Autómatas, Teoría del Lenguaje y Compiladores 72.39

Trabajo Práctico Especial

Integrantes:

- Boccardi, Luciano 59518 lboccardi@itba.edu.ar
- Piñeiro, Eugenia 59449 epineiro@itba.edu.ar
- Puig, Tamara 59820 tpuig@itba.edu.ar
- Ricarte, Matías 58417 mricarte@itba.edu.ar

Año: 2020

Cuatrimestre: 20202Q

Idea Subyacente y Objetivo del lenguaje

Clux se originó tras varias sesiones de brainstorming grupales, donde notaremos 2 temáticas típicas entre las ideas que sugerimos: código corto (en un sentido más literal) y lenguajes esotéricos. Estas dos ideas fueron las que nos ayudaron a elegir el estilo del lenguaje, buscando acortar líneas de código, a pesar de que pueda volverse difícil de leer.

Tras analizar algunas competencias de códigos breves y algunas comunidades como Dwitter[1] (códigos de JavaScript en 140 caracteres), se determinó que no nos parecía tan interesante un código mínimo, sino uno breve y conciso, pero que conserve la capacidad de ser autodescriptivo, que no se requiera ejecutarlo para comprender cómo funciona. Jugando un poco con tipos de datos conocidos, e intentando aislar lo máximo posible el lenguaje de programación del lenguaje humano, llegamos a una reducción que consideramos adecuada.

Por ejemplo, para declarar una variable primero se escribe una letra para indicar el tipo de variable (i para enteros, c para caracteres, s para strings), seguido del símbolo $ y luego el nombre de la variable.

Por lo que las típicas declaraciones en Clux se verian asi:

```
i$i=1.
c$o='w.
s$grass='wwwwwwwwwwwwwww'.
```

Dado que con el parser convertimos código escrito en Clux a C, decidimos agregar algunas funciones de las librerías estándar: impresión a stdout, lectura desde stdin, una función para conseguir la longitud de un string y una función para convertir strings de caracteres que representen números a entero.

Creemos que estas son las funciones mínimas para que se puedan programar varios algoritmos típicos.

Consideraciones realizadas

A la hora de enriquecer el trabajo se decidieron agregar las siguientes funcionalidades:

- Existen tres tipos de datos, no únicamente dos como está propuesto en el enunciado, `ints, chars y strings.`
- Se proveyeron funciones para poder simplificar el código del main.
- Se implementaron arrays de los tres tipos de datos existentes.
- Considerando las necesidades básicas a la hora de realizar un programa en un lenguaje no tan rico como C, decidimos agregar dos de las funciones que consideramos podrían tener más utilidad para el usuario, `atoi()` y `strlen()`.
- Se implementaron constantes para mayor similitud con el lenguaje padre, C.

Descripción del desarrollo del TP

El desarrollo del trabajo práctico consistió de dos partes en lo referente a la organización del equipo:

[1] https://www.dwitter.net/

En primera instancia se decidió trabajar simultáneamente sobre lo que fue el inicio de nuestro proyecto. Para esto, utilizamos las tecnologías provistas por `Discord` y el `LiveShare` que permite `Visual Studio Code` donde se permite a los integrantes escribir código a la vez. Esta decisión se tomó no solo para poder agilizar el trabajo y ahorrarnos posibles problemas de merging sino para que todos contasemos con el mismo conocimiento respecto del funcionamiento del compilador y sus avances. La razón primordial la basamos en la premisa de "crear una gramática común, que comprendamos en su totalidad", y no juntar partes aisladas que representan distintas funcionalidades.

Una vez terminado lo que se consideró como la base del trabajo nos dividimos por funcionalidad para hacer todas las pruebas necesarias y pertinentes para asegurar el correcto funcionamiento del proyecto, arreglando bugs y agregando las últimas funcionalidades que a medida se hacían las pruebas se encontraban faltantes.

Durante todo el desarrollo se fueron creando programas de prueba a partir de los cuales no solo se lograba testear el TPE sino que también se descubren posibles mejoras o agregados que debían de tenerse en cuenta.

Estructura del Proyecto

El proyecto se divide en tres grandes secciones: la gramática, el scanner y el compilador.

Se desarrolló un scanner en Lex (*scanner.l*) que procesa los lexemas de nuestro lenguaje. La gramática se definió en Yacc (*grammar.y*) y utiliza funciones para manejar la lógica del proyecto *(tree.c),* como por ejemplo almacenado de variables, liberar recursos de memoria alocados, comprobaciones de correspondencia entre tipos de datos, entre otros.

El compilador está desarrollado en el lenguaje C (*compiler.c*) recibe la dirección donde está el archivo a procesar y luego genera código intermedio (*intermediate.c*) y de salida en C usando *gcc*.

Clux Cheat Sheet

1. Tipos de datos

`i$` (INT), `c$` (CHAR) y `s$` (STRING)

Ejemplo	
Clux	Equivalencia en C
i$a=1. c$a='c. s$a='hello'.	int a= 1; char a = 'c'; char * a = "hello";

2. Constantes

`i#` (INT), `c#` (CHAR) y `s#` (STRING)

Ejemplo	
Clux	Equivalencia en C
i#MAX 1 c#A ’c s#B ’hello’	#define MAX 1 #define A ‘c’ #define B “hello”

3. Delimitadores

Las líneas terminan con un . (punto), excepto aquellas que definen una constante

Ejemplo	
Clux	Equivalencia en C
i$=0. (a<1)?->{ a = 1+2 . }	int i = 0; if(a<1){ a = 1+2; }

4. Operadores

- Aritméticos: + - * / %
- Relacionales: < <= > >= ~ (equal) !~ (not equal)
- Lógicos: | (or) & (and)
- Asignación: =

Ejemplo	
Clux	Equivalencia en C
a ~ b a !~ b a | b a & b	a == b a != b a | | b a && b

5. Bloque condicional

```
(comparación)?->{código}
```

Ejemplo	
Clux	Equivalencia en C
(i<2)?->{a=1+2.}	if(i<2){ a=1+2;}
(i<2)?->{a=1+2.}:{a=1.}	if(i<2){ a=1+2;}else{a=1;}

6. Bloque de repetición hasta condición

```
(condición)?-->{código}
```

Ejemplo	
Clux	Equivalencia en C
(i<2)?-->{a=1+2.}	while(i<2){ a=1+2;}

7. Palabras reservadas

S representa a la declaración del `int` **`main`**`(void){}`
F representa a la declaración de **funciones**.
G es el caracter que utilizamos para la **lectura por entrada estándar**.
P es el caracter que utilizamos para la **escritura por salida estándar**.
C es el caracter que utilizamos para la conversión de **string a int.**
L es el caracter que utilizamos para obtener la **longitud de una cadena**.
R es el caracter que se utiliza para el **`return`**.

8. Funciones

Se utiliza la palabra reservada F

```
tipo_de_dato F nombre_de_funcion(tipo arg1, tipo arg2,...)
```

Ejemplo		
	Clux	Equivalencia en C

Definición	i$ F hello(s$ str1, c$ ch){
	… code ....
	R 1.
	}
		int hello(char * str1, char ch){
		… code ....
		return 1;
		}

Prototipo	i$ F hello(s$ str1, c$ ch).	int hello(char * str1, char ch);
Llamado	s$str1. c$ch. i$a = hello(str1, ch).	char * str1; char ch; int a = hello(str1, ch);

9. Arrays

Soporta arrays de integers, chars y strings.

```
tipo nombre_tamaño
```

Ejemplo	
Clux	Equivalencia en C
i$array_4=1,2,3,4. c$array_4='a,'b,'c,'d. s$array_4="clux","es","muy","bueno"	int array[4]={1,2,3,4}; char array[4]={'a','b','c','d'}; char *array[4]= {"clux","es","muy","bueno"};

10. Main

Se utiliza la palabra reservada S

Ejemplo	
Clux	Equivalencia en C

S {
… program ....
 	
R 0.
}
	int main( ){
	… program ....
	
	return 0;
	}

11. Stdout

Se utiliza la palabra reservada P.

```
P[ texto 'nombre_var' texto ].
```

Ejemplo	
Clux	Equivalencia en C
s$a='Clux'. i$num= 1. c$var = 'n.	char * a="Clux"; int num= 1; char var = 'n'.

P[‘a’ es el lenguaje ‘c’umero ‘num’].	printf(“%s es el lenguaje %cumero %d”, a, c, num);
Output: Clux es el lenguaje numero 1	

12. Stdin

Se utiliza la palabra reservada G.

```
G(nombre_var , max_caracteres_a_leer).
```

Cabe destacar que no es necesario definir previamente la variable que recibe por argumento.

Ejemplo	
Clux	Equivalencia en C
P[Enter a word:]. G(str, 20). P[‘str’].	char str[20]; fgets(str,20,stdin); printf(“%s”, str);

Descripción de la gramática.

NOTA: DATATYPE no es un símbolo o producción real, es una notación que se usara para hacer más legible esta sección.
Se definió una gramática con las siguientes producciones:

start → constant start | function start | S{program }
constant → i# ALPHA DIGIT | c# ALPHA ‘ALPHA | s# ALPHA ‘ALPHA’
function → DATATYPE F ALPHA (params){program} | DATATYPE F ALPHA (params).
params → type ALPHA | type ALPHA, params | λ
type→ i$ | c$ | s$
args → assignment | assignment,args | call | λ
call → ALPHA (args)
program → var. program | call. program | RETURN assignment. | get program | STDOUT out. program | (rule operator rule)? WHILE { program } program | (rule operator rule)? IF {program} : { program} program | λ
get → STDIN(ALPHA, DIGIT).
out → SPACE_ALPHA out | ‘ SPACE_ALPHA ‘ out | ALPHA out | ‘ ALPHA ‘ out | λ
var→ i$ ALPHA = DIGIT operation | i$ ALPHA = ALPHA operation | s$ ALPHA = STRING_VALUE | DATATYPE ALPHA = ALPHA | c$ ALPHA = ‘ALPHA | c$ ALPHA = ’DIGIT | i$ ALPHA_DIGIT = digit_array | s$ ALPHA_DIGIT = alpha_array | c$ ALPHA_DIGIT = char_array | i$ ALPHA = CONVERT(ALPHA) | i$ ALPHA =LENGTH(ALPHA) | DATATYPE ALPHA = ALPHA_DIGIT | DATATYPE ALPHA = ALPHA_ALPHA | DATATYPE ALPHA |

DATATYPE ALPHA_DIGIT | DATATYPE ALPHA_ALPHA | ALPHA = ALPHA | ALPHA = ‘ALPHA | ALPHA = STRING_VALUE | ALPHA = var_init operation | ALPHA = ALPHA_DIGIT | ALPHA = ALPHA_ALPHA | ALPHA =CONVERT(ALPHA) | ALPHA=LENGTH(ALPHA) | DATATYPE ALPHA = call | ALPHA = call | ALPHA_DIGIT = DIGIT | ALPHA_DIGIT = ‘ALPHA | ALPHA_DIGIT = ‘DIGIT | ALPHA_DIGIT = STRING_VALUE | ALPHA_ALPHA = ALPHA | ALPHA_ALPHA = DIGIT | ALPHA_ALPHA = STRING_VALUE | ALPHA_ALPHA = ALPHA_ALPHA | ALPHA_ALPHA = ‘ALPHA | ALPHA_ALPHA = ‘DIGIT

operation → op DIGIT operation | op ALPHA operation | λ

assignment → ALPHA | DIGIT | STRING_VALUE | ‘ALPHA | ALPHA_DIGIT| ALPHA_ALPHA

var_init → ALPHA | DIGIT

alpha_array → STRING_VALUE , alpha_array | STRING_VALUE

char_array → DIGIT , digit_array | DIGIT

rule → (rule op rule) | assignment

operator → op | < | > | <=| >= | ~ | !~ | | | &

op → + | - | / | * | %

DATATYPE → i$ | c$ | s$

Dificultades encontradas en el desarrollo.

En un principio resultó complejo entender cómo encajaban todas las partes juntas: el scanner, la gramática y el compilador.

Se encontraron dificultades para decidir qué lenguaje era conveniente usar como lenguaje intermedio y cuál como salida del compilador.

Aunque se agregó un tipo de datos, se notó la falta de punteros a la hora de querer implementar algoritmos en programas de prueba (por ejemplo, el programa de ejemplo `hanoiSort` originalmente iba a ser una implementación del algoritmo con el mismo nombre descrito por David Morgan-Mar en 2006, donde sugiere que el algoritmo está optimizado según la ley de Murphy).

Una de las mayores dificultades a la que nos enfrentamos fue a la hora de garantizar que el compilador gcc no tuviese ningún error. Si bien esto era completamente razonable, para poder realizar el correcto testeo de todas las posibles implementaciones del lenguaje nos vimos en la necesidad de priorizar el tiempo y la prevención de cualquier posible error a, entre otros, tener un código ameno, con una gramática pequeña como en un principio nos habíamos planteado. Tuvimos que deshacer varios de los cambios que habíamos realizado, como el factoreo de ciertos puntos de nuestra gramática, en medida de ahorrarnos tiempo a la hora de comprobar inicialización de variables, quedándonos a fin de cuentas mucho código repetido.

Adicionalmente a este inconveniente le siguió que poseemos a la hora de compilar un warning de reduce/reduce que si bien es correcto, no pudimos eliminar dado que este no considera los chequeos que hacemos por detrás para saber si la gramática es realmente correcta o no. En particular, esto sucede por el siguiente escenario:

a = a+b.

Solo puede existir ese tipo de producción si la variable a la izquierda del igual es de tipo `int`. Pero, esta variable como el resto de las variables para nuestro lenguaje se considera

como el terminal ALPHA. Esto, choca con la producción que busca atajar los siguientes casos:

a = b.

Siendo a y b del mismo tipo pero no necesariamente enteros.

Futuras extensiones

En un futuro se podría agregar al lenguaje el manejo de matrices para todos los tipos de datos soportados. Sería similar a la declaración de un array pero con dos guiones:

```
i$array_i  i$matrix_i_j
```

También se podrían incorporar comentarios. Consistiría en encerrar una serie de palabras entre guiones:

```
- este es un comentario -
```

Se podría incorporar el tipo de datos float, definiendo con una “f” al principio. Lo complejo sería separar la parte entera de la mantisa ya que usamos el punto para finalizar una línea y la coma como separador entre elementos de un array. Por lo tanto, implicaría hacer un refactoreo de esas dos funcionalidades o usar un carácter distinto pero con el que usuario no estaría acostumbrado.

```
f$var
```

A su vez, se podrían definir *structs* de la siguiente manera:

Ejemplo	
Clux	Equivalencia en C

`name^{ i$a. s$b. }`
`S{`
`  name$a.`
`  a^a=1.`
`  a^b=2.`
`}`
	`struct name{ int a; char*b; }`
	`int main(){`
	`  struct name a;`
	`  a.a=1;`
	`  a.b=2;`
	`}`

También se podrían acortar las operaciones haciendo uso de una notación como i++ , i+=2, i-=4, i/=2 … etc, y poder declarar numerosas variables del mismo tipo de la siguiente forma; i$i,a,b. lo cual sería equivalente en C a hacer int i,a,b.

Referencias

Hanoi Sort - https://www.dangermouse.net/esoteric/hanoisort.html