



**2024**

***™***

**SUG - Compilador**

Buchieri Giovanni

Vázquez Santiago

Guzman A. Uriel

Taller de Diseño de Software

Informe de Implementación

**Introducción**

En el desarrollo del proyecto en cuestión, compilador de mini-lenguaje, tuvimos que ordenar muchas ideas referidas a la implantación del mismo, tanto para la lectura del archivo (parte lexicográfica), como así también en la parte sintáctica (parser) y en el parte de la semántica (interprete/compilador); para ello optamos por mantener las siguientes convenciones:

## **Análisis Léxico**

En la definición de tipos de datos, los cuales puede enteros o booleanos, se usan los términos *int* y *bool* para su representación, además, se cuenta con el tipo *void* para darle la posibilidad a una función de no devolver nada, la cual se podría tomar como un procedimiento.

**Tipos de datos**

Para la notación de los términos numéricos se emplea la idea de definición recursiva *{digito}+,* donde digito es un entero comprendido en el rango [0-9], estos van a trabajar con las operaciones arrítmicas clásicas suma (+), resta (-), producto (\*), división (/) y modulo (%), su representación original simplifica la lectura y comprensión.

Para la notación de los valores booleanos se usan las palabras reservadas *true* y *false*, las cuales aparte de reflejar la idea, son útiles para la comprensión. Este tipo de dato, a su vez cuenta con las operaciones lógicas binarias de and (&&) y or (||), y con la operación unaria not (!), habiéndose elegido dichos símbolos por razones similares a las de los valores numéricos.

**Identificadores**

Para la notación de las variables o nombres de los procedimientos se emplea los identificadores (id), denotados de la forma recursiva *{letra}({letra}|{digito}) \*,* donde letra es un carácter del código ASCII comprendido en el conjunto [a-z] ∪ [A-Z].

**Operaciones de comparación**

Para el manejo de las estructuras y flujos del programa, se definieron las operaciones: asignación (=), valida para ambos tipos de datos mencionados, tendrá el tipo de los valores que se apliquen sobre ella; igualdad (==), válida para ambos tipos, contendrá el valor booleano correspondiente a la evaluación correspondiente; mayor que (>) y menor que (<), ambos validos solo para los valores enteros, y al igual que la igualdad, contendrá el valor de su evaluación.

**Estructuras**

En cuanto a estructura se refiere, como se mencionó anteriormente se cuenta con sentencias de control, estas se pueden clasificar en condicionales (*if\_then\_else*), iterativas (*while*) y lo que viene siendo un bloque principal (*Program*), el cual a su vez cuenta con una función *main,* la cual funciona como bloque de ejecución, la cual se define como una función y puede tener diferentes tipos de retorno, además de una sentencia de retorno (en caso de que el tipo no sea *void*) la cual denotamos *return*. Además de *main*, se definen las funciones en general, con la misma idea que *main*, con un *id* y tipo de rotorno.

**Complementos**

Para la incorporación de elementos externos al archivo corriente, se opto por definir la palabra reservada *extern* como puente o direccionador para los mismos.

**Comentarios**

Para permitir realizar comentarios, anotaciones o aclaraciones de código, definimos los comentarios de una línea (//) y de varias líneas (/\*\*/), dejando evidente su parecido al de otros lenguajes, lo que facilita su uso.

**Delimitadores**

Para una correcta lectura, se emplean ( ) como delimitadores, para saber que acción debe tener precedencia en las operaciones numéricas/booleanas y también, como encapsulante de sección de parámetros en una función. Otra cuestión a destacar es el uso de { } como delimitadores de bloque, útiles para saber donde empieza y donde termina una secuencia de sentencias. Sin embargo, las llaves solo son empleadas en bloque, para diferenciar cada sentencia particular se emplea ‘*;*’, y para la separación de parámetro se emplea ‘,’.

En consecuencia, de lo mencionado, cabe destacar que puntos (.), espacios en blanco ( ) y saltos de línea (\n) se podrán emplear, pero siendo ignorados. Cualquier otro símbolo que esté mesclado con los previos o en caso de simple aparición, se producirá un error por ser una secuencia invalida para el lexer.

Para el posterior manejo de los símbolos, se devolverá un token asociado a cada uno de los mencionados anteriormente, los cuales serán trabajados y descriptos en el análisis sintáctico.

## **Análisis Sintáctico**

Para la manipulación de los tokens rescatados del análisis léxico, optamos que estos sean las etiquetas (nombre) que decoren los nodos de un árbol binario. En dicho nodo almacenamos un símbolo (estructura con los campos: identificador, valor, tipo, tamaño, etc.), y dos referencias, una para el hijo izquierdo y otra para el hijo derecho. De esta manera, al ir leyendo el archivo, se va creando el árbol sintáctico correspondiente, el cual es útil en la evaluación realizada en el análisis semántico. De forma paralela, al mientras se crea el árbol, cada símbolo creado que corresponda a una variable o una función/procedimiento, se ira agregando a una tabla de símbolos.

La tabla de símbolos se creó a partir de la idea recursiva de usar una lista enlazada, mientras que el árbol sintáctico se basa en un árbol binario con dos referencias a sus nodos sucesores.

## **Análisis Semántico**