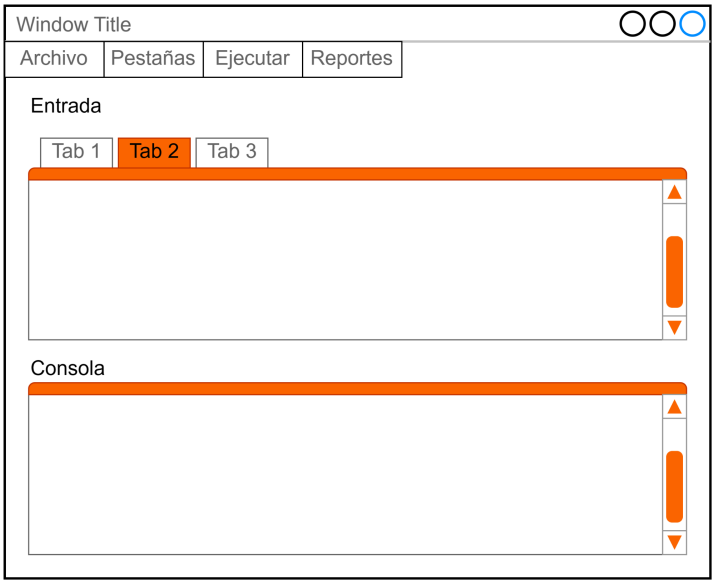


Figura 1. Propuesta de interfaz

5. Descripción del Lenguaje

5.1 Case Insensitive

El lenguaje es case insensitive por lo que no reconoce entre mayúsculas y minúsculas.

```
var a: int = 1;
Var A :Int = 1;
```
Nota: Ambos casos son lo mismo

5.2 Comentarios

5.2.1 Comentarios de una línea

Este comentario comenzará con // y deberá terminar con un salto de línea.

5.2.2 Comentarios multilínea

Este comentario comenzará con /* y terminará con */.

```
// Esto es un comentario de una sola línea
/*
Esto es un
comentario multilínea
*/
```

5.3 Tipos de Dato

El lenguaje JavaCraft es fuertemente tipado, esto quiere decir que se conoce el tipo de una variable desde el momento en que se declara. También se tiene un tipado estático, lo que indica que una variable puede tener solo un valor durante toda la ejecución.

Tipo	Definición	Descripción	Ejemplo	Observaciones	Default
Entero	int	Este tipo de dato aceptará solamente números enteros	1, 50, 100, -120, etc	Del -2147483648 al 2147483647	0
Decimal	double	Admite valores numéricos con decimales	1.2, 50.23, -00.34. etc	Se maneja cualquier cantidad de decimales	0.0
Booleano	bool	Admite valores que indican verdadero o falso	true, false	Si se asigna un valor booleano a un entero se tomará como 1 o 0 respectivamente	true
Carácter	char	Tipo de dato que únicamente aceptará un único carácter, y estará delimitado por comillas simples	‘a’, ‘b’, ‘c’, ‘E’, ‘1’, ‘&’, ‘\n’, etc	En caso de querer escribir comilla simple, se escribirá \ y después la comilla simple \’. Si se quiere escribir \ se escribirá \\. Existirá también \n, \t, \r, \”.	‘\u0000’ (carácter 0)
Cadena	String	Es un grupo o conjunto de caracteres que pueden tener cualquier carácter, y este se encontrará delimitado por comillas dobles. “ “	“cadena”, “- cad”	Se permitirá cualquier carácter entre las comillas dobles, incluyendo las secuencias de escape	“” (string vacío)

5.4 Secuencias de escape

Las secuencias de escape se utilizan para definir ciertos caracteres especiales dentro de cadenas de texto. Las secuencias de escape disponibles son las siguientes:

Secuencia	Descripción	Ejemplo
\n	Salto de línea	“Hola\nMundo”
\\	Barra invertida	“C:\\miCarpeta”
\”	Comilla doble	“\”esto es una cadena\””
\t	Tabulación	“\tEsto es una tabulación”
\’	Comilla simple	“\’Estas son comillas simples\’”

5.5 Operadores Aritméticos

5.5.1 Suma

Es la operación aritmética que consiste en realizar la suma entre dos o más valores. Para esta se utiliza el signo más **(+)**.

Especificaciones de la operación suma

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

+	Entero	Decimal	Booleano	Carácter	Cadena
Entero	Entero	Decimal		Entero	Cadena
Decimal	Decimal	Decimal		Decimal	Cadena
Booleano					Cadena
Carácter	Entero	Decimal		Cadena	Cadena
Cadena	Cadena	Cadena	Cadena	Cadena	Cadena

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.2 Resta

Es la operación aritmética que consiste en realizar la resta entre dos o más valores. Para esta se utiliza el signo menos **(-)**.

Especificaciones de la operación resta

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

-	Entero	Decimal	Carácter
Entero	Entero	Decimal	Entero
Decimal	Decimal	Decimal	Decimal
Carácter	Entero	Decimal	

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.3 Multiplicación

Es la operación aritmética que consiste en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). El signo para representar la operación es el asterisco **(*)**.

Especificaciones de la operación multiplicación

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

*	Entero	Decimal	Carácter
Entero	Entero	Decimal	Entero
Decimal	Decimal	Decimal	Decimal
Carácter	Entero	Decimal	

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.4 División

Es la operación aritmética que consiste en partir un todo en varias partes, al todo se le conoce como dividendo, al total de partes se le llama divisor y el resultado recibe el nombre de cociente. El operador de la división es la diagonal **(/)**.

Especificaciones de la operación división

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

/	Entero	Decimal	Carácter
Entero	Decimal	Decimal	Decimal
Decimal	Decimal	Decimal	Decimal
Carácter	Decimal	Decimal	

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.5 Potencia

Es una operación aritmética de la forma **a**b** donde a es el valor de la base y b es el valor del exponente que nos indicará cuantas veces queremos multiplicar el mismo número.

Especificaciones de la operación potencia

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

**	Entero	Decimal
Entero	Entero	Decimal
Decimal	Decimal	Decimal

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.6 Módulo

Es una operación aritmética que obtiene el resto de la división de un número entre otro. Para realizar la operación se utilizará el signo **(%)**.

Especificaciones de la operación módulo

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

%	Entero	Doble
Entero	Decimal	Decimal
Decimal	Decimal	Decimal

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.7 Negación Unaria

Es una operación que niega el valor de un número, es decir que devuelve el contrario del valor original. Se utiliza el símbolo menos **(-)**.

Especificaciones de la operación negación

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

-exp	Resultado
Entero	Entero
Decimal	Decimal

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.6 Operadores Relacionales

Son los símbolos que tienen como finalidad comparar expresiones, dando como resultado valores booleanos. A continuación, se definen los símbolos que serán aceptados dentro del lenguaje:

Operador	Descripción	Ejemplo
==	**Igualación**: compara ambos valores y verifica si son iguales	1==1 25.654 == 54.34
!=	**Diferenciación**: compara ambos lados y verifica si son distintos	1 != 2 50 != 30
<	**Menor que**: compara ambos lados y verifica si el derecho es mayor que el izquierdo.	25.5 < 30 50 < 'F'
<=	**Menor o igual que**: Compara ambos lados y verifica si el derecho es mayor o igual que el izquierdo.	25.5 <= 30 50 <= 'F'
>	**Mayor que**: compara ambos lados y verifica si el izquierdo es mayor que el derecho.	25.5 > 30 50 > 'F'
>=	**Mayor o igual que**: Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho.	25.5 >= 30 50 >= 'F'

Especificaciones de los operadores relacionales

A continuación, se especifica en una tabla los resultados que se deberán obtener con estas operaciones.

Relacional	Entero	Decimal	Booleano	Carácter	Cadena
Entero	Booleano	Booleano		Booleano	
Decimal	Booleano	Booleano		Booleano	
Booleano			Booleano		
Carácter	Booleano	Booleano		Booleano	
Cadena					Booleano

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.7 Operadores Lógicos

Son los símbolos que tienen como finalidad comparar expresiones a nivel lógico (verdadero o falso). A continuación, se definen los símbolos que serán aceptados dentro del lenguaje:

Operador	Descripción	Ejemplo
||	**OR**: compara expresiones lógicas y si al menos una es verdadera, entonces devuelve verdadero y en otro caso retorna falso.	true || 5<2 Devuelve true
&&	**AND**: compara expresiones lógicas y si ambas son verdaderas, entonces devuelve verdadero y en otro caso retorna falso.	true && “hola” == “hola” Devuelve true
^	**XOR**: compara expresiones lógicas y si ambas no son iguales retorna verdadero y en cualquier otro caso retorna false.	true ^ false devuelve true
!	**NOT**: devuelve el valor inverso de una expresión lógica si esta es verdadera entonces devolverá false, de lo contrario retorna verdadero	!true Devuelve false

5.8 Signos de Agrupación

Los signos de agrupación serán utilizados para agrupar operaciones aritméticas, lógicas o relacionales. Los símbolos de agrupación están dados por **(** y **)**.

```
var edad:int = (10 + 10) - 5;

var bandera: bool = (5 == 5) == true;

var bandera2: bool = (1 + 2) > (3 - 1);
```

5.9 Precedencia de Operaciones

La precedencia de operadores nos indica la importancia de que una operación debe realizarse por encima del resto. A continuación, se define la misma:

Nivel	Operador	Asociatividad
0	-	Derecha
1	**	No asociativa
2	*, /	Izquierda
3	+, -	Izquierda
4	==, !=, <, <=, >, >=	Izquierda
5	!	Derecha
6	^	Izquierda
7	&&	Izquierda
8	||	Izquierda

Nota: el nivel 0 es el nivel de mayor importancia

5.10 Caracteres de finalización y encapsulamiento de sentencias

El lenguaje se verá restringido por dos reglas que ayudan a finalizar una instrucción y encapsular sentencias:

- **Finalización de instrucciones:** para finalizar una instrucción se utilizará el signo **;**

```
var edad:int = 18;
```

- **Encapsular sentencias:** para encapsular sentencias dadas por ciclos, métodos, funciones, etc, se utilizará los signos **{** y **}**.

```
if (true) {
  var edad:int = 18;
}
```

5.11 Declaración de variables

Las variables deben ser declaradas antes de su uso, especificando su mutabilidad, un nombre de identificador y un tipo de dato. Pueden declararse tanto a nivel global como local.

Aquellas variables definidas con la palabra reservada ***const*** conservarán su valor constante durante toda la ejecución, mientras que las declaradas con ***var*** podrán cambiar su valor. Si no se especifica un valor inicial, la variable tomará el valor por defecto correspondiente a su tipo de dato.

```
<MUTABILIDAD> <ID> : <TIPO> ;
<MUTABILIDAD> <ID> : <TIPO> = <EXPRESION> ;

var num : int;
const nota : double;

var num2 :  int = 10;
const nota2 : double  = 20.6;
```

Las variables no pueden cambiar de tipo de dato, se deben mantener con el tipo declarado inicialmente, por lo que se debe de validar que el tipo de la variable y el valor sean compatibles.

5.12 Asignación de variables

Esta instrucción permitirá modificar el valor de la variable que fue previamente declarada, siempre y cuando sea posible modificarla.

```
var num : int;
const nota : double;

num = 10;
nota = 20.6;
```

Nota: La asignación de num, actualizará el valor de esta de 0 (valor por defecto) a 10, mientras que nota no sería actualizada ya que fue declarada como const y debería ser un error semántico

5.13 Casteos

Los casteos son una forma de indicar al lenguaje que convierta un tipo de dato en otro, por lo que, si queremos cambiar un valor a otro tipo, es la forma adecuada de hacerlo. Para hacer esto, se colocará la palabra reservada del tipo de dato destino entre paréntesis seguido de una expresión.

(**<TIPO>**) **<EXPRESION>**

El lenguaje aceptará los siguientes casteos:

- int a double
- double a int
- int a char
- char a int
- char a double

```
var edad : int = (int) 18.6;
const letra : char = (char) 70;
var numero : double  = (double) 16;
```

Nota:

- La variable edad tomaría el valor de 18.
- La variable letra tomaría el valor de ‘F’ ya que 70 es el ascii de F.
- La variable numero tomaría el valor de 16.0

5.14 Incremento y Decremento

Los incrementos y decrementos nos ayudan a realizar la suma o resta continua de un valor de uno en uno, es decir si incrementamos una variable, se incrementará de uno en uno, mientras que, si realizamos un decremento, hará la operación contraria. Esta instrucción es válida tanto para Enteros como para Decimales.

```
<EXPRESION> + + ;
<EXPRESION> - - ;

var edad : int = 18;
edad++; // tiene el valor de 19
edad--; // tiene el valor de 18
```

5.15 Sentencias de control

Estas sentencias modifican el flujo del programa introduciendo condicionales. Las sentencias de control para el programa son el IF y el Match.

5.15.1 Sentencia IF

El lenguaje JavaCraft posee la sentencia IF similar a otros lenguajes de programación, la cual permite ejecutar bloques de código se ejecuten si la condición a evaluarse es verdadera. Esta sentencia se define por las instrucciones if, else if, else.

Consideraciones

- Las instrucciones else if y else son opcionales
- La instrucción else if se puede utilizar tantas veces como se desee

```
if ( <EXPRESION> ) {
    <INSTRUCCIONES>
}
|
if ( <EXPRESION> ) {
    <INSTRUCCIONES>
} else {
    <INSTRUCCIONES>
}
|
if ( <EXPRESION> ) {
    <INSTRUCCIONES>
} else <IF>

/*
 Son 3 variantes en total
    1.  IF
    2.  IF-ELSE
    3.  IF- ELSE IF
*/
```

5.15.2 Sentencia Match

El lenguaje JavaCraft posee la sentencia Match similar al switch case de otros lenguajes de programación, la cual es una estructura utilizada para agilizar la toma de decisiones múltiples, trabaja de la misma manera que lo harían sucesivos if.

```
/*
Estructura principal del match, donde se indica la expresión a evaluar.
*/
match <EXPRESION> {
    <CASES_LIST>
    <DEFAULT>
}
|
match <EXPRESION> {
    <CASES_LIST>
}
|
match <EXPRESION> {
   <DEFAULT>
}

/* Casos: Estructura que contiene las diversas opciones a evaluar con la expresión establecida en el match
*/
<EXPRESION> => {
    <INSTRUCCIONES>
}
 /* Default: Estructura que contiene las sentencias si en dado caso no se ha encontrado coincidencias con las anteriores.
 */
_ =>{
    <INSTRUCCIONES>
}
```

5.16 Sentencias Cíclicas

Los ciclos o bucles son una secuencia de instrucciones de código que se ejecutan una vez tras otra mientras la condición, que se ha asignado para que pueda ejecutarse, sea verdadera. En el lenguaje actual, se podrán realizar 3 sentencias cíclicas que se describen a continuación.
Observaciones:

- Es importante destacar que pueden tener ciclos anidados entre las sentencias a ejecutar.
- También, entre las sentencias pueden tener ciclos diferentes anidados

5.16.1 While

El ciclo o bucle While, es una sentencia que ejecuta una secuencia de instrucciones mientras la condición de ejecución se mantenga verdadera.

```
while ( <EXPRESION> ) {
   <INSTRUCCIONES>
}
```

5.16.2 For

El ciclo o bucle for, es una sentencia que nos permite ejecutar N cantidad de veces la secuencia de instrucciones que se encuentra dentro de ella.

Observaciones:

- Para la actualización de la variable del ciclo for se puede utilizar
  - **Incremento** | **Decremento**: i++ | i--
  - **Asignación**: como i = i+1, i = i-1, etc, es decir, cualquier tipo de asignación
- Dentro pueden venir N instrucciones

```
for ( <ASIGNACIÓN> ; <CONDICIÓN> ; <ACTUALIZACIÓN>  ) {
   <INSTRUCCIONES>
}
```

5.16.3 Do-While

El ciclo o bucle Do-While, es una sentencia que ejecuta al menos una vez el conjunto de instrucciones que se encuentran dentro de ella y que se sigue ejecutando mientras la condición sea verdadera.

Observaciones:

- Dentro pueden venir N instrucciones

```
do {
    [ <INSTRUCCIONES> ]
} while ( <EXPRESION> ) ;
```

5.17 Sentencias de Transferencia

Las sentencias de transferencia nos permiten manipular el comportamiento de los bucles, ya sea para detenerlo o para saltarse algunas iteraciones. El lenguaje soporta las siguientes sentencias:

5.17.1 Break

La sentencia break hace que se salga del ciclo inmediatamente, es decir que el código que se encuentre después del break en la misma iteración no se ejecutara y este se saldrá del ciclo.

```
break ;

var i : int = 0;
for(i = 0; i < 3; i++){
    if (i==2){
        break; // me salgo en i = 2 y ya no se ejecuta lo demás
    }
    println(i);
}

// Salida
/*
0
1
*/
```

5.17.2 Continue

La sentencia continue puede detener la ejecución de la iteración actual y saltar a la siguiente. La sentencia continue siempre debe de estar dentro de un ciclo, de lo contrario será un error

```
continue ;

var i: int = 0;
for(i = 0; i <= 2; i++){
     println(i);
    if (i==1){
        continue;
          /*me salte el resto de la iteración y continue en i == 3 */
    }
    println(i*5);
}
// Salida
/*
0
0
1
2
10
*/
```

5.17.3 Return

La sentencia return finaliza la ejecución de un método o función y puede especificar un valor para ser devuelto a quien llama a la función.

```
return ;
return <EXPRESION> ;

// Ejemplos
return 10 * 10;
return ;
return 1;
```

5.18 Vectores

Los vectores son una estructura de datos de tamaño fijo que pueden almacenar valores de forma limitada. **El lenguaje permitirá únicamente el uso de arreglos de una o dos dimensiones.**

Observaciones:

- La posición de cada vector será N-1. Por ejemplo, si se quiere acceder al primer valor de un vector se accede como vector[0].

5.18.1 Declaración de Vectores

En esta declaración, se indica por medio de una lista de valores separados por coma, los valores que tendrá el vector, en este caso el tamaño del vector será el de la misma cantidad de valores de la lista.

```
<MUTABILIDAD> <ID> : <TIPO> [ ] = [ <LISTAVALORES> ] ;
<MUTABILIDAD> <ID> : <TIPO> [ ] [ ] = [ <LISTAVALORES2>  ] ;

var vector1 : string [] = [“Hola”, “Mundo”];
const vector2 : int [][] = [ [1, 2], [3, 4] ];
```

5.18.2 Acceso a vectores

Expresion

Para acceder al valor de una expresión de un vector, se colocará el nombre del vector seguido del acceso.

```
<ID> [ <EXPRESION> ]

var vector3:string[] = [“Hola”, “Mundo”];
var valor3 : string = vector3[0]; // Almacena el valor “hola”

const vector4:int [][] = [ [1, 2], [3, 4] ];
const valor4:int = vector4[0][0]; // Almacena el valor 1
```

5.18.3 Asignación de Vectores

Para modificar el valor de una posición de un vector, se debe colocar el nombre del vector seguido del acceso y la expresión a asignar.

Observaciones:

- A una posición de un vector se le puede asignar el valor de otra posición de otro vector o del mismo vector.
- A una posición de un vector se le puede asignar el valor de una posición de una vector.
- Si el vector fue declarado como const no se puede realizar ninguna modificación al valor que posee. Se considera como error semántico.

```
<ID> [ <EXPRESION> ] = <EXPRESION> ;
<ID> [ <EXPRESION> ] [ <EXPRESION> ] = <EXPRESION> ;

var vector3:string[] = [“Hola”, “Mundo”];
vector3[0] = “OLC1”;
vector3[1] = “1er Semestre”;
```

5.19 Listas Dinámicas

Las listas dinámicas son una estructura de datos de tamaño variable que pueden almacenar valores de forma ilimitada.

5.19.1 Declaración de Listas

```
List< <TIPO> > <ID> = new List();

List<int> miLista = new List();
```

Observaciones:

- Las listas dinámicas siempre son variables por lo que no se coloca la palabra reservada var.
- Las listas dinámicas no se pueden declarar usando la palabra const.

5.19.2 Acceso a listas Expresion

Para acceder al valor de una expresión de una lista, se colocará el nombre de la lista seguido del acceso.

```
<ID> [ <EXPRESION> ]

List<int> miLista = new List();
miLista.append(1); // el método append se explicará más adelante

const valor:int = miLista[0]; // Almacena el valor 1
```

5.19.3 Asignación de Listas

Para modificar el valor de una posición de una lista, se debe colocar el nombre de la lista seguido de una expresión.

```
<ID> [ <EXPRESION> ] = <EXPRESION> ;

List<int> miLista = new List();
miLista.append(1);
miLista[0] = 2;
```

5.19.4 Append Instruccion

Para agregar valores a la lista, se utiliza la palabra reservada append, la cual agrega el elemento al final de la lista.

```
<ID>.append( <EXPRESION>);

List<int> miLista = new List();
miLista.append(1);
```

5.19.5 Remove

Para eliminar valores a la lista, se utiliza la palabra reservada remove, la cual recibe la posición del elemento a eliminar de la lista y lo retorna.

```
<ID>.remove( <EXPRESION>)

List<int> miLista = new List();
miLista.append(1);
miLista.append(2);
var miVar:int = miLista.remove(0); //almacena el valor de 1
```

5.20 Structs

Los structs son tipos de datos que permiten agrupar diferentes variables bajo un único nombre.

5.20.1 Declaración de Structs

Se definen mediante la palabra reservada "struct", seguida de un bloque de campos, cada uno con su tipo correspondiente, y finalmente un identificador.

```
struct {
<LISTA_STRUCT>
} <ID> ;


struct{
   nombre:string;
   edad:int;
}estudiante;
```

5.20.2 Instanciación de Struct

La instanciación de un struct se refiere al proceso de crear una variable utilizando la definición de la estructura.

```
<MUTABILIDAD> <ID> : <NOMBRE_STRUCT> = {
<VALORES_STRUCT> };

struct{
   nombre:string;
   edad:int;
}estudiante;

var e1:estudiante = { nombre: “prueba”, edad: 10 };
const e2: estudiante = { nombre: “prueba2”, edad: 20 };
```

5.20.3 Acceso de struct

El acceso a un struct se refiere a la capacidad de acceder a los campos individuales dentro de una estructura utilizando su nombre seguido de un operador de acceso, como el punto ".".

```
<ID> .<CAMPO> ;

struct{
   nombre:string;
   edad:int;
}estudiante;

var e1:estudiante = { nombre: “prueba”, edad: 10 };

                                                    HashMap
const edadEstudiante:int = e1.edad; // debe ser 10
```

5.20.4 Asignación de struct

Para poder modificar un struct, primero se debe tener una instancia del struct que deseas modificar. Luego, se debe acceder a cada uno de los campos del struct utilizando el operador de acceso punto ‘.’ y asignarles nuevos valores según sea necesario. Si la instancia fue declarada como const no se puede cambiar y se considera un error semántico.

```
<ID> . <CAMPO> = <EXPRESION> ;

struct{
   nombre:string;
   edad:int;
}estudiante;

var e1: estudiante = {nombre: “prueba”, edad: 10};
 e1.edad = 28;
```

5.21 Funciones

Una función es una subrutina de código que se identifica con un nombre, tipo y un conjunto de parámetros. Para este lenguaje las funciones serán declaradas definiendo primero su tipo, luego un identificador único, seguido de una lista de parámetros dentro de paréntesis (esta lista de parámetros puede estar vacía en el caso de que la función no utilice parámetros).

Cada parámetro debe estar compuesto por su tipo seguido de un identificador, para el caso de que sean varios parámetros se debe utilizar comas para separar cada parámetro y en el caso de que no se usen parámetros no se deberá incluir nada dentro de los paréntesis. Luego de definir la función y sus parámetros se declara el cuerpo de la función, el cual es un conjunto de instrucciones delimitadas por llaves {}.

Para las funciones es obligatorio que las mismas posean un valor de retorno que coincida con el tipo con el que se declaró la función, en caso de que no sea el mismo tipo o de que no venga un retorno dentro del cuerpo de la función debería lanzarse un error de tipo semántico.

```
<TIPO> <ID> ( <PARAMETROS> ) {
   <INSTRUCCIONES>
}

PARAMETROS → PARAMETROS , <ID> <TIPO>
                        | <ID><TIPO>

int conversion (size;int, tipo:string){
   if(tipo==”metro”){
      return size/3*3.281;
   } else{
     return -1;
   }
}
```

Cabe destacar que **no habrá sobrecarga de funciones y métodos** dentro de este lenguaje por lo que solo puede existir una función o método con el id declarado por lo que si se crea otra función o método con un id previamente utilizado esto debe de generar un error de tipo semántico.

5.22 Métodos

Un método también es una subrutina de código que se identifica con un tipo, nombre y un conjunto de parámetros, aunque a diferencia de las funciones estas subrutinas no deben de retornar un valor. Para este lenguaje los métodos serán declarados haciendo uso de la palabra reservada 'void', seguido de un identificador del método, seguido de una lista de parámetros dentro de paréntesis (esta lista de parámetros puede estar vacía en el caso de que la función no utilice parámetros).

Cada parámetro debe estar compuesto por su tipo seguido de un identificador, para el caso de que sean varios parámetros se debe utilizar comas para separar cada parámetro y en el caso de que no se usen parámetros no se deberá incluir nada dentro de los paréntesis. Luego de definir el método y sus parámetros se declara el cuerpo del método, el cual es un conjunto de instrucciones delimitadas por llaves {}.

void **<ID>** (**<PARAMETROS>**) {
  <INSTRUCCIONES>
}

PARAMETROS → PARAMETROS , **<ID> <TIPO>**
| **<ID><TIPO>**

Cabe destacar que **no habrá sobrecarga de funciones y métodos** dentro de este lenguaje por lo que solo puede existir una función o método con el id declarado por lo que si se crea otra función o método con un id previamente utilizado esto debe de generar un error de tipo semántico.

5.23 Llamadas

La llamada a una función específica la relación entre los parámetros reales y los formales y ejecuta la función. Los parámetros se asocian normalmente por posición, aunque, opcionalmente, también se pueden asociar por nombre.

La llamada a una función devuelve un resultado que ha de ser recogido, bien asignándole a una variable del tipo adecuado, bien integrándose en una expresión. La sintaxis de las llamadas de los métodos y funciones será la misma.

LLAMADA → **<ID>** (**<PARAMETROS_LLAMADA>**)
| **<ID>** ( )

PARAMETROS_LLAMADA → PARAMETROS_LLAMADA , **<EXPRESION>**
| **<EXPRESION>**

Observaciones:

- Al momento de ejecutar cualquier llamada, no se diferenciarán entre métodos y funciones, por lo tanto, podrá venir una función que retorne un valor como un método, pero la expresión retornada no se asignará a ninguna variable.
- Para la llamada a métodos como una instrucción se deberá de agregar el punto y coma (;) como una instrucción.

5.24 Función Println

Esta función nos permite imprimir expresiones y agrega un salto de línea al final del contenido

```
println ( <EXPRESION> ) ;

var arreglo : int [] = [1, 2, 3];

struct{
   nombre:string;
   edad:int;
}estudiante;

var e1: estudiante = {nombre: “prueba”, edad: 10};

println(“Hola mundo”);
println(arreglo);
println(arreglo[0]);
println(e1.edad);

//Salida esperada
/*
Hola mundo
[1, 2, 3]
```

```
1
10
*/
```

Expresion

5.25 Función round

Esta función recibe como parámetro un valor numérico. Permite redondear los números decimales según las siguientes reglas:

- Si el decimal es mayor o igual a 0.5, se aproxima al entero superior.
- Si el decimal es menor que 0.5, se aproxima al número inferior.

```
round ( <EXPRESION> )
```

5.26 Length

Esta función recibe como parámetro un vector, una lista o una cadena y devuelve el tamaño de este.

Observación:

- Si se usa para otro parámetro de tipo de dato no especificado, se considera un error semántico.

```
length ( <EXPRESION> )
```

5.27 ToString

Esta función permite convertir un valor de tipo numérico, caracter, bool o struct a texto.

Observación:

- Si se usa para otro parámetro de tipo de dato no especificado, se considera un error semántico.

```
toString ( <EXPRESION> )

// Ejemplo
struct{
   nombre:string;
   edad:int;
}estudiante;
```

```
var e1: estudiante = {nombre: “prueba”, edad: 10};
println(toString(e1));

const cad:string = toString(1); // almacena “1” como cadena
const cad2:string = toString(true); // almacena “true” como cadena

//Salida
// estudiante { nombre: “prueba”, edad: 10 }
```

5.28 Find

Esta función retorna true si la expresión existe dentro del vector o lista, caso contrario retorna false.

```
<ID>.Find ( <EXPRESION> )

// Ejemplo
var arr: int[] = [0, 1, 2];
const flag:bool = arr.Find(1); // almacena true
```

5.29 Función START_WITH

Para poder ejecutar todo el código generado dentro del lenguaje, se utilizará la sentencia START_WITH para poder indicar qué método o función es la que iniciará con la lógica del programa.

```
start_with <ID> ( ) ;
start_with <ID> ( <PARAMETROS> ) ;
```

6. Reportes

Los reportes son parte fundamental de javaCraft, ya que muestra de forma visual las herramientas utilizadas para realizar la ejecución del código. A continuación, se muestran ejemplos de estos reportes. Queda a discreción del estudiante el diseño de estos, solo se pide que sean totalmente legibles.

6.1 Tabla de errores

El reporte de errores debe contener la información suficiente para detectar y corregir errores en el código fuente.

#	Tipo	Descripción	Línea	Columna
1	Léxico	El carácter “$” no pertenece al lenguaje	5	3
2	Sintáctico	Se encontró Identificador y se esperaba Expresión.	6	3
3	Semántico	No se puede realizar resta entre CADENA y CADENA	8	10

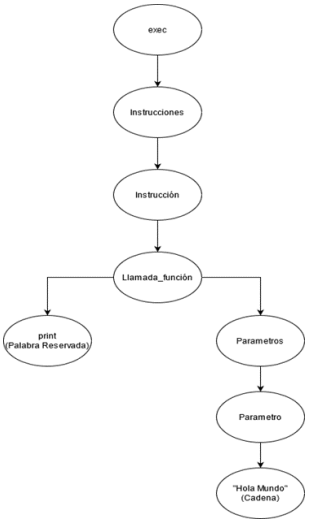
6.2 Tabla de Símbolos

Este reporte mostrará la tabla de símbolos después de la ejecución. Deberá mostrar las variables y arreglos declarados, así como su tipo, valor y toda la información que considere necesaria.

#	Id	Tipo	Tipo	Entorno	Valor	Línea	Columna
1	var	Variable	Entero	Funcion1	1	15	20
2	var2	Variable	Bool	Funcion2	true	20	13

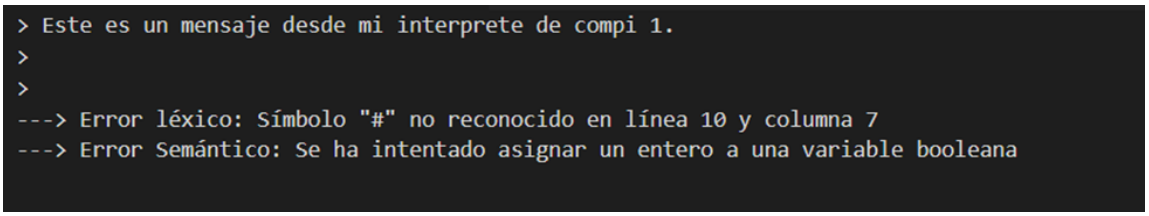
6.3 AST

Este reporte muestra el AST producido al analizar los archivos de entrada. Este debe de representarse como un grafo. Se deben mostrar los nodos que el estudiante considere necesarios para describir el flujo realizado para analizar e interpretar sus archivos de entrada.



6.4 Salidas en consola

La consola es el área de salida del intérprete. Por medio de esta herramienta se podrán visualizar las salidas generadas por la instrucción “println”, así como los errores léxicos, sintácticos y semánticos.



7. Restricciones

- La aplicación deberá de desarrollarse utilizando el lenguaje JAVA.
- Las herramientas para realizar los analizadores serán JFLEX y CUP.
- **El proyecto debe ser realizado de forma individual.**
- **Copias completas/parciales** de: código, gramática, etc. serán merecedoras de una **nota de 0 puntos**, los responsables serán reportados al catedrático de la sección y a la Escuela de Ciencias y Sistemas.
- La calificación será sobre el archivo ejecutable de su aplicación, de cumplir con este requerimiento **NO SE CALIFICARÁ**.
- La entrega debe ser realizada mediante UEDI.
- El nombre del repositorio de Github debe ser **OLC1_VJ24_#Carnet**
- Se debe agregar al auxiliar como colaborador del repositorio.
- Todas las salidas se deben de visualizar en la aplicación.

8. Entrega

El proyecto estará dividido en dos fases, a continuación se define como se van a realizar las entregas

- FASE 1
    - Funcionalidades a entregar
        - Interfaz gráfica
        - Comentarios
        - Tipos de datos
        - Secuencias de escape
        - Expresiones (Aritméticas, lógicas, relacionales y casteos)
        - Signos de agrupación
        - Declaración de variables
        - Asignación de variables

        - Acceso de variables
        - Sentencias de control
        - Sentencias cíclicas
        - Sentencias de transferencia (menos el return)
        - Función Println
        - Reporte Tabla de Símbolos
        - Reporte de Errores
    - Entregables
        - Código fuente de la aplicación
        - Archivo ejecutable de la aplicación
        - Archivo de gramática
    - **Fecha de entrega**
        - **Domingo 16 de junio hasta las 11:59 hrs**

- FASE 2
    - Funcionalidades a entregar
        - Vectores
        - Listas
        - Structs
        - Funciones
        - Métodos
        - Llamadas
        - Función Start_With
        - Todos los reportes
        - Resto de funciones nativas
    - Entregables
        - Código fuente de la aplicación
        - Archivo ejecutable de la aplicación
        - Archivo de gramática
    - **Fecha de entrega**
        - **Domingo 30 de junio hasta las 11:59 hrs**