Instituto Tecnológico de Buenos Aires

Autómatas, Teoría de Lenguajes y Compiladores (72.39)

Trabajo Práctico Especial

Autores:	*Legajo:*
Juan Pablo Oriana	60621
Agustin Naso Rodriguez	60065
Brittany Lin	60355
Leonardo Agustín D'Agostino	60335

Noviembre 2021

Índice

1. Introducción

El objetivo del trabajo fue el de diseñar un lenguaje de programación (conjunto a su compilador apropiado) en base a los conocimientos adquiridos durante la materia. Para esto se construyo una gramática, para luego utilizar Lex como herramienta de análisis lexico y Yacc como herramienta de análisis sintáctico. Finalmente se volcó todo esto en un compilador llamado **gcan** que genere un ejecutable en base a los archivos de código. El lenguaje generado tiene el nombre de **canvalize**

2. Canvalize: un vistazo

2.1. Idea general

La idea general de canvalize es la de generar un lenguaje imperativo al estilo de C que además incluye un tipo de dato **canvas** que permite hacer graficación rápida y sencilla en *STDOUT*.

La inspiración vino de nuestra propia experiencia como programadores. Muchas veces es un inconveniente cuando se trabaja en el bajo nivel o en aplicaciones que no son inherentemente gráficas encontrar la forma de visualizar eficientemente lo que uno esta haciendo. En C, esto suele recaer en escribir millones de **printf()** engorrosos que, no solo contaminan el código, si no que la mayoría de las veces no están bien formateados y se ven mal.

2.2. Capacidades

El lenguaje aporta cuatro tipos de datos. Los dos requeridos por la cátedra (cadenas y números enteros), con el adicional de dos tipos que consideramos fundamentales para nuestro trabajo: el tipo **canvas**, que es una suerte de matriz NxM graficable; y un tipo **double**, un tipo numérico de punto flotante para cálculos mas precisos.

Adicionalmente, el lenguaje permite el uso de constantes numéricas, textuales y de canvas (asignadas en definición y **no modificables**), loops while, estructuras condicionales if-else, funciones de read (numerico) y write (numerico y textual), un punto de entrada **main**, instructivas de retorno para finalizar la ejecución y múltiples operadores, tanto numéricos como para el uso del canvas (estos seran expandidos en secciones consiguientes).

2.3. Consideraciones realizadas

El mayor punto de incertidumbre para con la consigna al realizar el lenguaje era la cuestión de como realizar el punto de entrada del lenguaje. Para eso elegimos hacer un token **main:** que inicie el programa principal. Antes de la declaración de main solo esta permitido hacer declaraciones y asignaciones de variables. El main se puede detener en cualquier momento con un return. Al no haber funciones todavía

en el lenguaje no nos tuvimos que preocupar con el uso de llaves para la contención del programa principal.

Otra decisión tomada es que las funciones de read sean solo numéricas (se pueden recibir texto pero esto resulta en leer -1). Tomamos esta decisión, no por la incapacidad de realizar lectura en variables textuales (que es bastante fácil) , si no porque el lenguaje aun no soporta operaciones con cadenas de texto (comparaciones, strlen, sprintf, etc.). Esto implica que si bien uno podría leer una cadena de STDIN, poco podría hacer con ella mas que imprimirla.

3. Descripción del desarrollo

El lenguaje se realizo en cinco segmentos: La definición de los tokens en **Lex**, el análisis sintáctico en **Yacc**, la generación de un AST para trabajar fácilmente con el programa parseado, la validación de variables en base al AST, y finalmente el parseo del AST a un programa .c para poder compilarlo con gcc.

3.1. Lex y Yacc para la construcción de un AST

Lex y Yacc se utilizaron en conjunto para el análisis completo del input. Su funcionalidad final, además de encontrar errores en la sintaxis del programa, es la de transformar el archivo en texto plano a una estructura que sea mas manejable para la compilación del código. En el Lexer se definen los tokens con los cuales voy a representar símbolos finales en la gramática, luego los utilizo en el Parser (Yacc) para ir reconociendo si lo que me esta llegando efectivamente es una entrada valida y coherente con el lenguaje definido [1]. Para ver la definición completa de la gramática ver la sección 4. En cada reducción, el parser crea un nodo representativo a lo que se esta reduciendo (por ejemplo, si se reduce un while, se crea un nodo while [fig. 1]) que consecuentemente es agregado a una estructura mayor llamada **Abstract Syntax Tree (referido como AST) [2]**. El AST tiene un nodo raiz que inicialmente tiene como hijos las instrucciones del programa en su capa mas exterior. Cada instrucción es de un tipo particular y por ende se puede llegar a ramificar de forma diferente. Este genera una estructura jerárquica (y no text/plain como un archivo fuente) que facilita mucho el análisis de alto nivel del input. [3]

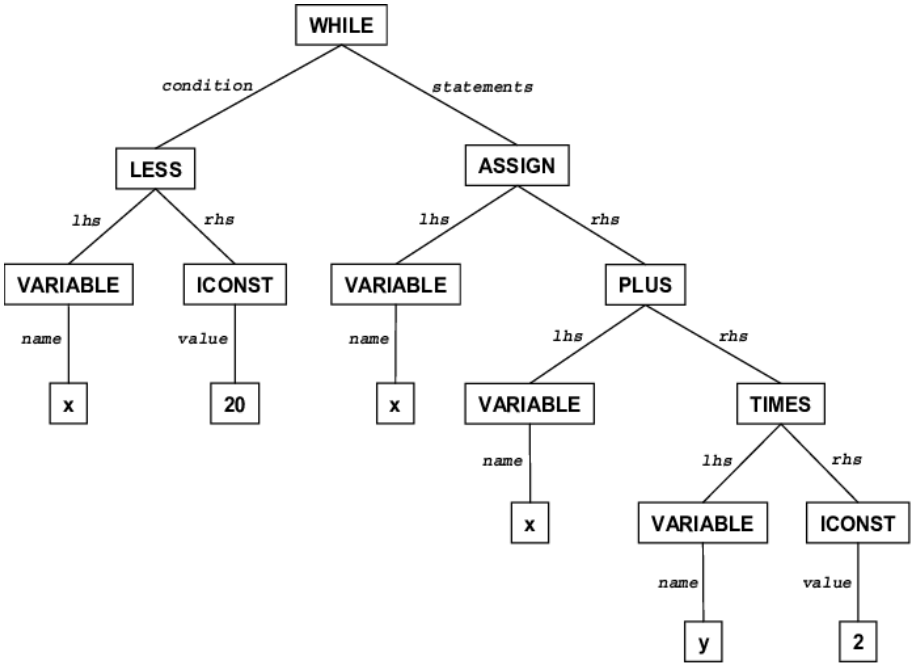

Figura 1: La expansion de un nodo while en un AST

3.2. Validación de las variables

Como en el momento de parseo, los nombres de variables son simplemente tokens no diferenciables entre si, es muy difícil hacer en ese momento la validación de tipos y asignaciones que corresponden a todo compilador. Sobre todo cuando un considera que una variable puede estar internamente anidada en una expresión compleja. Para solucionar esto, la validación se generar recursivamente una vez que el AST esta generado en su completitud, recorriendo todas las instrucciones y expresiones construyendo una lista de símbolos [4] mientras va validando que no se rompan ninguna de las siguientes reglas:

- No se declare mas de una vez una variable
- No se asigne o use una variable que no fue declarada
- No se asigne o use una variable que, si bien fue declarada previamente, esto sucedió en un nivel de anidamiento mas profundo (adentro de un if, por ejemplo).
- No se le asigne a una variable declarada un tipo incorrecto (la excepción a esto son ints en doubles y doubles en ints, ahí el lenguaje castea automáticamente al tipo de la variables).
- No se asigne mas de una vez una variable de tipo final (constante).
- No se usen tipos no numéricos en expresiones. Por ejemplo ((canvas + string) - int) es algo completamente sin sentido e incorrecto.
- No se usa write en un canvas

- No se usa read en un tipo no numérico
- No se plotea algo que no sea un canvas
- Las operaciones a canvas siempre se realizan en variables de tipo canvas y llevan como parámetros tipos numéricos (luego casteados a int).

En caso de encontrar cualquiera de estos comportamientos inesperados, la compilación finaliza con los mensajes de error correspondientes para que el usuario los pueda arreglar.

Algoritmo 1: Ejemplo de un algoritmo que rompe validaciones

```
int i = 0;
while (i == 0) do
    if (i == 0) then
        i = i + 1;
        int n = i;
    else
        i = "hola";
        n = 2;
    write(n);
```

El algoritmo 1 podra ser sintáctica y semánticamente correcto para su lenguaje, sin embargo veamos que rompe alguna de las reglas establecidas. **n** se usa sin inicializar en el else. **i** se le asigna un string cuando esta declarado como un int. **n** se usa afuera del if-else cuando es definido solamente dentro de este.

3.3. Llevar el AST a C

Una vez que se realizaron todas las validaciones pertinentes, tenemos la seguridad que nuestro AST ya esta listo para ser parseado a C. Para esto usamos una libreria de funciones **ast_to_c** que va a dejar la representacion correcta de nuestro AST construido en un archivo temporal **aux.c**. Este archivo sera compilado en conjunto a una libreria de funciones auxiliares para el canvas que generaran nuestro ejecutable final. La traduccion del AST a C sirve tambien para poder ir liberando la estructura construida.

4. Descripción de la gramática

4.1. Producciones de la gramática

- program → instruction program | FINAL_EOF
- instruction → full_declare DELIMETER | full_cv_declare DELIMETER| assign DELIMETER | assign DELIMETER | write DELIMETER | read DELIMETER | if | while | plot DELIMETER | cv_op DELIMETER | return DELIMETER | MAIN
- block → instruction block | instruction
- if → IF PARENTH_OPEN expresssion PARENTH_CLOSE CURLY_OPEN block if_end
- while → WHILE PARENTH_OPEN expression PARENTH_CLOSE CURLY_OPEN block CURLY_CLOSE
- return → RETURN expression
- if_end → CURLY_CLOSE | CURLY_CLOSE ELSE CURLY_OPEN block CURLY_CLOSE
- full_declare → declare | declare ASSIGN value | final_declare ASSIGN value
- full_cv_declare → cv_declare ASSIGN cv_value
- declare → type SYMBOL_ID
- final_declare → FINAL type SYMBOL_ID
- cv_declare → CANVAS_TYPE SYMBOL_ID
- type → INTEGER_TYPE | STRING_TYPE | DOUBLE_TYPE
- assign → SYMBOL_ID ASSIGN value
- value → expression | STRING
- cv_value → CURLY_OPEN INTEGER ',' INTEGER CURLY_CLOSE
- write → WRITE PARENTH_OPEN expression PARENTH_CLOSE | WRITE PARENTH_OPEN STRING PARENTH_CLOSE
- read → READ PARENTH_OPEN SYMBOL_ID PARENTH_CLOSE
- plot → PLOT PARENTH_OPEN SYMBOL_ID PARENTH_CLOSE

- cv_op → SYMBOL_ID BIN_CV_OP CURLY_OPEN expression ',' expression CURLY_CLOSE | SYMBOL_ID BIN_CV_OP_CHAR CURLY_OPEN expression ',' expression ',' ASCII CURLY_CLOSE | SYMBOL_ID TRI_CV_OP CURLY_OPEN expression ',' expression ',' expression CURLY_CLOSE | SYMBOL_ID UNI_CV_OP CURLY_OPEN expression CURLY_CLOSE | SYMBOL_ID QUAD_CV_OP_CHAR CURLY_OPEN expression ',' expression ','expression ',' expression ',' ASCII CURLY_CLOSE

  expression → PARENTH_OPEN expression PARENTH_CLOSE | UNI_OP expression | '-' expression | expression BIN_OP expression | expression '-' expression | INTEGER | DOUBLE | SYMBOL_ID

4.2. Explicación de los símbolos:

Todo lo que esta en mayúscula es un terminal y lo que esta en minúscula es un no terminal, aquí una breve descripción de cada uno

4.2.1. No terminales

- **program**: Inicio del programa.
- **instruction**: Inicio de una instruccion (Monolinea terminada en ; o while/if statements.
- **block**: Bloques de codigo que van dentro de un if/else o un while, rodeados de .
- **full_declare**: Declaración completa, contempla declaración con y sin asignación.
- **full_cv_declare**: Idem para tipos canvas.
- **declare**: Declaración sin asignación.
- **cv_declare**: Idem para tipos canvas.
- **final_declare**: Declaración de constantes.
- **type**: Tipo de datos con el que arranca una declaración.
- **assign**: Asignación de variable.
- **value**: Valor que puede ser asignado a una variable.
- **write**: Función de escritura.
- **read**: Función de lectura.
- **plot**: Función de ploteo de canvas.
- **cv_op**: Operaciones de modificación de un canvas.
- **expression**: Expresión lógico/matemática.

4.2.2. Terminales

Se muestran los pares id con la cadena/regex a la que corresponden en lex.

- **INTEGER** → **[0-9]**+: Numero entero.
- **DOUBLE** → **f[0-9]+.[0-9]**+: Numero con punto flotante.
- **INTEGER_TYPE** → **int**
- **DOUBLE_TYPE** → **double**
- **CANVAS_TYPE** → **canvas**
- **STRING_TYPE** → **string**.
- **WRITE** → **write**
- **READ** → **read**
- **MAIN** → **main**
- **IF** → **if**
- **WHILE** → **while**
- **ELSE** → **else**
- **ASSIGN** → =
- **DELIMETER** → **;**
- **CURLY_OPEN** → **{**
- **CURLY_CLOSE** → **}**
- **PARENTH_OPEN** → (
- **PARENTH_CLOSE** →)
- **FINAL** → **final**
- **BIN_OP** → **!**=|<=|>=|==|<|>|&&| || |+|-|*| %: Operadores binarios
- **UNI_CV_OP** → **color**: Operador canvas con un parámetro.
- **BIN_CV_OP** → **dot**: Operador canvas con dos parámetros.
- **BIN_CV_OP_CHAR** → **char**: Operador canvas con dos parámetros y un char.
- **TRI_CV_OP** → **hor**|**vert**: Operador canvas con tres parámetros.
- **QUAD_CV_OP_CHAR** → **fill**: Operador canvas con cuatro parámetros y un char.
- **SYMBOL_ID** → **"[a-zA-Z_][a-zA-Z0-9_]*"**: Nombre de un simbolo.

5. Especificaciones técnicas del lenguaje

5.1. Similitud con C

Exceptuando las especificaciones encontradas en las subsecciones 5.2 y 5.3 de este apartado, fue de gran importancia para nosotros que el lenguaje se comportase, en tanto como sea posible, al lenguaje de programación C. La idea es que alguien que ya tenga los conocimientos de C (o cualquier lenguaje tipado del estilo) pueda comenzar a programar inmediatamente en **Canvalize** sin ningún problema. Es por eso que los bloques condicionales y los while se construyen de forma identica a C. Los tipos de variables son también análogos (a excepción de las cadenas que se prefijan con string).

Sin embargo, como se menciono previamente, hay algunas diferencias a nivel semántico: las constantes se definen de forma tipada y precedida por un token **final**. El bloque principal del codigo no se encierra entre un int main(){} si no que simplemente se arranca con un **main:**

5.2. Control numérico

Como se menciono previamente, el lenguaje habilita tanto el uso de enteros como de tipos numéricos con punto flotante, denominados int y float (siempre antecedidos por la letra **f**) respectivamente. El lenguaje se programo con la idea de que siempre maneje *safe casting*[5]. Esto quiere decir que uno puede operar **siempre** entre enteros y floats sin jamas tener un conflicto de tipado. Las operaciones unarias y binarias que se permiten utilizar con los tipos numéricos (y **solo** con los tipos numéricos) son:

Operación	**Binaria?**	**¿Castea los numeros a int?**	**¿Devuelve siempre un int?***
!	No	No	Si
+	Si	No	No
-	A veces	No	No
*	Si	No	No
/	Si	No	No
<= , >= , <, >	Si	No	Si
== , !=	Si	No	Si
&&, ||	Si	No	Si
%	Si	Si	Si

Cuadro 1: Tabla de operaciones
* *Esta pregunta solo tiene sentido si hay un double en la operación*

Siguiendo esta linea de *safe casting*[5], cualquier número que se le pase a una función o operación que deba ser entero (como el índice matricial en las operaciones del canvas) sera oportunamente convertido.

5.3. Canvas

El canvas es realmente el diferencial de este lenguaje. En su forma mas concreta es simplemente una matriz de caracteres ascii de tamaño NxM que además se le puede asociar un color. Pero el secreto cae en su simplicidad de uso.

Un canvas se puede definir de la siguiente manera:

```
canvas cv = {N,M};
```

Donde **N** es el ancho y **M** el alto en caracteres ascii del canvas. Es importante destacar que los canvas son por definición un tipo final, es decir que no pueden ser reasignados o definirse sin inicializarse. Ademas, **N** y **M** no pueden ser valores variables.

En algunas operaciones del canvas se utilizaran coordenadas planares. Para esto hay que tener en cuenta que el 0 en **x** es el extremo izquierdo y el 0 en **y** es el extremo superior [fig. 2].

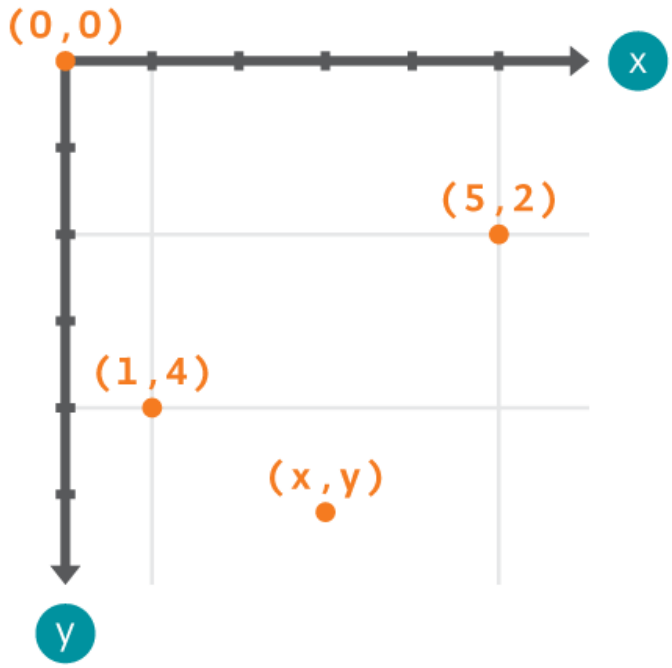

Figura 2: El sistema de cordenadas de un canvas.

A continuación se listaran todas las operaciones habilitadas con un canvas definido.

5.3.1. color

```
cv color {color};
```

Cambia el color de visualización del canvas. Donde **color** es un int que puede representar los siguientes colores:

- 0 - blanco
- 1 - rojo
- 2 - amarillo
- 3 - verde
- 4 - azul

Cualquier int fuera de estos no genera ningún cambio.

5.3.2. dot

```
cv dot {x,y};
```

Dibuja un * en el punto **(x,y)** del canvas, si existiese.

5.3.3. char

```
cv char {x,y,c};
```

Dibuja el ascii de **c** en el punto **(x,y)** del canvas, si existiese.

5.3.4. hor

```
cv hor {xStart,xEnd,yAxis};
```

Dibuja una linea horizontal del ***max(0,xStart)*** al ***min(xEnd,cv.width-1)*** en el eje y igual a **yAxis**.

5.3.5. vert

```
cv vert {yStart,yEnd,xAxis};
```

Dibuja una linea vertical del ***max(0,yStart)*** al ***min(yEnd,cv.height-1)*** en el eje x igual a **xAxis**.

5.3.6. fill

```
cv fill {x1,y1,x2,y2,'c'};
```

Si **(x1,y1)** es la esquina superior izquierda y **(x2,y2)** la esquina inferior derecha, llena ese rectángulo del canvas con el carácter ascii **c**

5.3.7. plot

```
plot(cv);
```

Gráfica el canvas **cv** tal cual es su estado actual.

6. Dificultades encontradas

Por mucho tiempo una de las dificultades que mas complicaciones generó fue la de lidiar con los conflictos de shift/reduce que generaban el ordenado de las operaciones, en particular los conflictos entre las operaciones unarias (!,-) y las operaciones binarias (+,-,/,etc). Esto pudo resolverse usando el operando %left de yacc para poder marcar el orden de operaciones. Afortunadamente, el trabajo final no presenta ninguno de estos conflictos.

Otro problema fue el de la resolución de la interoperabilidad entre doubles e integers. Queríamos que el usuario no se deba preocupar por los casteos de parámetros, si no que simplemente escriba las expresiones que desee y el compilador automáticamente tome esas decisiones de casteo por el (por ejemplo, un integer mas un double deberia dar un double)[5]. Sorpresivamente, esto genero mas dificultades de las que esperábamos ya que algunas operaciones binarias castean siempre a integer (&& y ||, por ejemplo), mientras que otras castean siempre a double, de ser posible (+,-,/,*). Entender bien en que tipo se cae en cada caso es **fundamental** para que el usuario este seguro que el numero que piensa que esta almacenando sea efectivamente el correcto. La forma de solucionar esto es que los nodos de expresiones tengan un flag recursivo que vaya decidiendo si

- a) No hay que castear nada (todos los tipos son iguales)
- b) La operación contiene un double y debe especificar a double
- c) La operación tiene un operador que castea a integer si o si.

7. Futuras extensiones

Pensando en Canvalize como un proyecto a futuro, nos gustaría poder realizar las siguientes extensiones al lenguaje:

- **Adición de funciones:** Poder trabajar con funciones y subrutinas es casi esencial para cualquier lenguaje de orientación imperativa como es **Canvalize**. La complejidad de esta extensión es bastante alta, ya que implicaría cambiar la estructura del lenguaje que, en muchos puntos, asume que es una cadena de ejecución continua sin saltos.
- **Implementación de vectores/matrices:** Si bien el canvas es una matriz NxM en el fondo, su función es exclusivamente gráfica, y no habilita operaciones mas allá de eso. Al usuario le gustaría muchas veces llevar tracking numérico de lo que hay adentro de una matriz de datos mas allá de poder graficarla. La complejidad de implementar tipos de datos como vectores o matrices es moderada pero puede llegar a ser alta si se quiere hacer una buena integración con las operaciones del canvas.
- **Casteos:** Aunque el compilador se encarga de castear los tipos de datos de la mejor forma posible, estaría bueno que el usuario pueda hacer un manejo

de casteos manual. La complejidad de esta implementación es relativamente baja.

- **Importación de librerías creadas por el usuario:** Esta implementación es mucho mas dependiente de la primera extensión que otra cosa. Si se llegan a desarrollar funciones, esto debería ser trivial.

8. Ejemplos

Tal como se requirió, se incluyeron en el directorio ***examples*** cinco (5) ejemplos que ayudan a mostrar el rango completo de las capacidades del lenguaje. Se recomienda compilarlos y probarlos en este orden:

- **prime.can**: Se elije un numero máximo y se evalua la primalidad de todos los enteros hasta ese numero. Luego de eso grafican los primos encontrados en un canvas (hasta el numero 99 inclusive). Este es un ejemplo integrador donde se utilizan **todas** las estructuras lógicas del lenguaje, asi como tipos numéricos, cadenas, canvas, lectura, escritura y mas.

- **ops.can**: Inicialmente muestra el comportamiento de las distintas operaciones habilitadas. Finalmente le ofrece al usuario la posibilidad de elegir un paso N. Divide al numero uno en N pasos y luego los suma para volver a aproximar uno. Luego verifica si el resultado de la suma es realmente igual a uno (chequeo de precisión numérica).

- **fibo.can**: Imprime la secuencia fibonacci hasta una iteración i seleccionada por el usuario (máximo 46 iteraciones).

- **faces.can**: Grafica uno de cuatro emoticones distintos dependiendo de como te sientas del uno al diez.

- **flags.can**: Grafica algunas banderas del mundo en canvas.

Adicionalmente a estos ejemplos funcionales, se incluyo en el subdirectorio ***examples/error_cases*** tres (3) ejemplos de compilaciones fallidas producto de inconsistencia simbólicas para que se evalué el buen comportamiento del compilador.

- **final_error.can**: Muestra los errores en asignacion de constantes.

- **scope_error.can**: Muestra los errores de scoping al usar variables fuera de donde corresponden.

- **type_error.can**: Muestra los errores en type casting.

9. Conclusión

En resumen, nos encontramos realmente satisfechos con el lenguaje resultante al final de tanto trabajo. Nos hubiese encantado poderlo expandirlo aún mas, pero dado como empezamos y el conocimiento que poseíamos previo a cursar la materia, es increíble que hayamos podido usar lo aprendido para una aplicación tecnológica que consideramos podría llegar a ser útil para la comunidad informática.

Estamos firmemente convencidos que **Canvalize** es un proyecto con amplia capacidad de expansión a futuro, si es que nos enfocamos a los puntos de desarrollo importantes en el lenguaje. Estamos ansiosos y expectantes de como pueda llegar a evolucionar.

Referencias

[1] Thomas Niemman. A guide to lex & yacc. URL https://arcb.csc.ncsu.edu/~mueller/codeopt/codeopt00/y_man.pdf.

[2] Dominik Kundel. Abstract syntax trees, 2020. URL https://www.twilio.com/blog/abstract-syntax-trees.

[3] Ruslan Spivak. Let's build a simple interpreter. part 7: Abstract syntax trees, 2015. URL https://www.twilio.com/blog/abstract-syntax-trees.

[4] Symbol table. URL https://en.wikipedia.org/wiki/Symbol_table.

[5] Barbara Thompson. Type casting in c: Type conversion, implicit, explicit with example, 2021. URL https://www.guru99.com/c-type-casting.html.