Especificación Léxica

Estudiante	Escuela	Asignatura
Jerson Ernesto Chura Pacci jchurap@ulasalle.edu.pe	Carrera Profesional de Ingeniería de Software	Compiladores

Estudiante	Escuela	Asignatura
Andrea J. Ticona Mamani ifloresp@ulasalle.edu.pe	Carrera Profesional de Ingeniería de Software	Compiladores

Estudiante	Escuela	Asignatura
Iben Omar Flores Polanco jortizr@ulasalle.edu.pe	Carrera Profesional de Ingeniería de Software	Compiladores

Estudiante	Escuela	Asignatura
Joshua David Ortiz Rosas aticonam@ulasalle.edu.pe	Carrera Profesional de Ingeniería de Software	Compiladores

Informe	Tema	Duración
01	Especificación lexica	06 horas

Semestre académico	Fecha de inicio	Fecha de entrega
2025 - I	18/03/24	21/03/24

Índice

1. Introducción

El desarrollo de software ha evolucionado constantemente, buscando un equilibrio entre rendimiento, seguridad y facilidad de uso. Mientras que lenguajes como C ofrecen control y eficiencia, otros como Python y JavaScript priorizan la accesibilidad a costa del rendimiento. Sin embargo, la necesidad de un lenguaje que combine lo mejor de ambos mundos sigue siendo un desafío.

En este contexto nace FusionCod, un lenguaje diseñado no solo para programadores experimentados, sino también para estudiantes y nuevos desarrolladores que desean aprender programación sin enfrentar la complejidad de lenguajes de bajo nivel. FusionCode ofrece una sintaxis clara y estructurada, **facilitando la enseñanza de conceptos fundamentales** mientras mantiene un alto rendimiento y control sobre los recursos del sistema.

Este informe presenta **FusionCod**, su justificación en el contexto actual, sus objetivos y la propuesta que ofrece como una herramienta accesible y eficiente para la programación.

1.1. Justificación

El aprendizaje de programación y el desarrollo de software eficiente suelen estar en extremos opuestos. Lenguajes como Python facilitan el aprendizaje, pero sacrifican rendimiento, mientras que C y Rust ofrecen control a costa de una curva de aprendizaje más empinada. Esta dicotomía ha creado la necesidad de un lenguaje que sea accesible para principiantes sin perder la capacidad de optimización para sistemas más avanzados.

FusionCod busca cerrar esta brecha al proporcionar una sintaxis intuitiva con una gestión segura de memoria, evitando errores comunes como accesos indebidos y fugas de memoria. De esta forma, permite a los desarrolladores escribir código robusto sin preocuparse por problemas complejos de bajo nivel.

1.2. Objetivo

FusionCod tiene como propósito ofrecer un lenguaje versátil que combine eficiencia y accesibilidad. Su enfoque se basa en una estructura clara, un sistema de tipos seguro y una ejecución optimizada. Con esto, busca ser útil tanto para el desarrollo de software de alto rendimiento como para la enseñanza de programación a nuevos estudiantes.

2. Propuesta

FusionCod integra características de lenguajes como C, Rust y Python, equilibrando control, seguridad y simplicidad. Su compilador genera código eficiente sin necesidad de intervención manual en la gestión de recursos, permitiendo una experiencia de desarrollo fluida y accesible. Con ello, se presenta como una solución innovadora tanto para estudiantes como para programadores experimentados.

3. Especificación léxica

En esta sección, se detallará la especificación léxica de **FusionCod**, abordando los aspectos fundamentales que definen el conjunto de símbolos y las reglas de formación de los elementos del lenguaje. A continuación, se detalla cada punto:

3.1. Definición de los comentarios

La sintaxis de uno o varios comentarios en **FusionCod** (realizados de manera lineal) será de la siguiente manera:

Listing 1: Comentarios

```
# programa FusionCode, hola mundo

fn main () int {
   show("Hola mundo");
    return 0;
}
```

Los comentarios en **FusionCod** seran escritos después de poner #

3.2. Definición de los identificadores

Los identificadores (id) en **FusionCod** tendran las siguientes características:

- Los identificadores deben iniciar siempre con una letra, ya sea minuscula o mayúscula.

- Después, pueden incluir o: números (0-9), letras reconocidas en el abecedario (A-Z) o (a-z) o sub guiones (_).

- No pueden existir espacios en blanco en los identificadores.

- Los identificadores **no** pueden ser palabras reservadas del lenguaje.

- Toda sentencia, creación de un identidicador o asignación de un identificador debe terminar con ;.

Ejemplo:

Listing 2: Identificadores

```
fn main () int {
   n1 int = 10;
    nombre text = "Jerson";
    n2 float = 1.23;
}
```

3.3. Definición de las palabras clave

Las palabras clave en **FusionCod** serán las siguientes:

- **fn** : representa la definición de una función.

- **main** : representa el punto de entrada principal del programa.

- **void** : representa que una función no retorna ningún valor en especifico.

- **show** : representa el poder imprimir variables, números, funciones, etc.

- **return** : representa (en algunos casos) el retorno de un valor especifico.

- **stop** : representa (en bucles) el detenimiento del mismo.

- **for** : representa el inicio de una iteración definida.

- **if** : representa la evaluación de una condición, si se llega a cumplir se ejecutará un bloque de código.

- **or -** `||` : se utiliza para realizar una operación lógica de disyunción.

- **true** : representa un valor lógico de afirmación o veracidad. Indica que una condición o expresión lógica es verdadera.

- **false** : representa un valor lógico de negación o falsedad. Indica que una condición o expresión lógica no es verdadera.

- **and - &&** : se utiliza para realizar una operación lógica de conjunción.

- **else** : se utiliza en estructuras condicionales para especificar un bloque de código que se ejecuta si la condición del **if** no se cumple.

- **elif** : se usa como una condición alternativa en una estructura **if-else**.

- **while** : inicia un bucle que se ejecuta mientras se cumpla una condición.

- **int** : representa un tipo de dato para números enteros.

- **float** : representa un tipo de dato para números de punto flotante.

- **bool** : representa un tipo de dato para valores lógicos, es decir, **true** o **false**.

- **text** : representa un tipo de dato para cadenas de caracteres.

Listing 3: Algunas palabras clave

```
fn fibonacci_recursivo (n int) int {
   if (n <= 1) {
       return n;
    }
   return fibonacci_recursivo(n - 1) + fibonacci_recursivo(n - 2);
}

fn main () int {
   numero int = 5;
   show("La secuencia finbonacci en " + numero + " es: " + fibonacci_recursivo(numero));
   return 0;
}
```

3.4. Definición de los literales

Los literales en **FusionCode** son valores constantes utilizados directamente en el código fuente y representan datos específicos:

- **Literales numéricos** : representan valores numéricos y pueden ser enteros o de punto flotante. Por ejemplo, `42` o `3.14`.

- **Literales de cadenas** : representan secuencias de caracteres y se encierran entre comillas dobles. Por ejemplo, `"Hello, World!"` y `"1234"`.

- **Literales booleanos** : representan valores lógicos, específicamente `true` o `false`.

3.5. Definición de los operadores

En **FusionCode**, los operadores son símbolos que realizan operaciones sobre uno o más operandos. Estos operadores permiten la manipulación de datos y la implementación de lógica en el código. A continuación, se describen los principales operadores disponibles:

3.5.1. Operadores aritméticos

Se utilizan para realizar operaciones matemáticas básicas. Tales como:

`+` : suma dos operandos. Ejemplo: `a + b`.

`-` : resta el segundo operando del primero. Ejemplo: `a - b`.

`*` : multiplica dos operandos. Ejemplo: `a * b`.

`/` : divide el primer operando por el segundo. Ejemplo: `a / b`.

`%` : devuelve el residuo de la división entre dos operandos. Ejemplo: `a% b`.

3.5.2. Operadores de comparación

Se utilizan para comparar dos operandos y retornan un valor booleano. Tales como:

`==` : verifica si dos operandos son iguales. Ejemplo: `a == b`.

`>` : verifica si el primer operando es mayor que el segundo. Ejemplo: `a >b`.

`<` : verifica si el primer operando es menor que el segundo. Ejemplo: `a <b`.

`>=` : verifica si el primer operando es mayor o igual que el segundo. Ejemplo: `a >= b`.

`<=` : verifica si el primer operando es menor o igual que el segundo. Ejemplo: `a <= b`.

3.5.3. Operadores lógicos

Se utilizan para combinar expresiones booleanas. Tales como:

`and` : realiza una operación lógica de conjunción. Ejemplo: `a and b` u tambien `a && b`.

`or` : realiza una operación lógica de disyunción. Ejemplo: `a or b` u tambien `a || b`.

3.5.4. Operadores de asignación

Se utilizan para asignar valores a variables. Tales como:

`=` : asigna el valor del operando derecho al operando izquierdo. Ejemplo: `a = b`.

Ejemplos:

Listing 4: Algunos operadores

```
fn suma (a int, b int) int {
   return a + b;
}
fn main () int {
   a float;
   read(a);
   show(suma(a,a));
   return 0;
}
```

Listing 5: Algunos operadores

```
fn generar_patron(n int) void {
    i int = 1;
    while (i <= n) {
        j int = 1;
        while (j <= i) {
            if (j % 2 == 0) {
                show("* "); # Si j es par, muestra un asterisco
            } elif (j % 3 == 0) {
                show("# "); # Si j es mltiplo de 3, muestra un numeral
            } else {
                show(j + " "); # De lo contrario, muestra el nmero
            }
            j = j + 1;
        }
```

```
15          show("\n");
16          i = i + 1;
17      }
18  }
19  fn main() int {
20      num int;
21      show("Ingrese un nmero para generar el patrn:");
22      read(num);
23      if (num <= 0) {
24          show("El nmero debe ser mayor que 0.");
25      } else {
26          generar_patron(num);
27      }
28      return 0;
29  }
```

4. Expresiones regulares

Token	Expresión regular
funcion	fn
principal	main
pabierto	(
pcerrado	)
imprimir	show
coma	,
id	[A-Z-a-z]$^+$ | _([$A-Z$-$a-z$] | [$0-9$] | _)*
fsentencia	;
devolver	return
detener	stop
llaveabi	{
llavecerr	}
tentero	int
tcadena	text
tdecimal	float
tbooleano	bool
tvacio	void
si	if
y	&& | *and*
o	|| | *or*
sino	elif
entonces	else
mientras	while
para	for
suma	+
resta	-
mul	*
div	/
residuo	%
menorque	<
mayorque	>
menorigualque	<=
mayorigualque	>=
igual	=
igualbool	==
nentero	[0-9]$^+$ | $-[0-9]^+$
nflotante	[0-9]$^+$.[$0-9$]$^+$ | $-[0-9]^+$.[$0-9$]$^+$
ncadena	" (^") "
nbooleano	true | *false*
leer	read