



**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 retorno.

**Complementos**

Para la incorporación de elementos externos al archivo corriente, se optó 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 dónde 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 en el 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 empleamos en la evaluación durante el análisis semántico. De forma paralela, 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.

El árbol sintáctico se basa en un árbol binario con dos referencias a sus nodos sucesores, mientras que la tabla de símbolos se creó a partir de la idea recursiva de usar una lista enlazada. La definición de dichas estructuras se realiza en los archivos *AST.c* y *symbol.c* presentes en el directorio *src*.

Para la creación del Árbol Sintáctico Abstracto (*AST*) y su manejo general, se emplearon los siguientes métodos:

// crear un AST tomando el símbolo, un nodo izquierdo y un nodo derecho

struct AST\* createTree (Tsymbol\* *symbol*, struct AST \**l*, struct AST \**r*);

// construye la table de símbolos una vez que el AST fue creado

void createTable(AST\* *ar*);

// elimina todas las referencias de la tabla de símbolos asi como los nodos del AST

void elimArbol (AST\* *tree*);

// toma el AST y lo recorre tipando y añadiendo los valores correspondientes a las

// evaluaciones a los nodos (usado en la creación del interprete)

void evaluate (AST\* *ar*);

// plasman el AST en un archivo .dot para su visualización y verificar correcta creación

void showTreeDot (AST\* *tree*,FILE\* *file*);

void printDot (AST\* *tree*, const char\* *filename*);

Para la creación de la Tabla de Símbolos y realizar acciones sobre ella se emplean las siguientes funciones:

// crea un símbolo tomando el nombre del mismo, el tipo de valor que contiene,

// el tamaño que debe ocupar y la línea donde se encuentra en el código

struct Tsymbol \* CreateSymbol (char \**name*, enum TYPES *type*, int *size*, int *line*);

// Devuelve un puntero a la entrada de la tabla de símbolos para la variable,

// devuelve NULL en caso contrario

struct Tsymbol \*Lookup (char \* *name*);

// busca una subtabla por su número de identificación, devuelve el puntero

// correspondiente o NULL en caso contrario

struct Tsymbol \*LookupTable(int *size*);

// realiza la misma búsqueda que lookup pero una subtabla dentro de la principal

struct Tsymbol \*LookupInTable(char \* *name*,Tsymbol\* *symTabla*);

// agrega un símbolo dado a la tabla, verifica que este ya no se encuentre allí

void Install (Tsymbol \**symbol*);

// elimina todas las referencias de la tabla de símbolos

void DeleteList ();

// muestral el contenido de la tabla, útil para verificar que se evaluó de forma correcta

void printable ();

// agrega una subtabla perteneciente a una función dada a la tabla de símbolos principal

void InstallTable(Tsymbol \**symbol*,Tsymbol \**symTabla*);

Para facilitar la búsqueda y acceso a las subtablas pertenecientes a las declaraciones de funciones, optamos, como se expresa en la descripción de la función lookupTable, por agregarle un valor numérico único a cada bloque de código. Esto lo hacemos de forma global en el momento de creación de cada bloque, lo que nos permite mayor facilidad de búsqueda y movilidad entre los distintos ámbitos del programa en el análisis semántico.

En cuanto a los tokens empleados, se encuentran categorizados según la función que cumplen, a diferencia de la etapa léxica donde se agruparon según su significado o tipo con el que trabajaba. Las distinción mencionada quedo de la siguiente manera: *declaraciones, tipos de datos, símbolos, y palabras reservadas.*

En *declaraciones* se encuentra solamente el token *ID*, siendo este usado para representar el nombre de la variable o función correspondiente. En *tipos de datos* se encuentran los tokens: *INT, TTRUE, TFALSE, TYPE\_INT, TYPE\_BOOL, TYPE\_VOID*, siendo los que definirán los valores de los datos y tipos de retorno en las funciones. En *símbolos* se encuentran los tokens: *TMAS, TPOR, TMENOS, TDIVISION, TRESTO, ASIGNACION, TPAR\_OP, TPAR\_CL, TLLAVE\_OP, TLLAVE\_CL, OR, AND, NOT, MAYORQUE, MENORQUE* y *EQ*, usados en operaciones aritméticas como booleanas, además de estar presentes como delimitadores en los bloques de sentencias. Y en *palabras reservadas* se tienen los tokens: *PROGRAM, EXTERN, THEN, IF, ELSE, WHILE, RETURN* y *MAIN*, empleados como declaradores de bloques de código, importaciones y programa actual.

Cada token representa un tipo de evaluación diferente, pero en particular se tiene que los tokens: *ID, INT, TTRUE* y *TFALSE* son de tipo símbolo, con lo que al evaluarlos se deben tomar como dicho tipo de dato. Además, cada uno es empleado de manera particular según la regla gramatical que corresponda.

Para mantener coherencia y evitar conflictos de lectura, para algunos tokens definimos precedencia, esta viene dada de la siguiente manera: *OR, AND* >> *EQ* >> *MAYORQUE, MENORQUE* >> *TMAS, TMENOS* >> *TPOR, TDIVISION, TRESTO* >> *NOT*. Leyéndose de izquierda a derecha, el operador con mas precedencia entre los mencionados es el *NOT* por ser operador unario, además este es el único que le definimos asociatividad a derecha, esto por cómo se debe evaluar.

**Gramática**

prog: *PROGRAM* *TLLAVE\_OP* *list\_func* *main* *TLLAVE\_CL*

main: TYPE\_BOOL MAIN TPAR\_OP TPAR\_CL TLLAVE\_OP list\_decls list\_sents TLLAVE\_CL

| TYPE\_INT MAIN TPAR\_OP TPAR\_CL TLLAVE\_OP list\_decls list\_sents TLLAVE\_CL

| TYPE\_VOID MAIN TPAR\_OP TPAR\_CL TLLAVE\_OP list\_decls list\_sents TLLAVE\_CL

declaracion: TYPE\_INT ID ';' | TYPE\_BOOL ID ';'

list\_decls: ∅ | *list\_decls* *declaracion*

asignacion: *ID* *ASIGNACION* *expr* ';'

sentencia: asignacion

| retorno

| if\_else

| while

| call\_func ';'

valor: INT

| ID

| TMENOS INT

| TTRUE

| TFALSE

expr: valor

| call\_func

| NOT expr

| TPAR\_OP expr TPAR\_CL

| expr TMAS expr

| expr TMENOS expr

| expr TPOR expr

| expr TDIVISION expr

| expr TRESTO expr

| expr MAYORQUE expr

| expr MENORQUE expr

| expr EQ expr

| expr AND expr

| expr OR expr

list\_sents: ∅ | *list\_sents* *sentencia*

dec\_parametro : TYPE\_INT ID | TYPE\_BOOL ID

parametros: ∅ | dec\_parametro | dec\_parametro ',' parametros

list\_func: ∅ | list\_func declare\_funcion

argumento: ∅ | expr | argumento ',' expr

call\_func : *ID* *TPAR\_OP* *argumento* *TPAR\_CL*

declare\_funcion:

| TYPE\_INT ID TPAR\_OP parametros TPAR\_CL TLLAVE\_OP list\_decls list\_sents TLLAVE\_CL

| TYPE\_BOOL ID TPAR\_OP parametros TPAR\_CL TLLAVE\_OP list\_decls list\_sents TLLAVE\_CL

| TYPE\_VOID ID TPAR\_OP parametros TPAR\_CL TLLAVE\_OP list\_decls list\_sents TLLAVE\_CL

| TYPE\_INT ID TPAR\_OP parametros TPAR\_CL EXTERN ';'

| TYPE\_BOOL ID TPAR\_OP parametros TPAR\_CL EXTERN ';'

| TYPE\_VOID ID TPAR\_OP parametros TPAR\_CL EXTERN ';'

retorno: *RETURN* *expr* ';'

if\_else: IF TPAR\_OP expr TPAR\_CL THEN TLLAVE\_OP list\_sents TLLAVE\_CL

| IF TPAR\_OP expr TPAR\_CL THEN TLLAVE\_OP list\_sents TLLAVE\_CL *ELSE* *TLLAVE\_OP* *list\_sents* *TLLAVE\_CL*

while: *WHILE* *TPAR\_OP* *expr* *TPAR\_CL* *TLLAVE\_OP* *list\_sents* *TLLAVE\_CL*

## **Análisis Semántico**