Trabajo Práctico TLA

C aumentado

Nombre	Apellido	Legajo	E-mail
Marcos	Gronda	62067	mgronda@itba.edu.ar
Máximo	Rojas Pelliccia	62353	mrojaspelliccia@itba.edu.ar
Marcos	Casiraghi	62003	mcasiraghi@itba.edu.ar

Tabla de Contenidos

Introducción:

Para este proyecto nos pusimos como objetivo hacer una extensión al lenguaje de programación de C. Diseñar un compilador para interpretar declaraciones que C no reconoce para luego traducirlas a bloques de código que proveen una funcionalidad no nativa con el fin de reducir lo que el programador debe escribir. Consideramos que es una extensión de C dado que genera código C y no código de máquina.

Las funcionalidades que ofrece la extensión se basan en operaciones inmutables sobre el contenido de arreglos. Intentamos aproximar varios métodos de la librería "Stream" de Java, como Map, Reduce y Filter. También tomamos inspiración en la declaración de arreglos de python.

Desarrollo del proyecto:

Para gran parte del desarrollo backend del proyecto, nos dividimos el código a escribir en partes iguales por lo extensa que resultó ser la gramática.

Una vez tenido el frontend completo, lo primero que hicimos fue definir los nodos del árbol de sintaxis por cada variable no terminal de la gramática. En esta instancia decidimos que si un no terminal tenía más de una producción, distinguiríamos los casos con el uso de un enum.

El siguiente paso fue armar las funciones que declaran los nodos del árbol, los “grammar-action”. Como se mencionó anteriormente, se necesitaba una función por cada producción de la gramática.

Con las funciones hechas, se las podía invocar desde la gramática por cada caso. Cabe mencionar que este desarrollo fue “a ciegas”, no teníamos forma de saber de posibles errores en cualquiera de estas instancias que afectarán a la ejecución final.

En esta instancia, nos propusimos que el proyecto produjera un mínimo producto viable. Esto implicó hacer las funciones generadoras, que generan el código por cada nodo del árbol. Estas funciones corresponden una por cada variable no terminal de la gramática. Estas funciones luego se encargaban de distinguir entre los distintos casos de las producciones.

En esta instancia el proyecto ya producía un resultado final pero tenía que ser mejorado con el uso de una tabla de símbolo, aumentada por el uso de scopes. Es necesario realizar verificaciones sobre las variables que se le pasaba a la nuestra declaración especial, tenían que estar declaradas previamente.

Primero se implementó una tabla de símbolos que guardaba todas las declaraciones de variables y funciones. La tabla también tiene en cuenta: el tipo de dato, si es puntero, si es arreglo o si es una función. A partir de esto, se podía realizar una primera verificación al usarse una variable, para ver si la variable había sido declarada. Dicho esto, nos dimos cuenta que no funcionaba correctamente, dado que una variable podía haber sido declarada en cualquier scope del programa y se tomaba como válida.

Para afrontar este problema, se implementó el seguimiento de los scopes. Desde flex, cada vez que se ve una llave que abre o cierra, se pushea o popea un scope respectivamente. Por cada variable en la tabla de símbolos, ahora también se guarda en qué scope se encontraba para luego en la verificación considerar esto también. Esto ayudó a mejorar la funcionalidad del compilador y prever posibles errores para el compilador de c.

Dado que la mayoría de las funcionalidades que ofrece nuestra extensión de C se traducen a ciclos for (que internamente usan variables para iterar sobre los arreglos), era necesario implementar una función que genere el nombre de una que no exista. Se desarrolló una función que intenta a fuerza bruta nombre de variables empezando con “a”.

También se implementó la lista de errores. Dado que el orden en el que se generan los nodos del árbol sintáctico no corresponde con el mismo orden del

código de entrada, fue necesario mantener en orden la lista insertando a partir del valor de “yylineno”. Cada nodo de esta lista tiene un mensaje que se imprime en el caso de que haya habido un error con la verificación de alguna variable en la tabla de símbolos.

Hasta este punto en el desarrollo del proyecto, el código generado era en una sola línea. Con el fin de evitar tener que implementar un formater, se decidió usar una librería externa, la cual se ejecuta tras la compilación sin errores de un archivo.

Cambios de la primer entrega:

Uno de los cambios más significativos de la primera entrega fue el cambio de la sintaxis de nuestra declaración que aumenta al lenguaje de programación de c. La declaración de “REDUCE” pasó de ser escrita así:

```
<{REDUCE, array, {a += elem}}>
```

a ser escrita de la siguiente manera:

```
<{REDUCE, array, size, a, {a + elem}}>
```

El agregado de una variable size se realizó para mantener mayor consistencia con las otras declaraciones “...RANGE” que requieren el pasaje de 2 índices necesariamente. A las declaraciones de “FILTER” y “MAP” también se les añade un tamaño.

Por otro lado, también se optó por pasar la variable a donde se debe reducir el arreglo y cambiar la “lambda”. Esto se cambió por 2 motivos: Primero, hacía la implementación más simple y segundo, le daba más libertad al usuario al momento de definir que se aplicaba en cada iteración. Es importante notar que si en la expresión a ejecutarse no se utiliza la variable a donde se reduce (en este ejemplo es a), entonces el comportamiento final no va a ser el esperado para “REDUCE” dado que en cada ciclo, se pisaría el valor previo.

Otro cambio realizado es en la cláusula “FOREACH”. Previamente habíamos planeado que esta pudiera ejecutar cualquier código por cada valor del arreglo pasado. Por motivos de implementación, lo limitamos a obligatoriamente tener que llamar a una función definida por el usuario que se ejecutaría para cada elemento del arreglo. La alternativa era hacer un compilador que interprete funciones lambda pero hubiese sido complicado para implementar. Reconocemos que esto es una limitación para nuestro compilador pero consideramos que la implementación final resuelve la problemática descripta.

Con respecto a la transformación de nuestra declaración aumentada a código C, esta se mantuvo constante con lo explicado en la primera entrega. Las declaraciones se transforman en ciclos “for” que declaran un iterador nuevo. Este iterador se forma consultando la tabla de símbolos hasta que encuentre un nombre que no se haya usado en todo el código. El nombre que se intenta declarar arranca en “a” y avanza hasta encontrar uno que haya sido declarado en la tabla de símbolos. En el caso que se hayan declarado variables hasta la “z”, el iterador continua en “za”. Consideramos que la probabilidad de que un programa declare todas estas variables es muy baja. En el caso que haya algún código que lo haga, el compilador no funcionará.

Problemas encontrados:

Uno de los problemas encontrados durante el desarrollo de la parte de backend surgió a partir de cómo definimos la gramática para la inicialización de variables. Dado que consideramos como casos idénticos a la declaración de una variable (“`int i = 0`”) dentro o fuera de un bloque “for”, al momento de generar el código, nos encontramos con el problema de no saber si era necesario imprimir un ‘\n’. Para resolver esto, optamos por hacer que se encarguen las funciones generadoras; tener un argumento más en ciertas funciones que indique si es necesario imprimir una nueva línea. Este fue un ejemplo pero hubieron varios casos donde algo similar ocurría.

Fue también en esta etapa de crear las funciones generadoras donde nos encontramos con varios errores, típicamente de segmentation fault, de accesos ilícitos a variables no declaradas. Esto ocurrió porque recién en esta etapa se comprobaba que las funciones “bison-action” estuvieran bien hechas, más allá de errores de compilación. Los errores podían provenir de mal pasaje de parámetros en bison-grammar.y o incorrecta inicialización de los nodos del árbol en bison-actions.c o errores de lógica en generator.c. La incertidumbre del origen al encontrar un error hizo que sea complicada y frustrante de resolver.

Otra problemática que tuvimos que enfrentar es que, al ser la gramática tan compleja, realizar un cambio a la misma es muy costoso por todo lo que habría que cambiar posteriormente. Un ejemplo claro de esto es que en la gramática definida en “bison-grammar.y” la llamada a función no puede recibir arreglos como argumentos. Nos dimos cuenta una vez cuando ya estaba completado el backend y consideramos que no valía la pena cambiarlo y reconocemos como limitación. La gramática hace que el código sea poco modificable.

En un principio nos encontramos con una dificultad en la generación de código. ¿Cómo saber cuántos “tabs” meter a principio de línea? La forma en que lo solucionamos es, en la misma función que define un scope en flex a partir de si hay una llave que abre o cierra (“{“ o “}”), agregamos una variable global que lleva la cuenta de esto. Se le suma 1 si la llave abre y se le resta 1 si se cierra. Este valor es necesitado por la mayoría de los nodos al momento de crear el árbol de sintaxis, con lo cual fue necesario que cada nodo definido ahora tenga una variable más para guardar este valor. Por último, en la generación de código, después de cada ‘\n’, se imprimen ‘\t’ tantas veces como este valor por cada nodo. Esto no fue una solución completa del problema de la indentación porque las llaves que cierran están al mismo nivel que la del contenido que debería encapsular. Para la cláusula “switch” tampoco se puede plantear una buena indentación porque debería estar delimitado por los “:” de cada caso pero no se sabría cuándo es que termina esa indentación.

A partir de estas inconsistencias, decidimos que nuestro compilador no se iba a encargar de una correcta indentación. Sino que, llamaría a un “formatter” para que haga visualmente estético al código de salida. Las funciones generadoras ya no se encargan de imprimir nuevas líneas ni tabs. El código generado está todo en una línea y luego es corregido por el formatter que lo deja en un archivo de salida.

Futuras extensiones:

Consideramos que con este compilador pudimos aumentar el lenguaje de programación de C, reduciendo la cantidad de código que debe escribir un programador. Tomando inspiración en ciertas funciones de Java o Python, las pudimos implementar dentro del lenguaje. De todas formas, nuestro compilador tiene que poder interpretar el lenguaje de programación de C. Actualmente, acepta una gran parte del lenguaje pero no todo. Una futura extensión podría involucrar expandir lo que acepta el compilador como válido para no limitar al lenguaje de C.

Además, se podrían implementar más funciones de otros lenguajes que lo hagan más útil. De todas formas, reconocemos que es por la falta de funcionalidades de C unos de los motivos por el cual surgieron más lenguajes de programación.

Este proyecto nos dio una mayor apreciación por todo lo que tiene que contemplar el compilador de C. Son verdaderamente una cantidad extraordinaria de variables a considerar y nosotros ni siquiera raspamos la superficie de todo lo que debe suceder por atrás.

Referencias:

- El formatter que modifica al código de salida es clang-formatter