Trabajo Práctico Especial

Desarrollo del lenguaje ecc (easy C) y su compilador

72.39 - Autómatas, Teoría de lenguajes y Compiladores

Junio 2021

Autores:

Galende, Lautaro (60287)

Itokazu, Sebastián Raúl (60392)

Ratti, Valentín Segundo (60031)

Rosenblatt, Tommy (60393)

Docentes:

Arias Roig, Ana María

Ramele, Rodrigo Ezequiel

Santos, Juan Miguel

Índice

Introducción

El lenguaje *ecc* (easy C) nace a partir de un proyecto académico realizado para la materia **Autómatas, Teoría de Lenguajes y Compiladores**. El presente informe describe cuál es su propósito y las reglas del lenguaje de programación creado, así como también cuáles fueron las dificultades a la hora de inventarlo y eventuales mejoras para futuras versiones de *ecc*.

Idea subyacente y objetivo del lenguaje

El objetivo principal del lenguaje de programación *ecc (easy C)* es que facilite la introducción en la programación y también que permita entender código a personas sin conocimientos del tema. Es decir, que además de aplicar los conocimientos de la materia, se intenta ofrecer una experiencia para el usuario más lúdica. El mismo podría ser utilizado para enseñar programación en escuelas primarias y secundarias como incentivo a que los chicos se interesen en el mundo de la programación.

Consideraciones Realizadas

La sintaxis utilizada es parecida al vocabulario utilizado en la cotidianidad, en donde se busca que el programador establezca una conversación con el compilador. A diferencia de otros lenguajes de programación, el lenguaje consiste únicamente en caracteres comúnmente utilizados para la redacción de textos. No se utilizan llaves(“{“,”}”,”[“,”]”). Además se indica claramente dónde comienza un bloque de cógido y dónde finaliza.

A continuación se listan las restricciones/limitaciones del lenguaje:

- Es obligatorio declarar las variables antes de asignarlas.
- Los bloques condicionales y de repetición, deben ser finalizados con la palabra end.
- Todas las líneas de código, deben finalizar con un punto (“.”). Exceptuando que sea el final o comienzo de un bloque.
- Las variables deben arrancar con una letra.

- El alcance de todas las variables está dentro de la función principal. Es decir, no se pueden declarar dos variables con el mismo nombre por más que estén en distintos bloques de código.
- Los nombres de las variables de tipo *text*, deben ser palindromas. Es decir que el nombre se debe leer igual de izquierda a derecha, que de derecha a izquierda. por ejemplo: ojo, oso, seres, arenera, sometemos, somos, reconocer, etc.
- Los nombres de las variables de tipo *number*, deben tener una cantidad par de caracteres. Por ejemplo: azul, espejo, número, desarrollo, etc.
- Hay un número máximo de variables posibles, actualmente 50.
- Al menos el 75% de las asignaciones, deben tener un 'please' adelante. Esto se realizó para incentivar la educación del usuario.
- Los array, únicamente pueden ser de tipo *number*.
- Solo se pueden imprimir términos, la definición de los mismos dentro del contexto de ecc será explicada posteriormente.

A continuación se listan todas las palabras reservadas del lenguaje: "create", "const", "number", "text", "list of", "with", "items", "item", "called", "save", "into", "print", "read", "is_different_from", "is_equal_to", "is_greater_than", "is_lower_than", "plus", "minus", "multiplied_by", "divided_by", "module", "or", "and", "if", "else", "while", "repeat", "do", "end", "**.", "(", ")", ":",** start", "finish", "please", "ignore" y los símbolos de comillas.

Descripción del desarrollo del TP

El primer paso, al igual que para la creación de cualquier otro lenguaje de programación siempre es la idea, el por qué. En nuestro caso, quisimos buscar la manera de combinar una gramática hablada con un lenguaje que funcione para, además, inculcar algunos valores de educación. Creemos que de cara al presente y al futuro, será fundamental la enseñanza de programación desde la niñez y la manera de hacerlo es que sea didáctica y más entretenida para los chicos o cualquier principiante en este mundo.

Desde allí, se realizó un brainstorming de ideas donde se discutió la gramática del lenguaje que fue tomando forma hasta llegar a completar todas sus reglas de sintaxis, que se

tradujeron al lenguaje Lex y Yacc para el scanner léxico y el parser sintáctico, respectivamente.

El mayor desafío del trabajo estuvo en traducir código escrito con las reglas propuestas a un archivo en lenguaje C. Para ello, el analizador sintáctico a medida que parsea las líneas de código va armando el árbol con las distintas producciones que se van utilizando. Finalmente, este árbol se recorre utilizando el algoritmo DFS para imprimir el código en lenguaje C a un archivo temporal que finalmente será compilado para tener el ejecutable del código *ecc*.

Descripción de la gramática

A continuación se especifica cómo se estructura el código.

Estructura general

El código siempre comienza con un *start*. Se van escribiendo todas las líneas de instrucciones hasta que finaliza con *finish*. Como se mencionó anteriormente, todas la líneas deben finalizar con un punto, exceptuando el caso de inicio y finalización de bloques y en el caso de la primera línea(start) o la última(finish).

Variables

Existen dos tipos de variables: *text* y *number*.

Las variables de tipo text, son equivalentes a un string en otros lenguajes conocidos. Los mismos se pueden concatenar utilizando plus(explicado posteriormente).

Las variables de tipo *number* son equivalentes a un int. Con las mismas se pueden realizar sumas, restas, multiplicaciones, divisiones y resto.

Las declaraciones de las mismas son de la siguiente manera: "*create type called name.*". Siendo type el tipo de variable (*text* o *number*) y *name* el nombre de dicha variable.

Las asignaciones deben escribirse en una línea de código distinta a la declaración. La misma se realiza de la siguiente manera: "*save expresión into name.*". Donde name es una variable ya declarada y expresión se explicará en los siguientes ítems.

Además, como se detalló anteriormente, al menos en el 75% de las asignaciones se debe colocar un *please* antes de la asignación: "*please save expresión into name.*".

En el caso de una lista (en realidad son arreglos), se declara de la siguiente manera: "*create list of number with qty items called name.*". En donde *qty* es la cantidad de ítems que va a tener la lista, y name es el nombre de la misma. Qty debe ser un número literal, por ejemplo 5, y no puede ser una variable.

Luego para asignar cada valor, se realiza la siguiente linea: "*save expresión into namelist item index.*", donde namelist es el nombre de la lista ya declarada anteriormente, index la posición en donde se desea asignar el valor(empieza en cero) y expresión se explica en otro ítem posterior.

Print

Para imprimir un valor, se escribe la siguiente línea: "print value.", siendo value una variable o un texto entre comillas.

Read

Para leer texto de entrada, se escribe: "read into name.", siendo name una variable ya declarada.

Bloque de repetición

Para utilizar un bloque de repetición se debe escribir lo siguiente: "*while (condición) repeat: code. end*". La condición debe ser una afirmación, la conformación de la misma está especificada en la próxima sección en la definición de la gramática. code representa todas las líneas de código a repetir.

Bloque condicional

El caso del condicional es similar al *while*, se escribe de la siguiente manera: "*if (condition) do: code1. end*", o con la opción de else: "*if (condition) do: code2. end else do: code. end*". La condición representa lo mismo que en el bloque de repetición. code1 representa todas las líneas de código que se desean realizar en caso de que la condición se cumpla. En el caso del else, code2 se realiza cuando la condición no se cumple.

Condiciones

La condición se puede componer de distintas maneras: "*condition expression condition*", "*condition and condition*", "*condition or condition*". En el caso de *and*, el resultado será verdadero si las dos condiciones lo son. En el caso del *or*, el resultado será verdadero si cualquiera de las dos condiciones lo es.

Comparadores

Para realizar una comparación se utiliza uno de los siguientes comparadores: “is_different_from”, “is_equal_to”, “is_greater_than”, “is_lower_than”. Los mismos se utilizan como parte de una condición para comparar dos expresiones: “expression comparator expression”.

Expresiones

Las expresiones pueden ser variables u operaciones entre variables.

Operadores

Los operadores que se pueden utilizar únicamente para el tipo *number* son: “plus”, “minus”, “multiplied_by”, “divided_by”, “module”. Para el tipo *text*, se utiliza el operador plus para concatenar dos textos.

Los mismos se utilizan dentro de una expresión: “*expresión operador expresión*”.

Comentarios

Se puede comentar una línea de la siguiente forma: ignore:*, donde * puede ser cualquier carácter.

Definición de la gramática

A continuación se especifica la gramática:

program: START code FINISH

code: instruction

| instruction code

| λ

instruction: declaration END_OF_LINE

| assignment END_OF_LINE

| PRINT term END_OF_LINE

| READ INTO VARNAME END_OF_LINE

| WHILE OPEN_PAR condition CLOSE_PAR REPEAT COLON code END

| IF OPEN_PAR condition CLOSE_PAR DO COLON code END else

else: ELSE DO COLON code END

| λ

condition: expression comparator expression

| condition AND condition

| condition OR condition

| OPEN_PAR condition CLOSE_PAR

type: NUMBER

| TEXT

op: PLUS

| MINUS

| MULTIPLY

| DIVIDE

| MODULE

comparator: GREATER

| LOWER

| EQUAL

| DIFF

expression: term

| expression op expression

| OPEN_PAR expression CLOSE_PAR

term: NUMBER_VAL

| TEXT_VAL

| VARNAME

| VARNAME ITEM NUMBER_VAL

| VARNAME ITEM VARNAME

declaration: CREATE type CALLED VARNAME

| CREATE CONST type CALLED VARNAME

| CREATE LISTOF type WITH NUMBER_VAL ITEMS CALLED VARNAME

assignment: SAVE expression INTO VARNAME

| PLEASE SAVE expression INTO VARNAME

| PLEASE SAVE expression INTO VARNAME ITEM NUMBER_VAL

| SAVE expression INTO VARNAME ITEM NUMBER_VAL

| PLEASE SAVE expression INTO VARNAME ITEM VARNAME

| SAVE expression INTO VARNAME ITEM VARNAME

Dificultades encontradas en el desarrollo del TP

En un principio, las comparaciones se traducían a C de igual forma para las variables de tipo number y text; es decir ‘var is greater than var1’ se convertía en ‘var > var1’. Sin embargo esto era incorrecto para las variables de tipo text, ya que se terminaba realizando la comparación de punteros en lugar de la comparación de textos. Como solución, decidimos que para los text, las comparaciones utilicen la función *strcmp* de la librería estándar de C.

Para realizar la lectura de entrada estándar, se analizó entre solo permitir leer de a un carácter por vez (estilo getchar) o permitir leer enteros y strings (estilo scanf). Se decidió optar por la segunda opción ya que brinda más herramientas al desarrollador. Sin embargo, esto supuso una dificultad al traducir las variables tipo *text* a C, donde se las equipara con una variable de tipo char *, pues al hacer un scanf dentro de la variable se utiliza un puntero sin memoria alocada. Ante esto, se decidió inicializarlas con un tamaño de 1024 bytes de forma arbitraria.

Futuras extensiones

Arreglos que no requieren el tamaño en la declaración

A diferencia de los arreglos de tamaño estático implementados, en este caso, no haría falta que se especifique la cantidad de ítems que va a tener la lista. La dificultad de esta implementación es a la hora de no saber cuánta memoria se va a destinar inicialmente, sino que eso puede ir cambiando mediante realocación de memoria para poder cambiar el tamaño de los mismos.

Tipos de datos

Como en muchos lenguajes, estaría bueno poder ofrecer otros tipos de datos más variados que abren las puertas a poder realizar muchas más cosas. Por ejemplo: decimales o estructuras. Cuando ya nos funcionaba el compilador, decidimos agregar más tipos de datos. Estábamos entre agregar estructuras o arreglos estáticos. Por cuestiones de que preferíamos destinar el tiempo en otras funcionalidades, se decidió implementar únicamente arreglos estáticos.

Funciones

Las funciones son un elemento muy utilizado en la programación. Empaquetan y aíslan una parte del código que realiza alguna tarea específica del resto del programa. Por eso, en un futuro, agregaría muchos beneficios al lenguaje si se incluye la posibilidad de definir funciones nuevas y poder utilizarlas.

Referencias

Para familiarizarnos con los lenguajes Lex y Yacc, utilizamos la bibliografía dispuesta por la cátedra, *Levine, Mason, Brown, "Lex & Yacc", O'Reilly & Associates*.
Además, se utilizó el repositorio de GitHub de *YetAnotherCompilerClass*[1] como ejemplo de manejo de estas estructuras.
Para la realización del analizador semántico se utilizó como referencia el utilizado por el compilador de C ♭ [2], el cual encontramos en otro repositorio público de GitHub.

[1] https://github.com/faturita/YetAnotherCompilerClass
[2] https://github.com/sterenziani/chunchunmaru