Trabajo Práctico Especial

30/06/2020

Sofía Picasso
57700

Ignacio Sampedro
58367

Holger Donath
58110

Martín Ciccioli
58361

Idea subyacente y objetivo del lenguaje

El objetivo del del lenguaje es buscar una forma más simple y veloz de escribir en el lenguaje C. Busca funcionar de manera similar a C, reemplazando todo lo que nos pareció tedioso del mismo por algo más sencillo.

La idea surgió en base a las competencias de programación estilo “Hackathon”, donde los participantes deben escribir su código lo más rápido posible. En el lenguaje creado, las sentencias son más cortas en general y se evita mucho el uso de algunos símbolos (por ejemplo, no es necesario utilizar el signo “=” para poder darle el valor a una variable). Sin embargo, también puede ser utilizado por programadores que simplemente les da pereza escribir una sentencia entera.

Se recomiendo el lenguaje para usuarios que ya conocen C bastante bien, ya que puede resultar mucho menos intuitivo si nunca se ha programado en este (por ejemplo, para inicializar una variable tipo string, en C se utiliza “char *” y se puede inferir que se trata de un string de caracteres por el nombre. En nuestro lenguaje simplemente se utiliza “s” para inicializar ese tipo de variable).

Consideraciones realizadas

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

Lo principal era achicar la sentencia y reducir el uso de código boilerplate, por lo cual se intentó utilizar la menor cantidad de caracteres para definir cada función, declaración de variable, etc. sin perder su funcionamiento.

Se decidió utilizar C como lenguaje del output ya que los miembros del equipo se sienten más cómodos con el uso de este y tienen un mejor manejo, pero también porque se considera un lenguaje más universal y popular que Assembler por ejemplo. Se compila el archivo temporal con gcc por las mismas razones.

Descripción del desarrollo

El comienzo del proyecto fue dedicado a reforzar el conocimiento de Lex y Yacc antes de comenzar a programar. Se hicieron algunos de los ejercicios de la guía opcional para poder entender la gramática de Lex y Yacc de a poco con ideas más sencillas.

Luego se comenzó a desarrollar el proyecto delineando algo similar a un BNF donde se planteaban los símbolos terminales, no terminales y sus producciones.

Sin embargo, al empezar a programar el equipo se dio cuenta que había que modificar el plan inicial según los errores y necesidades que surgían durante esta etapa.

Se realizaron bastantes cambios a la gramática inicialmente planteada y sus producciones.

Una vez que se pensó que la gramática estaba lo suficientemente bien definida, se comenzaron los tests de ejemplos para ver si todo funcionaba como planteado. A prueba y error se fue mejorando el proyecto hasta obtener el prototipo final.

Descripción de la Gramática

- El lenguaje permite la declaración de funciones al principio de todo. Esto puede ser sumamente útil para evitar que haya declaración implícita de funciones luego.
- Para delimitar declaraciones de funciones, declaraciones de variables, asignaciones, llamadas de funciones, returns, end, operaciones, se requiere utilizar el “.” en lugar de “;”. Esto se debe a que se considera que el “.” es un símbolo que se utiliza mucho y está más “a mano” que el “;” en la mayoría de los teclados.
- Hay solamente 2 tipos de datos: i (int) y s (char *)
- Se puede terminar el programa utilizando “end.”.
- Los operadores aritméticos son :
    - +
    - -
    - *
    - /
- Los operadores relacionales son :
    - =
    - !
    - >
    - <
    - >=
    - <=
- Los operadores lógicos son :
    - a (and)
    - o (or)
- El lenguaje también posee funciones para leer (r) y escribir (w) por entrada estándar, para comparar dos strings y para incrementar o decrementar un valor numérico.

- El return (ret) funciona de manera similar a C.

Funcionamiento

Para poder correr satisfactoriamente el programa es necesario seguir los pasos detallados a continuación:

- En primer lugar, debemos tener tener instalados los analizadores FLEX y BISON. Para eso, hay que ejecutar en la terminal los siguiente comandos:
  - sudo apt-get install flex
  - sudo apt-get install bison
- Luego, es necesario crear el compilador. Esto se logra ejecutando el comando $>make all
- A continuación, corremos el ejecutable pasándole como argumento el programa que queremos compilar. ./C-mplify <*dirección programa*>
- Se pueden agregar las flags -h (si se desea obtener información de los comandos), -r <n*ombre nuevo*> (si se desea cambiar el nombre del archivo ejecutable, el default es "a.out") y -k (si se desea guardar el archivo temporal)

Explicación de los tests

En esta sección detallamos los distintos ejemplos que implementamos para probar el correcto funcionamiento del proyecto.

Test 1 (function test)

En este ejemplo testeamos el adecuado comportamiento de las funciones. Para eso, declaramos una función, "funcionpoderosa", la cual retorna el valor que le devuelve otra función, "funcionnotanpoderosa". Esta última función retorna un valor numérico. En el main llamamos a la función "funcionpoderosa", retornando el valor numérico de la "funcionnotanpoderosa".

Nos pareció útil emplear este ejemplo para testear que suceda lo mismo en caso de que retornemos un string en vez de un valor numérico. Con tal fin, implementamos dos funciones más siguiendo el mismo procedimiento que el desarrollado anteriormente pero devolviendo un string en la segunda función.

Por último, tanto el valor numérico como el string retornado fueron incluirlos en un DEFINE para corroborar el apropiado desempeño de esta etiqueta

Test 2 (arithmetic test)

El objetivo de este test fue probar el correcto funcionamiento de las operaciones aritméticas básicas que soporta nuestro lenguaje. Se prueban la suma, resta, multiplicación y división entre variables y los resultados se imprimen en pantalla.

Test 3 (loops test)

El objetivo de este test fue probar el correcto funcionamiento de las sentencias *WHILE* e *IF.* Se genera un loop donde una variable decrementa y se imprime por cada ciclado. Una vez que su valor llega a 0, se imprime por pantalla el string “done” para indicar que termino el ciclado. Ésta validación se realiza por medio de una sentencia *IF* que comprueba que el valor de la variable llegó a 0.

Test 4 (strings test)

El objetivo de este test fue probar el correcto funcionamiento tanto de las operaciones entre strings (concatenación y comparación) como la lectura de strings por la entrada estándar. En el test se imprime en pantalla el string “chau” y se espera a que el usuario ingrese un string. Si este coincide con “chau” se imprime en pantalla que los strings coinciden y luego se imprime la concatenación .En caso contrario, se imprime que los strings no coinciden y luego se imprime su concatenación.

Tests 5,6,7 (errors test)

Incompatible types

Este test chequea que, al hacer una operación que toma 2 elementos de distinto tipo (un int y un string), aparezca el error adecuado de YACC al hacer la compilación.

Incompatible operand

El objetivo de este test es probar que al hacer una operación inválida para strings, aparezca el error apropiado de YACC al momento de la compilación.

Undeclared Variable

Este programa genera un error al querer utilizar una variable que aún no ha sido declarada en alguna instrucción.

Dificultades encontradas

El primer obstáculo con el que nos encontramos fue la falta de conocimientos sobre el manejo del lenguaje Yacc y el uso de Lex. A pesar de haber visto el uso de árboles en materias anteriores, no nos encontrábamos muy familiarizados en cómo utilizarlos para cambiar de un lenguaje a otro. Como consecuencia, se necesitó mucha investigación lo que atrasó en parte los tiempo del desarrollo del proyecto.

También se encontraron dificultades a la hora de escribir los ejemplos para testear el correcto funcionamiento del lenguaje. A través de los mismos, percibimos que había reglas que no estaban funcionando como esperábamos. Esto desencadenó en una búsqueda exhaustiva y en varios cambios en los archivos “yacc.y” y “lex.l” para que finalmente termine funcionando de acuerdo a nuestras expectativas. Debuggear estos archivos resultó especialmente difícil.

La definición y uso de funciones distintas de main resultó especialmente difícil ya que en un comienzo se quería evitar tener que distinguir una variable de una función usando alguna palabra clave pero lamentablemente hacer esto traia muchos conflictos y se debió utilizar la palabra “func” para distinguir que se está utilizando una función.

Finalmente, se puede observar que al ejecutar el comando make sobre el proyecto aparece un warning notificando que hay un error de shift-reduce. Como este warning no impide que el programa funcione correctamente, se decidió hacerle caso omiso por falta de tiempo para tratarlo y, principalmente, por miedo a modificar el programa y que esto último resulte en un error inesperado.

Futuras extensiones

- Soportar un manejo de los tipos numéricos más amplio, es decir, tener la posibilidad de definir números dobles o de punto flotante. Esto se considera relativamente fácil si se definen correctamente estas variables en el archivo Lex, y se cambian las operaciones aritméticas para que acepten ciertas cuentas entre los enteros, dobles y punto flotantes.
- Aumentar la cantidad de tipos de datos que soporta el lenguaje incorporando matrices, arreglos y booleanos por ejemplo. El uso de booleanos parecería bastante sencillo, pero integrar arreglos y especialmente matrices puede resultar bastante complejo según el tipo de estos.
- Tener en cuenta el rango de las funciones y las definiciones a la hora de declarar variables. Esto se podría potencialmente resolver utilizando una variable dentro de

la estructura “variable” llamada rango donde se marca en qué rango se encuentra según donde se definió esta.

- Permitir la concatenación de tipo int con tipo string. Esto parece relativamente sencillo si se hace lo mismo que se hizo con la concatenación de strings y el scanf, una función aparte para que realice la concatenación y se agregue al principio del programa.
- Extender el manejo de errores para una mayor claridad y facilidad a la hora de encontrar errores en el código. La complejidad de esta implementación depende significativamente del error y de la dificultad para encontrarlo.

Recursos

- https://www.tutorialspoint.com/c_standard_library/c_function_system.htm

Para poder compilar gcc usando el comando system

- https://www.youtube.com/watch?v=54bo1qaHAfk

Video recurso para inicializarse con el tema

- http://westes.github.io/flex/manual/Start-Conditions.html

Para poner las start conditions del lex

- https://www.tldp.org/HOWTO/Lex-YACC-HOWTO.html#toc6

En general para entender mejor el YACC y el Lex

- https://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/conflicts.pdf
- https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dh2/index.html

Para entender de donde surgían los conflictos reduce/reduce y reduce/shift

- https://github.com/faturita/YetAnotherCompilerClass/tree/master/assemblermac

Ejemplo dado por la Cátedra para comprender la estructura del proyecto mejor

- https://www.cs.ccu.edu.tw/~naiwei/cs5605/YaccBison.html#:~:text=The%20Yacc%2FBison%20parser%20detects,error%2C%20using%20the%20macro%20YYERROR.

Para debuggear el parser