Autómatas, Teoría de Lenguajes y Compiladores

TPE

Chipa Compiler

Camila Mariana Ponce, 59220
Maria Victoria Conca, 58661
Roberto José Catalán, 59174
Uriel Mihura, 59039

Índice

Objetivo

El objetivo de este trabajo es desarrollar un nuevo lenguaje junto con su compilador. Para esto se creó el lenguaje “Chipa”, este es un lenguaje simple y fácil de entender para quienes hablan solamente español y están recién empezando a programar. Su sintaxis es simple y cómoda para los usuarios acostumbrados a programar y también para principiantes.

Chipa es un lenguaje similar a aquellos cómo C o Rust, y está implementado completamente en español. Es una alternativa para aquellas personas que no conocen el idioma inglés, o incluso para niños, ya que es muy sencillo de usar.

Lleva su nombre gracias a la comida típica de muchos países de América latina, ya que está pensado para personas de habla hispana.

Consideraciones Realizadas

A lo largo de todo el trabajo se intentó tener en cuenta todos los posibles errores de compilación que podrían ocurrir, de esta forma se pueden evitar los errores que generaría el output en C. Se optó por abortar el programa cuando ocurra un error de compilación indicando donde fue el error y por qué sucedió.

Se procuró mantener siempre una sintaxis simple, para evitar que nuestro lenguaje sea difícil de entender y de usar por los usuarios.

El lenguaje Chipa cumple con todo lo básico pedido por el enunciado, y además incorpora una nueva funcionalidad que permite concatenar strings.

Se realizó un benchmarking tomando cómo ejemplo el ejemplo 1 de nuestro trabajo, comparándolo con un programa en C cuyo objetivo es el mismo:

Programa en Chipa

```
uri@DESKTOP-FK2EC9H:~/TLA$ ./ejemplo1
inserte un numero, y te diré si es primo:
117
El numero no es primo
Time elapsed: 0.000345 seconds
```

```
uri@DESKTOP-FK2EC9H:~/TLA$ ./ejemplo1
inserte un numero, y te diré si es primo:
167
El numero es primo
Time elapsed: 0.000456 seconds
```

Programa en C

```
uri@DESKTOP-FK2EC9H:~/TLA$ ./alt_code
Enter a positive integer: 117
117 is not a prime number.
Time elapsed: 0.000217 seconds
```

```
uri@DESKTOP-FK2EC9H:~/TLA$ ./alt_code
Enter a positive integer: 167
167 is a prime number.
Time elapsed: 0.000286 seconds
```

Se notó que el código en C pudo resolver el problema con tiempos de ejecución menores.
Y se identificó que un factor de esto es que el código en C pudo utilizar el comando break; para salir del bucle *while* una vez encontrado el número primo, un comando que Chipa no tiene implementado. (aún).

Desarrollo del lenguaje

Para poder realizar el compilador se crearon sus componentes principales utilizando Lex para hacer el analizador léxico y Yacc para el analizador sintáctico (parser).

Lex se encargaría de reconocer los lexemas del lenguaje definidos por la gramática, mientras que Yacc se encarga del parseo.

Para almacenar las variables y sus nombres se creó una lista encadenada en la cual se guarda el nombre de cada variable y su tipo (numero o texto).

Una vez creado el lenguaje y su compilador se hicieron cinco programas de prueba:

1. Ejemplo 1: Dado un número que se pide al usuario me dice si este es primo o no.
2. Ejemplo 2: Programa que indica cuál es el mayor de tres números.
3. Ejemplo 3: Se pide al usuario que ingrese un texto y se lo concatena con un mensaje.
4. Ejemplo 4: Dado un número imprimir su factorial.
5. Ejemplo 5: Incrementa un contador mientras se cumple cierta condición.

Para poder trabajar nos manejamos con un repositorio de GitHub en el cual ibamos actualizando nuestros cambios, y además usamos una extensión llamada Live Share para VS Code. De esta manera pudimos implementar nuestro trabajo fácilmente y siempre trabajando en equipo. Por otra parte, consultamos el material teórico brindado por la cátedra sobre Lex y Yacc para poder resolver el trabajo correctamente.

Gramática

La gramática empleada cuenta con una lógica similar a la de C y tiene las siguientes características. Para comenzar un programa se necesita utilizar la sentencia *receta:* para indicar el punto de entrada

del programa. En cuanto a tipos de datos se tiene *numero* para las variables de números enteros y *texto* para variables que sean del tipo string (cadena de caracteres). Cabe aclarar que nuestro tipo numérico es equivalente al tipo int de C, por lo que tiene las mismas propiedades y limitaciones que este.

Las instrucciones que se pueden hacer son las siguientes: declarar variables, asignar variables, imprimir texto o variables. Las sentencias de control con las que cuenta esta gramática son un bloque condicional y un bloque Do-While.

Además esta gramática permite realizar operaciones básicas, como sumar, restar, dividir, multiplicar y obtener el resto de una división. También permite comparar números o incluso estas operaciones. Soporta expresiones booleanas junto con los conectivos *o*, *y*, *no*.

Por otro lado, puede leer tanto números cómo texto que el usuario ingrese por teclado y se soporta la funcionalidad de concatenar dos strings.

Finalmente, el delimitador que se utiliza entre cada instrucción es el *;* cómo se lo hace en C.

A continuación se muestra la gramática empleada.

G=<NT, T, S, P>

T = {FIN_LINEA, FIN, DOS_PUNTOS, MÁS, MENOS, POR, DIVIDIDO, MOD, VERDADERO, FALSO, MENOR, MAYOR, MENOR_IGUAL, MAYOR_IGUAL, IGUAL, PARENTESIS_ABRE, PARENTESIS_CIERRA, MIENTRAS, HAZ, SI, COMILLA, Y, O, NO, IMPRIMIR, SINO, VAR_NUMERO, VAR_TEXTO, TEXTO, NUMERO, NOMBRE, ASIGNACION, RECETA, LEER, CONCAT, COMA, DISTINTO}

NT = {begin, code, end, instrucción, instrucciones, declaracion, asignar, asignacion_num, asignacion_text, declara_y_asigna, print, read, dec_nombre_st, nombre_st, asignacion_st, valor, texto_st, operación, operador, parentesis_st_abre, parenteisis_st_cierra, super_si, si_st, entonces_haz, fin_si, super_haz, haz_st, fin_haz, mientras_st, fin_mientras, sentencia_booleana, sentencia_logica, sentencia_not, boolean, sentencia_comparativa, comparador, read, concat, control_logico, nombre_str_st, dec_nombre_st, coma_st, concat_op, operacion_texto, concat, coma_st, concat_op2 }

P = {

S → begin code end

begin → RECETA

end → λ

code → λ | instrucciones

instrucciones → instruccion FIN_LINEA | instruccion FIN_LINEA instrucciones | control_logico | control_logico instrucciones

instruccion → declaración | asignar |declara_y_asigna | print | read

declaracion → dec_nombre_st

asignar → nombre_st asignacion_num | nombre_st asignacion_text

asignacion_num → asignacion_st valor

asignacion_text → asignacion_st texto_st

nombre_st → NOMBRE

nombre_str_st → NOMBRE

dec_nombre_st → VAR_NUMERO NOMBRE | VAR_TEXTO NOMBRE

asignacion_st → ASIGNACION

texto_st → TEXTO

declara_y_asigna → dec_nombre_st asignacion_text |dec_nombre_st asignacion_num

print → IMPRIMIR PARENTESIS_ABRE TEXTO PARENTESIS_CIERRA | IMPRIMIR PARENTESIS_ABRE NOMBRE PARENTESIS_CIERRA

operacion → valor operador valor

operador → MAS | MENOS | POR | DIVIDIDO | MOD

parentesis_st_abre → PARENTESIS_ABRE

parentesis_st_cierra → PARENTESIS_CIERRA

valor → nombre_st | NUMERO | parentesis_st_abre operacion parentesis_st_cierra | λ

control_logico → super_si | super_haz

super_si → si_st sentencia_booleana entonces_haz instrucciones fin_si

si_st → SI

entonces_haz → DOS_PUNTOS

fin_si → FIN

super_haz → haz_st instrucciones fin_haz mientras_st sentencia_booleana fin_mientras

haz_st → HAZ

fin_haz → FIN

mientras_st → MIENTRAS

fin_mientras → FIN_LINEA

sentencia_booleana → boolean |boolean sentencia_logica boolean |parentesis_st_abre sentencia_booleana parentesis_st_cierra sentencia_logica sentencia_booleana |boolean sentencia_logica parentesis_st_abre sentencia_booleana parentesis_st_cierra |sentencia_not

parentesis_st_abre sentencia_booleana parentesis_st_cierra |sentencia_not boolean |parentesis_st_abre sentencia_booleana parentesis_st_cierra |sentencia_comparativa;

sentencia_logica → Y | O

sentencia_not → NO | NO sentencia_not

boolean → VERDADERO | FALSO

sentencia_comparativa → valor comparador valor

comparador → MENOR | MAYOR | MAYOR_IGUAL | MENOR_IGUAL | IGUAL | DISTINTO

read → LEER PARENTESIS_ABRE NOMBRE PARENTESIS_CIERRA

concat → concat_op operacion_texto

operacion_texto → PARENTESIS_ABRE valor_texto coma_st concat_op2 valor_texto PARENTESIS_CIERRA

valor_texto → nombre_str_st | TEXTO

coma_st → COMA

concat_op → CONCAT

concat_op2 → λ

}

<u>En resumen, la sintaxis de Chipa es la siguiente:</u>

- **Declaración y asignación de variables:**

```
numero a;
texto b;
a=5;
texto c=”hola”;
```

- **Bloque if:**

```
si condición:
     instrucción 1;
     instrucción 2;
             ….
     instrucción n;
fin
```

- **Do-while**

```
haz:
      instrucción 1;
      instrucción 2;
              ….
      instrucción n;
fin
mientras condición;
```

- **imprimir a pantalla y leer de teclado**

```
imprimir(“Hola”);
numero a=2;
imprimir(a);

texto b;
leer(b);
```

- **Operaciones y sentencias**

```
numero a = 3+2;
si n == 2:
mientras (i<=m)y(flag distinto 0);
numero b = 10 mod 3;
```

Dificultades Encontradas

La mayor dificultad encontrada fue a la hora de implementar y hacer funcionar correctamente la lista para almacenar las variables. Tuvimos varias instancias de errores o “segmentation fault” relacionados a este tema. Con ayuda de los ayudantes y profesores de la materia pudimos solucionarlos via mail o clase de consulta.

Futuras extensiones

Chipa ofrece una funcionalidad para concatenar dos strings, la cual funciona correctamente si se la usa con dos strings cómo parámetro, pero no se puede pasar cómo parámetro la concatenación de dos strings. Es decir:

```
concatenar (“hola”, “chau”);
```

Funciona correctamente, pero

```
concatenar (“hola”, concatenar(“hola”, “chau”);
```

No funciona correctamente. A futuro nos gustaría poder terminar de implementar esta funcionalidad.

Otra extensión que se considera hacer a futuro es poder trabajar con tipos de datos flotantes, que se pueda diferenciar entre un número decimal y un entero.

Asimismo, se podría agregar a la gramática que esta permita trabajar con vectores. Esta es una implementación más difícil pero que agregaría más funcionalidad a este lenguaje.

Finalmente, por lo descubierto en el testeo de benchmarking, podría ser útil agregar al lenguaje la funcionalidad de break; para poder escapar instantáneamente de ciclos en marcha.

Referencias

- Para la implementación de la lista nos basamos en: Implementacion LinkedList en C
- Clases y apuntes de Lex y Yacc encontrados en campus.
- Programa en C que se utilizó para el benchmark test: https://www.programiz.com/c-programming/examples/prime-number