Trabajo Práctico Especial

Autómatas, Teoría de Lenguajes y Compiladores

Integrantes

Gonzalo Hirsch	59089	(ghirsch@itba.edu.ar)
Ignacio Ribas	59442	(iribas@itba.edu.ar)
Augusto Henestrosa	59189	(ahenestrosa@itba.edu.ar)
Francisco Choi	59285	(fchoi@itba.edu.ar)

Índice

Objetivo

El objetivo del trabajo práctico es el desarrollo completo de un lenguaje y su compilador, construyendo en el proceso las partes principales de un compilador: el analizador léxico y el analizador sintáctico. Para lograr esto, primero hay que definir una gramática que soporte los requerimientos propuestos, y luego, utilizando Lex como analizador léxico y Yacc como analizador sintáctico, se puede construir el compilador. En este compilador, Lex actúa como scanner y Yacc actúa como parser.

Lenguaje Desarrollado

El concepto general del lenguaje desarrollado es el de un lenguaje que se enfoca en operaciones entre arreglos. La idea se basa en facilitar operaciones que generalmente no se encuentran disponibles en todos los lenguajes de programación, como por ejemplo, la suma punto a punto entre dos arreglos, o la multiplicación de los mismos. Por ende, nuestro lenguaje se llama **Arr**.

Consideraciones Realizadas

No se realizaron consideraciones por fuera de la consigna.

Desarrollo

El desarrollo del compilador se realizó en C para tener un mejor control del bajo nivel. El manejo de memoria dinámica nos otorgó la posibilidad de realizar una mejor traducción de nuestro lenguaje. Los errores de compilación se reportan al momento de hacer chequeos de tipos, verificación de existencia de variables o verificaciones de redefinición de variables, entre otras verificaciones realizadas.

Scanner

Para el Scanner se utilizó Lex como analizador léxico para la gramática. En el archivo que consume Lex se definieron las palabras claves (tokens) que acepta nuestra gramática. Muchos tokens en nuestro caso eran similares a los de C, porque el foco de nuestro lenguaje está en las operaciones entre los arreglos, principalmente difiere en las declaraciones de los arreglos y el manejo de los mismos.

Parser

Para el Parser se utilizó Yacc como analizador sintáctico para la gramática, en el archivo que consume Yacc se definió la gramática en un formato muy similar a BNF. Al ser un parser LALR, significa que es ascendente, lo que implica que a partir de las producciones definidas, genera nodos en un Abstract Syntax Tree con atributos que nosotros luego utilizamos para hacer la traducción a C.

Traducción

Se decidió traducir nuestro lenguaje a C, y para lograr esto, ya que las operaciones entre arreglos no están definidas en C, tuvimos que hacer uso de una gran cantidad de funciones y estructuras auxiliares para poder hacer que la definición de las estructuras sea lo más simple posible. En nuestro compilador primero se genera el AST y luego durante la etapa de traducción se hacen las verificaciones de tipos necesarias. Este AST se definió de manera que los nodos sean genéricos de forma que sea lo más simple posible realizar extensiones, y la estructura del AST es la de un árbol n-ario. A partir de la raíz del árbol se van traduciendo todas las expresiones que nacen del mismo.

Descripción de la Gramática

La gramática se intentó de mantener lo más simple posible, para evitar que el lenguaje sea demasiado verboso como Java, y que sea simple como Python pero manteniendo los tipos de datos.

Punto de Entrada

Todo programa escrito en Arr debe comenzar con la palabra reservada **start** y terminar con la palabra reservada **end** de la siguiente manera:

```
start
...
...
end
```

Al igual que C, todas líneas terminan con ";"

Tipos de Datos

Nuestro lenguaje es explícitamente tipado, y maneja 5 tipos de datos:

- int - Enteros
- double - Punto flotante de doble precisión, con el punto (".") como separador decimal
- str - Cadenas de caracteres, acepta caracteres escapados como en C
- int[] - Arreglo de int
- double[] Arreglo de double

En el caso de los tipos **int**, **double** y **str** se pueden declarar o declarar e inicializar de esta forma:

```
<tipo> <nombre_de_la_variable>;
```

ó

```
<tipo> <nombre_de_la_variable> = <valor>;
```

En el caso de los tipo **int[]** y **double[]**, solo se pueden declarar e inicializar, no se pueden hacer declaraciones sin inicializaciones para estos tipos, y se hace de la siguiente forma:

```
<tipo> <nombre_de_la_variable> = [<num_1>, <num_2>, ... ];
```

Los tipos de arreglos están indexados con índices comenzando en el 0, y se pueden usar tanto en asignaciones como en expresiones de la siguiente manera:

```
<tipo_de_arreglo> <nombre_de_la_variable> = [<num_1>, <num_2>,
...];
<nombre_de_la_variable>[n] = m;
<nombre_de_la_variable>[n] = x + y + z +
<nombre_de_la_variable>[m];
```

Es importante notar que las variables son globales para hacer que sea más simple el manejo del scope de las variables. Sus nombres son conjuntos de caracteres alfanuméricos que comienzan en una letra en mayúscula o minúscula seguido de una cadena de letras o números.

Operaciones

Aritméticas

Las operaciones aritméticas fueron definidas para tener en cuenta las posibles combinaciones de tipos que pueden haber, cuando se discutan los resultados de las operaciones se va a hacer referencia a un *tipo más abarcativo*. Este *tipo más abarcativo* hace referencia al resultado entre operar con los distintos tipos posibles del lenguaje, el tipo más abarcativo entre un tipo numérico y un tipo de arreglo es un arreglo, y entre el tipo numérico int y double, el tipo más abarcativo es el double. Con esto definido, operaciones entre los distintos tipos siempre retornan al tipo más abarcativo, un ejemplo de esto es sumar un int[] y un double, el resultado es un double[].

Algo a notar es que para poder realizar operaciones aritméticas entre arreglos, los mismos tienen que tener el mismo tamaño, sinó generará un error de ejecución.

Las operaciones aritméticas definidas son:

Operación	Descripción	Ejemplo

Suma (+)	Suma los valores entre ambos operandos, devolviendo el tipo más abarcativo para cada combinación de tipos involucrados.
	
	En el caso de que un operandos sea un arreglo, se hace una suma punto a punto en ese arreglo.
		`int[] a = [1, 2, 3];`
		`a = a + 3;`
		`a -> [4, 5, 6];`
		
		`int a =`
		
		`str a = "hola ";`
		`str b = "chau";`

	En el caso de **str**, se concatenan las cadenas.	`str res = a + b;` `res -> "hola chau"`

Resta (-)	Resta los valores entre ambos operandos, devolviendo el tipo más abarcativo para cada combinación de tipos involucrados.
	
	En el caso de arreglos, la resta se hace punto a punto, y no está definida la resta cuando el operando izquierdo es un número y el derecho un arreglo.
		`double[] a = [1.4, 1];`
		`a = 10 - a; -> ERROR`
		`a = a - 10; -> OK`

Multiplicación (*)	Multiplica los valores entre ambos operandos, devolviendo el tipo más abarcativo para cada combinación de tipos involucrados.
	
	En el caso de que un operando sea un arreglo, la multiplicación se hace punto a punto, y el tipo de retorno
		`double[] arr = [1.5, 4];`
		`arr = arr * 3.5;`
		`arr -> [5.25, 14]`
		
		`int[] arr = [2, 3];`
		`double[] arr2 = [2, 2];`
		`arr = arr * arr2; -> ERROR, arr no es de tipo double[]`

División (/)	Divide los valores entre ambos operandos, devolviendo el tipo más abarcativo para cada combinación de tipos involucrados (en el caso de dividir un tipo int, sea arreglo o no, se devuelve un tipo double).
	
	En el caso de arreglos, si el operando izquierdo es un tipo numérico y el derecho es un arreglo, la división no se encuentra definida.
		`int[] arr = [4, 6, 8];`
		`double[] arr2 = arr / 2;`
		`arr2-> [2, 3, 4];`
		
		`double[] arr = [4, 6, 8];`
		`a = 10 / a; -> ERROR`
		`a = a / 10; -> OK`

Comparación

Las operaciones de comparación solo fueron definidas para tipos numéricos, es decir, no están definidas para arreglos ni para cadenas de caracteres. Las operaciones retornan 1 como verdadero o 0 como falso.

Las operaciones de comparación definidas son:

Operación	Descripción	Ejemplo
Menor estricto (<)	Si el valor del operando izquierdo es menor estricto al del operando derecho, devuelve 1, si no 0.	`int a = 10 < 5;` `a -> 0 (falso)`
Menor o igual (<=)	Si el valor del operando izquierdo es menor o igual al del operando derecho, devuelve 1, si no 0.	`int a = 4 <= 5;` `a -> 1 (verdadero)`
Igualdad (==)	Si el valor de los operandos es igual, devuelve 1, si no 0.	`int a = 4 == 5;` `a -> 0 (falso)`
Diferencia (!=)	Si el valor de los operandos es diferente, devuelve 1, si no 0.	`int a = 4 != 5;` `a -> 1 (verdadero)`
Mayor estricto (>)	Si el valor del operando izquierdo es mayor estricto al del operando derecho, devuelve 1, si no 0.	`int a = 10 > 5;` `a -> 1 (verdadero)`
Mayor o igual (>=)	Si el valor del operando izquierdo es mayor o igual al del operando derecho, devuelve 1, si no 0.	`int a = 4 >= 5;` `a -> 0 (falso)`

Lógicas

En el lenguaje se definieron operaciones lógicas, en las que solo

Operación	Descripción	Ejemplo

And (&&)	Si el valor de verdad de ambos operandos es verdadero, el resultado es 1, si no 0.	`if (4 >= 5 && 4 <= 5){`
		`  print "Es verdad";`
		`}`
		`-> En este caso NO imprime`

Or (||)	Si el valor de verdad de algún operando es verdadero, el resultado es 1, si no 0.	`if (4 >= 5 || 4 <= 5){`
		`  print "Es verdad";`
		`}`
		`-> En este caso SI imprime`

Negación (!)	Niega el valor de verdad de del operando. Solo se puede usar para negación de variables o literales, no de otras expresiones	`int a = 4 >= 5;`
		`if (!a){`
		`  print "a es falso";`

	lógicas.	`}`

Arreglos

También se definieron dos operaciones propias de los vectores, las mismas son el producto punto y el producto cruz:

Operación	Descripción	Ejemplo

Producto Punto (*.)	Hace el producto punto entre los arreglos, no hay límite de dimensiones.
	
	En caso de que las dimensiones no sean correctas, lanza un error de tamaño inválido.
		`int[] a = [1, 2, 3];`
		`int[] b = [2, 3, 4];`
		`int c = a *. b;`
		`c -> 20`

Producto Cruz (*x)	Hace el producto cruz entre los arreglos, en el caso de que sean ambos de 2 dimensiones se devuelve un arreglo de una dimensión con el resultado. En el caso de que los arreglos sean de 3 dimensiones se devuelve un arreglo de 3 dimensiones.
	
	En caso de que las dimensiones no sean correctas, lanza un error de tamaño inválido.
		`int[] a = [1, 2, 3];`
		`int[] b = [2, 3, 4];`
		`b = a *. b;`
		`b -> [-1, 2, -1]`

Control de Flujo

Para el control de flujo del programa se definió el if, donde la sintaxis debe cumplir la siguiente estructura:

```
if (<condition>){
      ...
}
[else if (<condition>){
      ...
}]
...
[else if (<condition>){
      ...
}]
```

```
[else {
      ...
}]
```

Ciclos

Se definieron dos posibles formas de realizar ciclos en el lenguaje. La primera forma es iterar sobre todos los elementos de los arreglos en un formato similar al definido por Java en el *foreach.* La segunda forma es con el uso de un while.

Una nota importante a tener en cuenta es que se puede definir variables dentro del bloque de código de ambas iteraciones, pero estas variables serán globales a todo el programa.

ForEach

Para poder hacer uso de esta forma de iteración es necesario poseer un arreglo declarado e inicializado. Esta forma de ciclar itera sobre todos los elementos de un arreglo en orden, pasando por cada uno una sola vez. Estas funciones no se pueden anidar, se generará un error de compilación indicando esto.

Hay dos versiones de la sintaxis para esta iteración, una es con bloque y la otra es con una sentencia. La primera es utilizada cuando se quieren ejecutar más de una línea de código, y la segunda es para ejecutar una sola línea de código.

La sintaxis de la iteración con bloque es la siguiente:

```
int[] arr = [...];
arr.forEach(x->{
      ...
});
```

La sintaxis de la iteración con sentencia es la siguiente:

```
int[] arr = [...];
arr.forEach(x->...);
```

While

Esta forma de iteración es la forma estándar que se puede encontrar en la mayoría de los lenguajes de programación. La sintaxis para hacer uso de la misma es la siguiente:

```
while (<condition>){
            ...
}
```

Entrada Estándar

Para hacer uso del input del usuario se definieron 3 posibles tipos de entrada de datos, estos tipos definidos son la entrada de un int, double y un str. Para indicar la lectura de entrada estándar de estos tipos existen las siguientes funciones:

Función getInt(), lee un entero de entrada estándar, si no es válido se corta la ejecución del programa:

```
int i = getInt();
```

Función getDouble(), lee un double de entrada estándar, si no es válido se corta la ejecución del programa.

```
double d = getDouble();
```

Función getString(), lee una cadena de caracteres de la entrada estándar, si no es válido se corta la ejecución del programa.

```
str s = getString();
```

Salida Estándar

Existe un mecanismo para imprimir a salida estándar, la función `print`, esta permite imprimir todos los tipos de variables. La sintaxis de la misma es la siguiente:

```
print [<var>|<literal>];
```

Con esta función se pueden imprimir tanto literales como valores de variables. Es importante notar que no se pueden imprimir resultados de expresiones.

Funciones Extra

Existe la posibilidad ante un error o si el usuario lo desea de causar la terminación normal del programa. Esto se hace a partir de la sentencia exit().

```
exit();
```

Cuyo resultado ser equivalente al exit(0) de C.

Programas de Prueba

Dentro del repositorio de Git se encuentra una carpeta con el nombre *tests*, dentro de la misma se encuentran los archivos **.arr** que se utilizan para realizar las pruebas. En el archivo README.md se encuentra una guía comprensiva como cómo compilar y ejecutar los programas de prueba.

Dificultades durante el Desarrollo

Durante el desarrollo del trabajo práctico nos encontramos con varias dificultades a lo largo del camino. Si bien muchas de estas dificultades implicaron retrasos en términos de tiempo, nos obligaron a poner en perspectiva las ideas que teníamos y cómo las queríamos ejecutar. Esto eventualmente llevó a que varias ideas se descarten mientras que otras surgían, y al mismo tiempo logramos una estructura más simple y más escalable.

Una de las primeras dificultades fue entender cómo hacer para traducir correctamente el código de nuestro lenguaje a C, el enfoque inicial fue orientado a un intérprete en vez de un compilador porque no estábamos seguros sobre cuál debería ser el flujo que debería seguir Yacc para hacer la correcta traducción. Luego de mucha investigación comprendimos que deberíamos construir un Abstract Syntax Tree que tuviera atributos para poder realizar la correcta traducción.

Una vez definido que la forma de traducción y almacenamiento era un Abstract Syntax Tree, se nos presentó la dificultad de definir la estructura de los nodos para establecer cómo se organiza el árbol en cuestión. Nuestro primer enfoque fue hacer varios tipos de nodos basados en que había que procesar, pero debido a las dificultades que esto presentaba decidimos simplificar y hacer dos tipos de nodos genéricos: Uno con hijos y otro sin hijos, agregando además variables internas para describir particularidades de los mismos. De esta forma se simplificó en gran parte la gramática y su posterior procesamiento.

A su vez, se nos presentaron conflictos del tipo reduce-reduce y shift-reduce en la gramática, especialmente en la parte de operaciones y expresiones. Tras un breve análisis logramos re definir esa sección de la gramática para evitar estos inconvenientes.

Otra dificultad encontrada fue el cómo representar un arreglo en el código C subyacente generado de forma que siempre se pueda saber el tamaño para poder realizar las iteraciones y operaciones necesarias sobre los mismos. Esto nos llevó a varias iteraciones sobre posibles diseños, hasta que llegamos a un diseño que nos permitía la reasignación y simplificaba la traducción a C.

Junto a la dificultad anterior se sumaba el desafío de tener que definir operaciones entre variables de arreglos y numéricas que podían tener diferentes órdenes de los operandos de forma que se pueda definir correctamente cada operación y no sea extremadamente la traducción. Una vez que se logró definir y probar correctamente la estructura subyacente para los arreglos, comenzar a definir y programar estas

funciones dejó de ser un desafío. En esto fue una desventaja generar código en C, porque C no contiene sobrecarga de funciones, entonces había que definir las funciones utilizadas para realizar las operaciones de forma descriptiva y única al orden de los operadores.

Sumado a lo anterior, por portabilidad, fue necesario cambiar la traducción de los statements forEach, para la cual se aprovechaba la posibilidad de definir funciones dentro de otras funciones en C, característica que no soportan todos los compiladores. Esto trajo nuevas complicaciones que llevaron a la eliminación de la característica de statements forEach anidados.

Posibles Extensiones

Hay varias extensiones que serían grandes adiciones a nuestro lenguaje, algunas de ellas eran muy complejas para implementar todas juntas en el trabajo práctico y otras simplemente no hubo el suficiente tiempo como para desarrollar. Junto a cada posible extensión se agrega una posible estimación del grado de dificultad de la misma.

Algo que sería necesario para que este lenguaje crezca son los **comentarios** ya que no fueron incluidos en esta primera versión para poder implementar más funcionalidades y opciones para los usuarios. Esta extensión implicaría un gran cambio en la gramática ya que habría que agregar la posibilidad de tener comentarios en todas las producciones ya que los comentarios pueden estar en todas partes generalmente en un lenguaje de programación estándar.

Una extensión que resultaría clave serían las **funciones**, si bien ahora mismo se pueden definir expresiones similares a un for, no se pueden definir funciones. Esto no sería muy complejo de agregar, ya que sería extremadamente similar a la implementación de la función forEach que ya existe.

Una extensión interesante podría ser la posibilidad de definir **arreglos dentro de arreglos**, es decir, matrices. Actualmente no es algo que se encuentra soportado, pero creemos que el agregado de matrices podría ser una adición que junto con el lenguaje actual podría habilitar una nueva gama de operaciones muy interesantes. Esta extensión creemos que resultaría altamente compleja de realizar, ya que implicaría una posible definición recursiva de las estructuras subyacentes en C, y esto puede llegar a ser muy complejo de manejar, más que nada por el lado del alojamiento de memoria y el correcto manejo de la misma.

Otra extensión que sería muy útil es el agregado de **propiedades a los arreglos** que permitan obtener métricas interesantes sobre los datos que contiene. Por ejemplo, propiedades como la suma de todos los elementos, o el promedio de los elementos o el mayor o menor elemento. Obviamente cada propiedad por detrás se traduce a propiedades dentro de la estructura del arreglo subyacente en C que se calculan por primera vez al momento de crear el arreglo y se actualizan con cada operación o simplemente son llamados a funciones que se calculan en el momento que se accede a esa propiedad. Esto no resultaría muy complejo de implementar ya que en parte ya se encuentra implementado, actualmente se posee la longitud del arreglo como una propiedad, aunque no existe definida ninguna función para el acceso de la misma.

Otra extensión interesante podría ser la **comparación entre arreglos**, ya que actualmente no se soporta porque se puso el foco en las operaciones aritméticas y las formas de ciclar sobre arreglos. Esto presentaba también un desafío al momento de la definición ya que depende de qué criterio se tome para definir una relación de orden entre los arreglos. Se podría usar una definición similar a la utilizada por *Matlab* que realiza comparaciones entre cada par de elementos con los mismos índices o por el orden lexicográfico. Esta extensión es extremadamente similar a las operaciones que ya vienen definidas con nuestro lenguaje, de forma que no sería muy complejo de implementar.

Por otra parte, una extensión de gran utilidad sería la **anidación de forEach**, que actualmente no soportamos. Esto implicaría un cambio en el procesamiento de las metavariables y una mayor dificultad al analizar y detectar las meta variables anidadas.

Otra extensión importante que resultaría en un mayor rango de posibilidades para el lenguaje es la **declaración de arreglos sin inicialización** ya que actualmente no está soportado. Esto implicaría un gran cambio en la estructura del código del compilador para poder también guardar el tamaño de los arreglos definidos de forma que se pueda generar una inicialización con los elementos en 0 para evitar problemas. Aunque es un cambio extenso, no es uno muy complejo.

Otra extensión clave sería la **negación de expresiones lógicas**, por el momento no está definido, pero sería una gran extensión y no sería complicado agregarlo.

Una última extensión interesante podría ser **distintos tipos de arreglos** como por ejemplo arreglos que sean vectores de 2 dimensiones y tengan definidas operaciones propias de arreglos de 2 dimensiones. También se podría usar para cuaterniones[1] como en el caso del framework de *Unity* para *C#*, y poder definir las operaciones propias para el manejo de estas estructuras que sirven para realizar cálculos físicos. La dificultad de esta extensión depende de la estructura que se quiera implementar, de forma que estimamos que el agregado de una estructura como un vector de 2 dimensiones sería mucho más simple que el agregado de un cuaternión.

[1] Estructuras con una parte escalar y otra vectorial, se utilizan en gran medida para poder representar rotaciones en espacios de 3 dimensiones. Una aplicación es en el desarrollo de videojuegos por ejemplo. Referencia: https://en.wikipedia.org/wiki/Quaternion#:~:text=In%20mathematics%2C%20the%20quaternions%20are,of%20two%20quaternions%20is%20noncommutative.

Referencias

IBM Knowledge Center, “Generating a lexical analyzer using lex” - https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.bpxa600/genlex.htm
IBM Knowledge Center, “Generating a parser using yacc” - https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.bpxa600/genyac.htm