Instituto Politécnico Nacional

Escuela Superior de Cómputo

Alumnos:

Lechuga Cervantes César Alejandro

Piña Ramírez Leo Yeudiel

Redondo González Omar

Unidad de Aprendizaje: Compiladores

Profesora: Albortante Morato Cecilia

Grupo: 5CM4

PROYECTO: Compilador PSeInt

Fecha: 3 de Julio del 2024

ÍNDICE

INTRODUCCIÓN

En este proyecto final tenemos la tarea de realizar el compilador de un lenguaje de programación a elección, teniendo la oportunidad de elegir uno existente o uno personalizado en donde creemos nuestras propias reglas de sintaxis.

Decidimos realizar el compilador para el lenguaje de programación en español que permite realizar pseudocódigo completamente funcional conocido como **“PseInt”**.

PSeInt

PseInt significa *PseudoIntérprete.*

PSeInt es una herramienta educativa diseñada para ayudar a las personas principiantes que recién comienzan a programar. Es un software que permite la creación y simulación de algoritmos escritos en un lenguaje de pseudocódigo, todos sabemos que para realizar un código primero debemos analizarlo mediante un pseudocódigo.

PSeInt utiliza un lenguaje de pseudocódigo que es fácil de entender y que está diseñado para parecerse al español. Esto permite a los principiantes concentrarse en la lógica de la programación sin preocuparse por la sintaxis compleja de los lenguajes de programación reales.

PSeInt es gratuito y está disponible para múltiples plataformas, incluyendo Windows, Linux y macOS.

Sintaxis de PSeInt

En este apartado describiremos las normas de sintaxis que contiene este lenguaje de programación y también las que implementaremos en este proyecto. Como se mencionó anteriormente, PSeInt cuenta con palabras reservadas y una interfaz completamente en español.

Este lenguaje no distingue mayúsculas de minúsculas, por lo que es bastante estricto en el manejo de sus palabras reservadas, normalmente empiezan por una mayúscula.

Declaración de un algoritmo

Algoritmo *NombreDelAlgoritmo*

Cuerpo (Declaraciones y Definiciones)

FinAlgoritmo

El código debe estar entre las palabras reservadas “Algoritmo” y “FinAlgoritmo”, el Algoritmo debe tener un nombre obligatorio.

Declaración de variables

Algoritmo Ejemplo

Definir var1, var2 Como Entero

Definir cadena Como Cadena

Definir flag Como Logico

FinAlgoritmo

Para declarar variables, la palabra reservada “Definir” debe colocarse antes del identificador (nombre) de la variable a declarar, posteriormente se escribe la palabra reservada “Como” y el tipo de variable, los tipos de datos existentes en este lenguaje se describirán más adelante.

Funciones

Funcion *NombreDeLaFuncion(param1, param2)*

Cuerpo de la función

Retornar *valor a retornar*

FinFuncion

Las funciones van dentro del algoritmo y cuentan con un nombre, así como parámetros de entrada y un valor de retorno en “Retornar”, la función irá dentro de las palabras reservadas “Función” y “FinFuncion”.

Estructura de control IF

Si *(condición*) Entonces

Instrucciones si la condición es verdadera

Sino

Instrucciones si la condición es falsa

FinSi

La forma de uso de una sentencia if es utilizando “Si” con la condición lógica entre paréntesis y antes de una palabra reservada “Entonces”, si la condición no se cumple el código pasa al else “Sino”, para cerrar esta sentencia utilizamos “FinSi”.

Estructura de control WHILE

Mientras *(condición)* Hacer

Instrucciones mientras la condición sea verdadera

FinMientras

La forma de uso de una sentencia While es utilizando “Mientras” con la condición lógica entre paréntesis y le sigue la palabra “Hacer”, si la condición no se cumple el ciclo termina, para cerrar esta sentencia utilizamos “FinMientras”.

Estructura de control FOR

Para i <- 1 Hasta 10 Con Paso 1 Hacer

// Instrucciones para cada iteración

FinPara

La forma de uso de una sentencia For, es utilizando “Para”, comenzamos con la condición de la variable de tipo “Entero” a iterar, seguido de un “Hasta” en donde se indica en una constante o variable el valor por el cual finalizará y finalmente “Con Paso” que indica la forma en que se incrementará o decrementará el iterador en cuestión. Para cerrar esta sentencia utilizamos “FinPara”.

Listado de Palabras reservadas

- Algoritmo
- Definir
- Leer
- Escribir
- FinAlgoritmo
- Como
- Entero
- Real
- Cadena
- Logico
- Funcion
- FinFuncion
- Si
- Entonces
- Sino
- FinSi
- Mientras
- Hacer
- FinMientras
- Para
- Hasta
- Con
- Paso
- FinPara
- Retornar
- Verdadero
- Falso
- Y
- O
- No

Listado de Operadores y Símbolos

- Suma: +
- Resta: -
- Multiplicación: *
- División: /
- Igual que: ==

- Asignación: =
- Mayor ó igual que: >=
- Menor ó igual que: <=
- Menor qué: <
- Mayor qué: >
- Paréntesis de apertura: (
- Paréntesis de cierre:)
- Separador: ,

Tipo de datos

Este lenguaje de programación cuenta con los siguientes tipos de datos:

- Real (float)
- Entero (int)
- Lógico (boolean)
- Cadena (string)

Existirán dos formas de definir variables con estos tipos de datos

Declaración sin tipo específico

Definir *variableEntera*

Definir *variableReal*

Definir *variableCadena*

Definir *variableLogica*

Cuando se define una variable y no se indica el tipo de dato, PSeInt determina el tipo según el valor asignado.

Declaración con tipo

Definir variableEntera Como Entero

Definir variableReal Como Real

Definir variableCadena Como Cadena

Definir variableLogica Como Lógico

Extensión de los archivos de PSeInt

La extensión del archivo siempre será .psc, el compilador tiene la opción de importar un archivo de este tipo para leer el contenido y finalmente compilarlo en nuestra interfaz.

Funciones del lenguaje

- Leer *NombreVariable*

Leer se encargará de obtener el valor que el usuario introduzca vía teclado, el valor se asigna a la variable indicada en *NombreVariable*

- Escribir *NombreVariable*|*"Contenido"*

Escribir se encargará de imprimir en la salida estándar del sistema, es decir en la pantalla lo que contenga la variable o si es que se especifica un contenido fijo en el mismo.

DESARROLLO

Explicaremos cada uno de los archivos incluidos en este proyecto, tanto archivos de compilador como el diseño de la interfaz empleada en esta. El compilador PSeInt se ejecuta en un navegador web (página web).

- Interfaz (index.html)

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Compilador Pseint</title>
  <link rel="stylesheet" href="estilos.css">
</head>
<body>
  <nav>
    <h1>Compilador Pseint</h1>
  </nav>
  <main>
```

```
    <div class="buttons">
      <button class="mainButton" onclick="compilar()">Compilar</button>
      <input type="file" name="" id="archivo">
    </div>
    <div class="inputs">
      <div class="programacion">
        <div class="intruccion">Escriba el código a compilar en el textarea</div>
        <textarea name="" id="progra" placeholder="Escriba su código así"
></textarea>
      </div>
      <div class="console">
        <div class="intruccion">Consola: </div>
        <textarea name="" id="conso" placeholder="Su respuesta es..."></textarea>
      </div>
    </div>
  </main>
</body>
<script src="analisisSintactico.js"></script>
<script src="analisisLexico.js"></script>
<script src="leerArchivo.js"></script>
<script src="https://cdn.jsdelivr.net/npm/sweetalert2@11"></script>
</html>
```

La interfaz fue desarrollada en HTML (Hyper Text Markup Language), visualizándose de la siguiente manera:

Figura 1. Interfaz del Compilador

- Botón Compilar: El botón invoca a la función compilar() que se encuentra en el fichero “analisislexico.js” el cual se describirá más adelante. Este botón es el encargado de comenzar con las fases principales del compilador en este caso del lenguaje PSeInt.

- Input Selector de Archivos: Este campo de entrada permite la lectura de un archivo a la vez y con extensión .psc como se indicó anteriormente. Cuando el archivo se lee correctamente, el contenido del archivo se copia en el input de nombre “progra”.

- Input proga (Escriba su código aquí): Este campo es de suma importancia, dado que es el único en donde podremos introducir nuestro código de tipo PSeInt, cuando el código quiera compilarse debe presionarse el botón de Compilar.

- Output Consola (Su respuesta es): Este campo es de salida, es decir solamente de visualización. Aquí se verá el resultado de la compilación e indicará si fue exitosa o si ocurrió un fallo describiendo el primer error encontrado.

○ analisisLexico.js

En este archivo, se encuentra el analizador Léxico del compilador, se definen las reglas de la gramática que compone el lenguaje PSeInt. Cada componente se describirá a continuación:

Definición de Tokens

```
//Definición de Tokens
const TokenType = {
  //Palabras reservadas
    //Cabecera y final del algoritmo
  ALGORITMO: 'ALGORITMO',
  FINALGORITMO: 'FINALGORITMO',

    //Declaración de variables
  DEFINIR: 'DEFINIR',
  COMO: 'COMO',

    //Funciones nativas para lectura y escritura
  LEER: 'LEER',
  ESCRIBIR: 'ESCRIBIR',

    //Para la declaración de una función
  FUNCION: 'FUNCION',
  FINFUNCION: 'FINFUNCION',
  RETORNAR: 'RETORNAR',
```

```
    //Estructura condicional If
  SI: 'SI',
  ENTONCES: 'ENTONCES',
  SINO: 'SINO',
  FINSI: 'FINSI',

    //Estructura ciclicas while o for
  MIENTRAS: 'MIENTRAS',
  HACER: 'HACER',
  FINMIENTRAS: 'FINMIENTRAS',
  PARA: 'PARA',
  HASTA: 'HASTA',
  CON: 'CON',
  PASO: 'PASO',
  FINPARA: 'FINPARA',

    //Valores lógicos
  VERDADERO: 'VERDADERO',
  FALSO: 'FALSO',

    //Tipo de datos
  ENTERO: 'ENTERO',
  REAL: 'REAL',
  CADENA: 'CADENA',
  LOGICO: 'LOGICO',

    //Operadores lógicos
  Y: 'Y',
  O: 'O',
  NO: 'NO',

  //Identificadores y literales
  IDENTIFICADOR: 'IDENTIFICADOR',  // Para identificar cualquier otro
identificador
  FLOTANTE: 'FLOTANTE',  // Para números
  NUMERO: 'NUMBER',  // Para números
  STRING: 'STRING',  // Para cadenas de texto

  //Operadores
  SUMA: 'SUMA',
  RESTA: 'RESTA',
  MULTIPLICACION: 'MULTIPLICACION',
  DIVISION: 'DIVISION',
```

```
  //Operadores Lógicos
  IGUAL: 'IGUAL',
  MENORQUE: 'MENORQUE',
  MAYORQUE: 'MAYORQUE',
  MAYORIGUAL: 'MAYORIGUAL',
  MENORIGUAL: 'MENORIGUAL',

  //Delimitadores
  IZQPAREN: 'IZQPAREN', //(
  RIGPAREN: 'RIGPAREN', //)
  ASIGNACION: 'ASIGNACION', //=
  COMA: 'COMA', //,

  //Fin de línea o Input
  SALTO: 'SALTO', //Fin de la línea
  EOF: 'EOF', //Fin de la línea
}
```

En esta variable TokenType no hay mucho que agregar, es un objeto en donde se almacenan todas las palabras reservadas y de la manera en la que se leerán ciertos caracteres especiales como el salto de línea, el fin de línea, coma, paréntesis, etc.

Definición de expresiones regulares de los tokens

```
const TokenPatterns = [
  { type: TokenType.ALGORITMO, pattern: /^Algoritmo\b/ },
  { type: TokenType.DEFINIR, pattern: /^Definir\b/ },
  { type: TokenType.LEER, pattern: /^Leer\b/ },
  { type: TokenType.ESCRIBIR, pattern: /^Escribir\b/ },
  { type: TokenType.FINALGORITMO, pattern: /^FinAlgoritmo\b/ },
  { type: TokenType.COMO, pattern: /^Como\b/ },
  { type: TokenType.ENTERO, pattern: /^Entero\b/ },
  { type: TokenType.REAL, pattern: /^Real\b/ },
  { type: TokenType.CADENA, pattern: /^Cadena\b/ },
  { type: TokenType.LOGICO, pattern: /^Logico\b/ },
  { type: TokenType.FUNCION, pattern: /^Funcion\b/ },
  { type: TokenType.FINFUNCION, pattern: /^FinFuncion\b/ },
  { type: TokenType.SI, pattern: /^Si\b/ },
  { type: TokenType.ENTONCES, pattern: /^Entonces\b/ },
  { type: TokenType.SINO, pattern: /^Sino\b/ },
  { type: TokenType.FINSI, pattern: /^FinSi\b/ },
  { type: TokenType.MIENTRAS, pattern: /^Mientras\b/ },
  { type: TokenType.HACER, pattern: /^Hacer\b/ },
```

```
  { type: TokenType.FINMIENTRAS, pattern: /^FinMientras\b/ },
  { type: TokenType.PARA, pattern: /^Para\b/ },
  { type: TokenType.HASTA, pattern: /^Hasta\b/ },
  { type: TokenType.CON, pattern: /^Con\b/ },
  { type: TokenType.PASO, pattern: /^Paso\b/ },
  { type: TokenType.FINPARA, pattern: /^FinPara\b/ },
  { type: TokenType.RETORNAR, pattern: /^Retornar\b/ },
  { type: TokenType.VERDADERO, pattern: /^Verdadero\b/ },
  { type: TokenType.FALSO, pattern: /^Falso\b/ },
  { type: TokenType.Y, pattern: /^Y\b/ },
  { type: TokenType.O, pattern: /^O\b/ },
  { type: TokenType.NO, pattern: /^No\b/ },
  { type: TokenType.SUMA, pattern: /^\+/},
  { type: TokenType.RESTA, pattern: /^\-/},
  { type: TokenType.MULTIPLICACION, pattern: /^\*/},
  { type: TokenType.DIVISION, pattern: /^\//},
  { type: TokenType.IGUAL, pattern: /^==/},
  { type: TokenType.ASIGNACION, pattern: /^=/},
  { type: TokenType.MAYORIGUAL, pattern: /^>=/},
  { type: TokenType.MENORIGUAL, pattern: /^<=/},
  { type: TokenType.MENORQUE, pattern: /^</},
  { type: TokenType.MAYORQUE, pattern: /^>/},
  { type: TokenType.IZQPAREN, pattern: /^\(/},
  { type: TokenType.RIGPAREN, pattern: /^\)/ },
  { type: TokenType.COMA, pattern: /^,/ },
  { type: TokenType.SALTO, pattern: /^\n/ },
  { type: TokenType.IDENTIFICADOR, pattern: /^[a-zA-Z_]\w*\b/ },  // Para otros
identificadores
  { type: TokenType.NUMERO, pattern: /^-?\d+(\.\d+)?\b/ },  // Para números
flotantes
  /* { type: TokenType.NUMERO, pattern: /^-?\d+\b/ },  // Para números enteros */
  { type: TokenType.STRING, pattern: /"([^"\\]|\\.)*"/ },  // Para cadenas de
texto
]
```

TokenPatterns es un objeto que almacena cada una de las expresiones regulares para cada token descrito en el objeto TokenType.

Puede visualizarse con facilidad que los símbolos de comparación como iqual qué o menor qué, tienen su expresión regular con dichos símbolos (== y < respectivamente)

Se puede observar por igual que una cadena de texto siempre debe comenzar y finalizar con comillas dobles, el contenido puede ser tanto de caracteres como de números.

Clase token

```
class Token{
  constructor(type, value, line) {
    this.type = type;
    this.value = value;
    this.line = line
  }

  toString() {
    return `Token (${this.type}, ${this.value}, ${this.line})`;
  }
}
```

Esta clase cuenta con un constructor, siempre recibe 3 parámetros, tipo, valor y línea. En tipo se invoca al tipo de token descrito arriba, el valor es la cadena de caracteres que contiene el token y la línea es el valor numérico de la línea donde se encuentra

La sobrecarga del método toString devuelve una cadena con estos valores en orden, seguido de “Token ”.

Clase Lexer

```
class Lexer {
  constructor(input) {
    this.input = input;
    this.pos = 0;
    this.line = 0;
  }

  advance() {
    this.pos++;
    if (this.pos >= this.input.length) {
      this.currentChar = null; // End of input
    } else {
      this.currentChar = this.input[this.pos];
    }
  }
```

```
  skipWhitespace() {
    while (this.currentChar !== null && /\s/.test(this.currentChar)) {
      this.advance();
    }
  }

  getNextToken(index) {
    if (this.pos >= this.input.length) {
      return new Token(TokenType.EOF, null, index);
    }
    this.currentChar = this.input[this.pos];
    this.skipWhitespace();
    const remainingInput = this.input.slice(this.pos);
    let value;
    /* console.log(remainingInput) */
    for (const { type, pattern } of TokenPatterns) {
      const match = remainingInput.match(pattern);
      if (match) {
        value = match[0];
        this.pos += value.length;
        const newToken = new Token(type, value, index);
        if (type == "SALTO")
          this.line++;
        return newToken;
      }
    }
    return new Token("UNDEFINED", value, index);
  }

}
```

La clase Lexer cuenta con 3 elementos, entrada, posición y línea. El constructor se encarga de almacenar el elemento de entrada que se recibe y almacenar un valor 0 por defecto en el elemento pos y line.

- advance(): función que se encarga de recorrer los caracteres incluidos en la cadena de entrada del compilador, si la posición supera el tamaño del input, devuelve “null” para finalizar con los analizadores.

- skipWhitespace(): función que como su nombre lo dice, permite avanzar en la posición de la entrada ignorando los espacios en blanco contenidos en esta.

- getNextToken(): Permite leer las palabras detectadas en los patrones o expresiones regulares para hallar el token correspondiente en el objeto TokenType.

Función Compilar

```
function compilar() {
  let input = document.getElementById("progra").value;
  const lineas = input.split('\n');
  console.log(lineas)

  let tokens = [];
  let token;
  //let i = 50;
  for (let i = 0; i < lineas.length; i++){
    const lexer = new Lexer(lineas[i].trim());
    do {
      token = lexer.getNextToken(i);
      if (token.type != TokenType.EOF) {
        console.log(token);
        tokens.push(token);
      }
      //i--;
    } while (token.type !== TokenType.EOF && token.type !== TokenType.UNDEFINED);
  }

  console.log(tokens)
  document.getElementById('conso').value = "Analizador lexico pasado"
  analizadorSintactico(tokens)
}
```

La función compilar es la más importante, comienza con el análisis léxico construyendo e identificando los tokens del archivo o de la entrada en cuestión, separa las líneas del input y comienza con la creación de tokens uno a uno almacenándose en el arreglo “tokens”, al finalizar pasa por el analizador sintáctico.

Función analizadorSintactico

```
function analizadorSintactico(tokens) {
  pos = 0;
  errores = []
```

```
  if (S(tokens)) {
    console.log(tokens[pos])
    document.getElementById('conso').value += `
Analizador sintactico pasado, y correcto`
    console.log("Si es correcto")
  } else {
    document.getElementById('conso').value += `
Analizador sintactico pasado, y NO correcto
    Se quedó en la palabra ${errores[errores.length - 1].value} en la línea
${errores[errores.length - 1].line + 1}`
    console.log(errores[errores.length - 1])
    console.log("No es correcto")
  }
}
```

La regla de raíz en S está compuesta de la siguiente manera:

S -> Algoritmo id CONTENIDO FinAlgoritmo

La función S está en el siguiente y último fichero, es decir analisisSintactico.js, Si no ha ocurrido ningún error, el compilador imprime el mensaje de “pasado y correcto”, si se encuentra cualquier error, este se muestra y en la línea donde se encuentra, significando que el analizador sintáctico y semántico no ha sido capaz de validar el código encontrado.

- analisisSintactico.js

Regla de producción S

```
//Gramatica
var pos = 0
var errores = []
//S -> Algoritmo id CONTENIDO FinAlgoritmo
function S(tokens) {
  let posIni = pos;
  if (TokenType.ALGORITMO == tokens[pos].type) {
    pos++;
    if (TokenType.IDENTIFICADOR == tokens[pos].type) {
      pos++;
      if (CONTENIDO(tokens))
        if (TokenType.FINALGORITMO == tokens[pos].type) {
          pos++;
```

```
          return 1;
        }
    }
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción CONTENIDO

```
//CONTENIDO -> INSTRUCCION CONTENIDO | e
function CONTENIDO(tokens) {
  let posIni = pos;

  //En caso de CONTENIDO -> INSTRUCCION CONTENIDO
  if (INSTRUCCION(tokens)) {
    if (CONTENIDO(tokens))
      return 1;
  }

  pos = posIni;
  //En caso de CONTENIDO -> e
  return 1;
}
```

Regla de producción INSTRUCCION

```
//INSTRUCCION -> DECLARACION | ASIGNACION | FUNCION | RETORNARVALOR | CONDICION |
CICLO | FUNCIONPREDER | INSTANCIAFUNCION
function INSTRUCCION(tokens) {
  let posIni = pos;

  //En caso de INSTRUCCION -> DECLARACION
  if (DECLARACION(tokens))
    return 1;

  //En caso de INSTRUCCION -> ASIGNACION
  if (ASIGNACION(tokens))
    return 1;

  //En caso de INSTRUCCION -> FUNCION
  if (FUNCION(tokens))
    return 1;
```

```
  //En caso de INSTRUCCION -> RETORNARVALOR
  if (RETORNARVALOR(tokens))
    return 1;

  //En caso de INSTRUCCION -> CONDICION
  if (CONDICION(tokens))
    return 1;

  //En caso de INSTRUCCION -> CICLO
  if (CICLO(tokens))
    return 1;

  //En caso de INSTRUCCION -> FUNCIONPREDER
  if (FUNCIONPREDER(tokens))
    return 1;

  //En caso de INSTRUCCION -> INSTANCIAFUNCION
  if (INSTANCIAFUNCION(tokens))
    return 1;

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción DECLARACION

```
//DECLARACION -> Definir REPETIRVARIABLE Como TIPODATO
function DECLARACION(tokens) {
  let posIni = pos;

  if (TokenType.DEFINIR == tokens[pos].type) {
    pos++;
    if (REPETIRVARIABLE(tokens))
      if (TokenType.COMO == tokens[pos].type) {
        pos++;
        if (TIPODATO(tokens))
          return 1;
      }
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción VARIABLES

```
//VARIABLES -> REPETIRVARIABLE | e
function VARIABLES(tokens) {
  let posIni = pos;

  //En caso de VARIABLES -> REPETIRVARIABLE
  if (REPETIRVARIABLE(tokens))
    return 1;

  pos = posIni;
  //En caso de VARIABLES -> e
  return 1;
}
```

Regla de producción REPETIRVARIABLE

```
//REPETIRVARIABLE -> id REPETIRVARIABLE'
function REPETIRVARIABLE(tokens) {
  let posIni = pos;

  if (TokenType.IDENTIFICADOR == tokens[pos].type) {
    pos++;
    if (REPETIRVARIABLE1(tokens))
      return 1;
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción REPETIRVARIABLE'

```
//REPETIRVARIABLE' -> , REPETIRVARIABLE | e
function REPETIRVARIABLE1(tokens) {
  let posIni = pos;

  //En caso de REPETIRVARIABLE1 -> , REPETIRVARIABLE
  if (TokenType.COMA == tokens[pos].type) {
    pos++;
    if (REPETIRVARIABLE(tokens))
      return 1;

    pos = posIni;
  }
```

```
  pos = posIni;
  //En caso de REPETIRVARIABLE1 -> e
  return 1;
}
```

Regla de producción TIPODATO

```
//TIPODATO -> Entero | Real | Cadena | Logico
function TIPODATO(tokens) {
  let posIni = pos

  //En caso de TIPODATO -> Entero
  if (TokenType.ENTERO == tokens[pos].type) {
    pos++;
    return 1;
  }

  pos = posIni;
  //En caso de TIPODATO -> Real
  if (TokenType.REAL == tokens[pos].type) {
    pos++;
    return 1;
  }

  pos = posIni;
  //En caso de TIPODATO -> Cadena
  if (TokenType.CADENA == tokens[pos].type) {
    pos++;
    return 1;
  }

  pos = posIni;
  //En caso de TIPODATO -> Logico
  if (TokenType.LOGICO == tokens[pos].type) {
    pos++;
    return 1;
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción ASIGNACION

```
//ASIGNACION -> id = VALOR
function ASIGNACION(tokens) {
  let posIni = pos

  if (TokenType.IDENTIFICADOR == tokens[pos].type) {
    pos++;
    if (TokenType.ASIGNACION == tokens[pos].type) {
      pos++;
      if (VALOR(tokens))
        return 1;
    }
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción VALOR

```
//VALOR -> Verdadero | Falso | String | Numero | Flotante | EXPRESION
function VALOR(tokens) {
  let posIni = pos;

  //En caso de VALOR -> Verdadero
  if (TokenType.VERDADERO == tokens[pos].type) {
    pos++;
    return 1;
  }

  pos = posIni;
  //En caso de VALOR -> Falso
  if (TokenType.FALSO == tokens[pos].type) {
    pos++;
    return 1;
  }

  pos = posIni;
  //En caso de VALOR -> String
  if (TokenType.STRING == tokens[pos].type) {
    pos++;
    return 1;
  }
```

```
  pos = posIni;
  //En caso de VALOR -> Numero
  if (TokenType.NUMERO == tokens[pos].type) {
    pos++;
    return 1;
  }

  pos = posIni;
  //En caso de VALOR -> Flotante
  if (TokenType.FLOTANTE == tokens[pos].type) {
    pos++;
    return 1;
  }

  pos = posIni;
  //En caso de VALOR -> EXPRESION
  if (EXPRESION(tokens))
    return 1;

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción EXPRESION

```
//EXPRESION -> TERMINO + EXPRESION | TERMINO - EXPRESION | TERMINO
function EXPRESION(tokens) {
  let posIni = pos;

  //En caso de EXPRESION -> TERMINO + EXPRESION
  if (TERMINO(tokens))
    if (TokenType.SUMA == tokens[pos].type) {
      pos++;
      if (EXPRESION(tokens))
        return 1;
    }

  pos = posIni;
  //En caso de EXPRESION -> TERMINO - EXPRESION
  if (TERMINO(tokens))
    if (TokenType.RESTA == tokens[pos].type) {
      pos++;
      if (EXPRESION(tokens))
        return 1;
```

```
    }

  pos = posIni;
  //En caso de EXPRESION -> TERMINO
  if(TERMINO(tokens))
    return 1;

  errores.push(tokens[pos]);
  return 0;
}

//TERMINO -> FACTOR * TERMINO | FACTOR / TERMINO | FACTOR
```

Regla de producción TERMINO

```
function TERMINO(tokens) {
  let posIni = pos;

  //En caso de TERMINO -> FACTOR * TERMINO
  if (FACTOR(tokens))
    if (TokenType.MULTIPLICACION == tokens[pos].type) {
      pos++;
      if (TERMINO(tokens))
        return 1;
    }

  pos = posIni;
  //En caso de TERMINO -> FACTOR / TERMINO
  if (FACTOR(tokens))
    if (TokenType.DIVISION == tokens[pos].type) {
      pos++;
      if (TERMINO(tokens))
        return 1;
    }

  pos = posIni;
  //En caso de TERMINO -> FACTOR
  if (FACTOR(tokens))
    return 1;

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción FACTOR

```
//FACTOR -> id | Numero | Flotante | (EXPRESION)
function FACTOR(tokens) {
  let posIni = pos;

  //En caso de FACTOR -> id
  if (TokenType.IDENTIFICADOR == tokens[pos].type) {
    pos++;
    return 1;
  }

  pos = posIni;
  //En caso de FACTOR -> Numero
  if (TokenType.NUMERO == tokens[pos].type) {
    pos++;
    return 1;
  }

  pos = posIni;
  //En caso de FACTOR -> Flotante
  if (TokenType.FLOTANTE == tokens[pos].type) {
    pos++;
    return 1;
  }


  pos = posIni;
  //En caso de FACTOR -> (EXPRESION)
  if (TokenType.IZQPAREN == tokens[pos].type) {
    pos++;
    if (EXPRESION(tokens))
      if (TokenType.RIGPAREN == tokens[pos].type) {
        pos++;
        return 1;
      }
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción FUNCION

```
//FUNCION -> Funcion id(VARIABLES) CONTENIDOFUNCION FinFuncion
function FUNCION(tokens) {
  let posIni = pos;

  if (TokenType.FUNCION == tokens[pos].type) {
    pos++;
    if (TokenType.IDENTIFICADOR == tokens[pos].type) {
      pos++;
      if (TokenType.IZQPAREN == tokens[pos].type) {
        pos++;
        if (VARIABLES(tokens))
          if (TokenType.RIGPAREN == tokens[pos].type) {
            pos++;
            console.log("Paso por aqui" + tokens[pos].type)
            if (CONTENIDOFUNCION(tokens))
              if (TokenType.FINFUNCION == tokens[pos].type) {
                pos++;
                return 1;
              }
          }
      }
    }
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción CONTENIDOFUNCION

```
//CONTENIDOFUNCION -> INSTRUCCIONFUNCION CONTENIDOFUNCION | e
function CONTENIDOFUNCION(tokens) {
  let posIni = pos;

  //En caso de CONTENIDOFUNCION -> INSTRUCCIONFUNCION CONTENIDOFUNCION
  if (INSTRUCCIONFUNCION(tokens)) {
    if (CONTENIDOFUNCION(tokens))
      return 1;

    pos = posIni;
  }

  pos = posIni;
```

```
  //En caso de CONTENIDOFUNCION -> e
  return 1;
}
```

Regla de producción INSTRUCCIONFUNCION

```
//INSTRUCCIONFUNCION -> DECLARACION | ASIGNACION | RETORNARVALOR | CONDICION |
CICLO | FUNCIONPREDER | INSTANCIAFUNCION
function INSTRUCCIONFUNCION(tokens) {
  let posIni = pos;

  //En caso de INSTRUCCIONFUNCION -> DECLARACION
  if (DECLARACION(tokens))
    return 1;

  pos = posIni;
  //En caso de INSTRUCCIONFUNCION -> ASIGNACION
  if (ASIGNACION(tokens))
    return 1;

  pos = posIni;
  //En caso de INSTRUCCIONFUNCION -> RETORNARVALOR
  if (RETORNARVALOR(tokens))
    return 1;


  pos = posIni;
  //En caso de INSTRUCCIONFUNCION -> CONDICION
  if (CONDICION(tokens))
    return 1;

  pos = posIni;
  //En caso de INSTRUCCIONFUNCION -> CICLO
  if (CICLO(tokens))
    return 1;

  pos = posIni;
  //En caso de INSTRUCCIONFUNCION -> FUNCIONPREDER
  if (FUNCIONPREDER(tokens))
    return 1;

  pos = posIni;
  //En caso de INSTRUCCIONFUNCION -> INSTANCIAFUNCION
  if (INSTANCIAFUNCION(tokens))
```

```
    return 1;

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción RETORNAR VALOR

```
//RETORNARVALOR -> Retornar id
function RETORNARVALOR(tokens) {
  let posIni = pos;

  if (TokenType.RETORNAR == tokens[pos].type) {
    pos++;
    if (TokenType.IDENTIFICADOR == tokens[pos].type) {
      pos++;
      return 1;
    }
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción CONDICION

```
//CONDICION -> Si (EXPRESIONCONDICION) Entonces CONTENIDOFUNCION Sino
CONTENIDOFUNCION FinSi | Si (EXPRESIONCONDICION) ENTONCES CONTENIDOFUNCION FinSi
function CONDICION(tokens) {
  let posIni = pos;

  //En caso de CONDICION -> Si (EXPRESIONCONDICION) Entonces CONTENIDOFUNCION
Sino CONTENIDOFUNCION FinSi
  if (TokenType.SI == tokens[pos].type) {
    pos++;
    if (TokenType.IZQPAREN == tokens[pos].type) {
      pos++;
      if (EXPRESIONCONDICION(tokens))
        if (TokenType.RIGPAREN == tokens[pos].type) {
          pos++;
          if (TokenType.ENTONCES == tokens[pos].type) {
            pos++;
            if (CONTENIDOFUNCION(tokens))
              if (TokenType.SINO == tokens[pos].type) {
```

```
                pos++;
                if (CONTENIDOFUNCION(tokens))
                  if (TokenType.FINSI == tokens[pos].type) {
                    pos++;
                    return 1;
                  }
              }
            }
          }
    }
  }

  pos = posIni;
  //En caso de CONDICION -> Si (EXPRESIONCONDICION) Entonces CONTENIDOFUNCION
FinSi
  if (TokenType.SI == tokens[pos].type) {
    pos++;
    if (TokenType.IZQPAREN == tokens[pos].type) {
      pos++;
      if (EXPRESIONCONDICION(tokens))
        if (TokenType.RIGPAREN == tokens[pos].type) {
          pos++;
          if (TokenType.ENTONCES == tokens[pos].type) {
            pos++;
            if (CONTENIDOFUNCION(tokens))
              if (TokenType.FINSI == tokens[pos].type) {
                pos++;
                return 1;
              }
          }
        }
    }
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción EXPRESIONCONDICION

```
//EXPRESIONCONDICION -> EXPRESION' < EXPRESION' | EXPRESION' > EXPRESION' |
EXPRESION' <= EXPRESION' | EXPRESION' >= EXPRESION' | EXPRESION' == EXPRESION' |
NO EXPRESION' | EXPRESION' O EXPRESION' | EXPRESION' Y EXPRESION' | EXPRESION'
function EXPRESIONCONDICION(tokens) {
  let posIni = pos;
```

```
//En caso de EXPRESIONCONDICION -> EXPRESION' < EXPRESION'
if (EXPRESION1(tokens))
  if (TokenType.MENORQUE == tokens[pos].type) {
    pos++;
    if (EXPRESION1(tokens))
      return 1;
  }

pos = posIni;
//En caso de EXPRESIONCONDICION -> EXPRESION' > EXPRESION'
if (EXPRESION1(tokens))
  if (TokenType.MAYORQUE == tokens[pos].type) {
    pos++;
    if (EXPRESION1(tokens))
      return 1;
  }

pos = posIni;
//En caso de EXPRESIONCONDICION -> EXPRESION' >= EXPRESION'
if (EXPRESION1(tokens))
  if (TokenType.MAYORIGUAL == tokens[pos].type) {
    pos++;
    if (EXPRESION1(tokens))
      return 1;
  }

pos = posIni;
//En caso de EXPRESIONCONDICION -> EXPRESION' <= EXPRESION'
if (EXPRESION1(tokens))
  if (TokenType.MENORIGUAL == tokens[pos].type) {
    pos++;
    if (EXPRESION1(tokens))
      return 1;
  }

pos = posIni;
//En caso de EXPRESIONCONDICION -> EXPRESION' == EXPRESION'
if (EXPRESION1(tokens))
  if (TokenType.IGUAL == tokens[pos].type) {
    pos++;
    if (EXPRESION1(tokens))
      return 1;
  }
```

```
  pos = posIni;
  //En caso de EXPRESIONCONDICION -> NO EXPRESION'
  if (TokenType.NO == tokens[pos].type) {
    pos++;
    if (EXPRESION1(tokens))
      return 1;
  }

  pos = posIni;
  //En caso de EXPRESIONCONDICION -> EXPRESION' O EXPRESION'
  if (EXPRESION1(tokens))
    if (TokenType.O == tokens[pos].type) {
      pos++;
      if (EXPRESION1(tokens))
        return 1;
    }

  pos = posIni;
  //En caso de EXPRESIONCONDICION -> EXPRESION' Y EXPRESION'
  if (EXPRESION1(tokens))
    if (TokenType.Y == tokens[pos].type) {
      pos++;
      if (EXPRESION1(tokens))
        return 1;
    }

  pos = posIni;
  //En caso de EXPRESIONCONDICION -> EXPRESION'
  if (EXPRESION1(tokens))
    return 1;

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción EXPRESION'

```
//EXPRESION' -> TERMINO' + EXPRESION' | TERMINO' - EXPRESION' | TERMINO'
function EXPRESION1(tokens) {
  let posIni = pos;

  //En caso de EXPRESION' -> TERMINO' + EXPRESION'
  if (TERMINO1(tokens))
    if (TokenType.SUMA == tokens[pos].type) {
      pos++;
```

```
      if (EXPRESION1(tokens))
        return 1;
    }

  pos = posIni;
  //En caso de EXPRESION' -> TERMINO' - EXPRESION'
  if (TERMINO1(tokens))
    if (TokenType.RESTA == tokens[pos].type) {
      pos++;
      if (EXPRESION1(tokens))
        return 1;
    }

  pos = posIni;
  //En caso de EXPRESION' -> TERMINO'
  if (TERMINO1(tokens))
    return 1;

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción TERMINO'

```
//TERMINO' -> FACTOR' * TERMINO' | FACTOR' / TERMINO' | FACTOR'
function TERMINO1(tokens) {
  let posIni = pos;

  //En caso de TERMINO' -> FACTOR' * TERMINO'
  if (FACTOR1(tokens))
    if (TokenType.MULTIPLICACION == tokens[pos].type) {
      pos++;
      if (TERMINO1(tokens))
        return 1;
    }

  pos = posIni;
  //En caso de TERMINO' -> FACTOR' / TERMINO'
  if (FACTOR1(tokens))
    if (TokenType.DIVISION == tokens[pos].type) {
      pos++;
      if (TERMINO1(tokens))
        return 1;
    }
```

```
  pos = posIni;
  //En caso de TERMINO' -> FACTOR'
  if (FACTOR1(tokens))
    return 1;

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción FACTOR'

```
//FACTOR' -> id | Numero | (EXPRESION') | Verdadero | Falso
function FACTOR1(tokens) {
  let posIni = pos;

  //En caso de FACTOR' -> id
  if (TokenType.IDENTIFICADOR == tokens[pos].type) {
    pos++;
    return 1;
  }

  pos = posIni;
  //En caso de FACTOR' -> Numero
  if (TokenType.NUMERO == tokens[pos].type) {
    pos++;
    return 1;
  }

  pos = posIni;
  //En caso de FACTOR' -> (EXPRESION')
  if (TokenType.IZQPAREN == tokens[pos].type) {
    pos++;
    if (EXPRESION1(tokens))
      if (TokenType.RIGPAREN == tokens[pos].type) {
        pos++;
        return 1;
      }
  }

  pos = posIni;
  //En caso de FACTOR' -> Verdadero
  if (TokenType.VERDADERO == tokens[pos].type) {
    pos++;
    return 1;
  }
```

```
  pos = posIni;
  //En caso de FACTOR' -> Falso
  if (TokenType.FALSO == tokens[pos].type) {
    pos++;
    return 1;
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción CICLO

```
//CICLO -> Mientras (EXPRESION) Hacer CONTENIDOFUNCION FinMientras
function CICLO(tokens) {
  let posIni = pos;

  if (TokenType.MIENTRAS == tokens[pos].type) {
    pos++;
    if (TokenType.IZQPAREN == tokens[pos].type) {
      pos++;
      if (EXPRESION(tokens))
        if (TokenType.RIGPAREN == tokens[pos].type) {
          pos++;
          if (TokenType.HACER == tokens[pos].type) {
            pos++;
            if (CONTENIDOFUNCION(tokens))
              if (TokenType.FINMIENTRAS == tokens[pos].type) {
                pos++;
                return 1;
              }
          }
        }
    }
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción FUNCIONPREDER

```
//FUNCIONPREDER -> Leer id | Escribir id
function FUNCIONPREDER(tokens) {
  let posIni = pos;

  //En caso de FUNCIONPREDER -> Leer id
  if (TokenType.LEER == tokens[pos].type) {
    pos++;
    if (TokenType.IDENTIFICADOR == tokens[pos].type) {
      pos++;
      return 1;
    }
  }

  pos = posIni;
  //En caso de FUNCIONPREDER -> Escribir  id
  if (TokenType.ESCRIBIR == tokens[pos].type) {
    pos++;
    if (TokenType.IDENTIFICADOR == tokens[pos].type) {
      pos++;
      return 1;
    }
  }

  errores.push(tokens[pos]);
  return 0;
}
```

Regla de producción INSTANCIAFUNCION

```
//INSTANCIAFUNCION -> id (VARIABLES)
function INSTANCIAFUNCION(tokens) {
  let posIni = pos;

  if (TokenType.IDENTIFICADOR == tokens[pos].type) {
    pos++;
    if (TokenType.IZQPAREN == tokens[pos].type) {
      pos++;
      if (VARIABLES(tokens))
        if (TokenType.RIGPAREN == tokens[pos].type) {
          pos++;
          return 1;
        }
    }
```

```
  }

  errores.push(tokens[pos]);
  return 0;
}
```

CONCLUSIONES

Desarrollar compiladores para lenguajes de alto nivel es una tarea compleja y técnicamente exigente de realizar, dado que ciertos lenguajes cuentan con sintaxis y especificaciones rígidas y precisas para ser visto como código funcional y correcto, pero un compilador no deja de ofrecer ciertos beneficios importantes que justifican el esfuerzo de los códigos realizados.

Los lenguajes de alto nivel proporcionan una mayor abstracción en comparación con el código máquina o los lenguajes de ensamblador, lo que permite a los programadores comprender el código que realizan con mayor facilidad, con el fin de resolver problemas computables a un nivel más conceptual finalmente facilitando el mantenimiento del código a un futuro.

Los programas escritos en lenguajes de alto nivel pueden ser compilados en diferentes sistemas operativos y arquitecturas de hardware sin necesidad de cambiar el código fuente, siempre que exista un compilador adecuado para cada plataforma.

Los compiladores realizan diversas verificaciones estáticas, como la comprobación de tipos y la detección de errores sintácticos y semánticos, antes de que el programa sea ejecutado, lo que reduce la posibilidad de errores en tiempo de ejecución.

El desarrollo de compiladores fomenta la innovación en la creación de nuevos lenguajes y paradigmas de programación, permitiendo la evolución de la tecnología de software, así como incluir mecanismos para soportar la evolución del lenguaje, asegurando que el código antiguo siga funcionando en versiones más nuevas del lenguaje.