Trabajo Práctico Especial
Teoría de Lenguajes y Autómatas

Lenguaje eaC

1er Cuatrimestre 2019

Integrantes del Grupo:

- Facundo Astiz Mayer
- Miguel Di Luca
- Pedro Remigio Pingarilho
- Pedro Vedoya

Índice:

Idea subyacente y objetivo del lenguaje

La idea del lenguaje surgió a partir de querer buscar un lenguaje más simple y más natural que el lenguaje C. Por eso mismo, el lenguaje se llama eaC o “easy”, y tiene como objetivo funcionar de una manera similar al lenguaje C, reemplazando todo lo que nos pareció que era tedioso por algo más simple. La idea general es que sirva como apoyo a alguien que es nuevo al mundo de la programación, y le aporte una manera más natural de programar para que luego sea más simple la transición a C.

El lenguaje hace esto haciendo que todos los tipos de datos tengan nombres de tres letras, así no solo son más cortos de escribir, sino que también se estandarizan para ser más fácil de recordar. También hicimos que los operadores lógicos sean “&” y “|” en vez de “&&” y “||”, y que el delimitador sea “.”, que hace que sea más natural y parecido a como se escribe normalmente. Hubo otros cambios que permiten que sea más sencillo, que serán analizados más adelante.

Consideraciones realizadas

Para la realización del TP, se utilizaron los conceptos aprendidos en clase y las guías de YACC y LEX, en conjunto con lo que investigamos en Internet. En base a estos conocimientos buscamos la manera de hacer el lenguaje más simple posible.

Para esto, tomamos las decisiones explicadas en la sección anterior, y a eso le añadimos que cuando se quiere utilizar una operación lógica, se debe encerrar entre paréntesis, así se puede ordenar bien la secuencia lógica y no hay confusiones. Además, cambiamos los operadores “if “ y “while “ por “when” y “during”, esto fue hecho para poder generar un lenguaje que fuese más parecido a lo que es escribir en la vida real, acompañado con el uso del “.” como delimitador. También cambiamos el operador lógico “!=” por el “<>”, que es como se expresa en lenguaje lógico, algo con lo que el usuario puede estar más familiarizado, y cambiamos el sistema de comentarios de el clásico a poder comentar haciendo “<-- comentario --> ” .

Finalmente, agregamos al lenguaje algo que no posee C, cambiamos el significado de ”!=” por el de un operador de asignación, al que a una variable numérica se le asigna el número deseado, multiplicado por -1.

Descripción del Desarrollo

El desarrollo comenzó a partir de las consideraciones habladas previamente, una vez teníamos diseñado el lenguaje, creamos el archivo lex.l, donde pasamos nuestras ideas a lexemas. Aquí no solo creamos las palabras reservadas, sino que también creamos los tipos de datos con los que íbamos a trabajar, que decidimos que serían int (num), char (var), char * (str), void (nul) y float (dec).

Luego, pasamos a diseñar el archivo yacc.y, en el que armamos la gramática que será explicada a continuación. Una vez creada ésta, buscamos una implementación de nodos en C para poder armar un árbol n-ario para poder imprimir fácilmente el código compilado en un archivo de salida, de github sacamos el código que utilizamos (adaptándolo a este caso, modificando los archivos node.h y node.c), y una vez que tuvimos esto, proseguimos a agregar el código de generación del árbol en yacc. Una vez que probamos y nos dimos cuenta que funcionaba, proseguimos a crear el makefile y ejemplos que demuestran el funcionamiento.

Funcionamiento

Si bien el funcionamiento se encuentra detallado en el ReadMe incluido en el proyecto, nos gustaría detallar lo básico en este informe:

1. El primer paso es crear el parser, para lo que se debe ejecutar el comando
   $>make all
   Siempre estando en el directorio raíz del proyecto
2. Luego, para poder compilar los ejemplos que armamos basta con correr
   $>make customExamples
   Esto generará los archivos ejecutables en la carpeta raíz, numerados del 1 al 5.
   Para ejecutarlos simplemente corra ./example#
3. Si desea limpiar la carpeta, el archivo makefile posee los comandos make clean (borra el parser), make cleanall (borra el parser y el archivo traducido a C), cleanExamples (borra los ejemplos).

Ejemplos

Para este proyecto armamos 5 ejemplos, que describiremos a continuación:

- Ejemplo 1: Muestra cómo funcionan las operaciones y los operadores de asignación, posee una muestra de cómo funciona el read y el printf.
- Ejemplo 2: Muestra cómo funcionan los comentarios y los strings.
- Ejemplo 3: Prueba de cómo funcionan los operadores lógicos, nuestro “if” y nuestro “while”.
- Ejemplo 4: Calcula la fórmula de fibonacci del número 9.
- Ejemplo 5: Calcula 3 elevado a la 4.

Descripción de la Gramática

La gramática produce el siguiente lenguaje haciendo uso de sus producciones:

- El lenguaje permite la declaración de funciones al principio de todo, esto es altamente recomendado para evitar que haya declaración implícita de funciones luego.
- Luego se debe crear el main, que debe ser declarado de la manera “num main() { código }”
- Para delimitar declaraciones de funciones, declaraciones de variables, asignaciones, llamadas de funciones, returns, end, operaciones, se requiere utilizar el “.” en lugar de “;”.
- Luego del main deben estar las funciones que se hayan declarado al principio, con su código correspondiente. La creación de funciones funciona igual a C.
- Hay 5 tipos de datos:
    - Num (int)
    - Str (char *)
    - Var (char)
    - Dec (float)
    - Nul (void)
- Para la declaración, es obligatorio asignar un valor a la variable, “tipo variable = valor”.
- Se pueden declarar constantes utilizando “tipo **fix** variable = valor”.
- Para el llamado de funciones se llaman igual que C, “funcion(argumentos).”.
- El return y la asignación de valores funcionan similar a C.
- Se puede terminar el programa utilizando “end.”.
- Los operadores aritméticos son (las operaciones aritméticas deben realizarse con espacio entre valores y los operadores):
    - +
    - -
    - *
    - /
    - %
- Los operadores relacionales son:
    - ==
    - <> (!=)
    - <=
    - >=
    - >
    - <
- Los operadores lógicos son (las sentencias lógicas se declaran entre paréntesis, del estilo (cond1 & cond2), o sino (!(cond1 | cond2) & (cond3 & cond4))):
    - & (&&)
    - | (||)
    - !

- Operadores de asignación (solo el = funciona para todos los tipos):
    - =
    - +=
    - -=
    - *=
    - /=
    - %=
    - !=
- Bloque condicional: “when(condición){ código } - else when(condición){ código } - else {código }”
- Bloque de repetición:
    - “during(condición){ código }”
    - “Repeat{ código } during(condición).”
- El lenguaje posee dos funciones especiales, una para leer de entrada estándar “read()” y otra para escribir “printf(argumentos).”, éstas funcionan igual que getchar y printf en C, respectivamente.

Dificultades Encontradas

Una dificultad fue que el grupo no estaba familiarizado con el lenguaje Yacc ni el uso de Lex, y de cómo poder utilizar un árbol para cambiar de un lenguaje a otro. Se utilizó el método de prueba y error, y se necesitó mucha investigación lo que atrasó el desarrollo del lenguaje.

Se puede notar que al hacer “make” en el proyecto, salen varios warnings diciendo que hay errores de repetición. No se pudieron revisar todos los conflictos por miedo a que esto genere un error inesperado, y como esto no impide que el programa funcione correctamente no se cambió. Se encontraron dificultades a la hora de escribir ejemplos para testear el funcionamiento del programa ya que los integrantes del grupo no estaban acostumbrados a las reglas del nuevo lenguaje creado, y escribiendo los ejemplos nos dimos cuenta que había reglas que pensamos que funcionaban, cuando no era el caso, y esto generó que tengamos que revisar una y otra vez y cambiar muchas cosas en el archivo yacc y el lex para que funcione.

Futuras Expansiones

- Agregar nuevos tipos de datos, entre ellos matrices, arrays y booleano, dado que no llegamos a agregarlos.
- Poder declarar variables globales, no pudimos lograr que nos funcione en esta instancia del proyecto.
- Arreglar todos los warnings que tenemos en el proyecto, para eliminar la ambigüedad y emprolijar la gramática.
- Un manejo de errores más comprensivo para el usuario.

- Poder agregar una función que funcione similar al sprintf de c, que puede leer todo lo que hay en entrada estándar.
- Poder hacer comentarios que funcionen en más de una línea.

Referencias

- https://github.com/ChuOkupai/n-ary-tree
- https://github.com/cderienzo/CASTLE
- https://github.com/faturita/YetAnotherCompilerClass
- Compilers: Principles, Techniques, and Tools, Alfred Aho
- Basamos conceptos de nuestro trabajo en lo que hablamos con compañeros de la materia, compartiendo lo que íbamos encontrando.