Autómatas, Teoría de Lenguajes y Compiladores

Trabajo Práctico Especial:

"Citronella"

Comisión: S

- Agustina Osimani 57526
- Juan Godfrid 56609
- Pablo Radnic 57013
- Ignacio Negro Caino 57509

Fecha de entrega: 1 de Diciembre de 2018

Índice

Idea Subyacente y Objetivo del Lenguaje

Citronella es un lenguaje de programación de tipo imperativo. El objetivo del equipo fue desarrollar un lenguaje que permita ejecutar instrucciones de manera secuencial, declarar variables de definidos tipos y soportar las aptitudes mínimas necesarias de un lenguaje de programación como bloques condicionales y ciclos.

Para la definición de la gramática optamos por no alejarnos demasiado de lo intuitivo; las instrucciones son palabras en inglés de lectura y escritura rápida. También reutilizamos el manejo de operadores booleanos y asignaciones de C que nos parece la forma óptima para asegurar la rapidez de lectura y comprensión.

Desarrollo

Lo primero que se hizo fue definir la gramática del lenguaje. Los objetivos estaban claros, hubo una puesta en común para definir cómo se lograrían.

Uno de los integrantes del grupo, ferviente fanático del agua acompañada con Citronella, propuso el nombre. El resto del equipo lo aceptó.

El primer paso para comenzar con la configuración fue capacitarse con LEX y YACC. Esto se logró leyendo el libro *“lex & yacc”* de *John R Levine.* Los primeros capítulos de este libro contienen ejemplos que facilitan la comprensión de las herramientas.

A partir de los ejemplos se comenzó a configurar la gramática y la sintaxis del lenguaje como se había definido en un principio. En esta instancia se fueron descubriendo errores y detalles que debían ser corregidos respecto de la definición inicial, por lo cual el lenguaje se fue adaptando a medida que surgieron los imprevistos.

Finalmente se hizo la capa de revisión de variables profundizada en la sección de dificultades encontradas.

Descripción de la Gramática

Tipos de Variables

Tipo	Declaración e Inicialización
Número Entero	num varName = 1
Cadena de Caracteres	text varName = “text”
Booleano	bool varName = true

Operadores Aritméticos

Operador	Descripción	Ejemplo
+	Calcula la suma de dos operadores	A + B
-	Calcula la resta de dos operadores	A - B
.	Calcula el producto de dos operadores	A . B
/	Calcula el cociente de dos operadores	A / B

Operadores Relacionales

Operador	Descripción	Ejemplo
==	Evalúa si hay igualdad en los operadores	A == B
not (A==B)	Evalúa si hay desigualdad en los operadores	not (A == B)
>	Evalúa si el operador izquierdo es mayor al derecho	A > B
<	Evalúa si el operador derecho es mayor al izquierdo	A < B
>=	Evalúa si el operador izquierdo es mayor o igual al derecho	A >= B
<=	Evalúa si el operador derecho es mayor o igual al izquierdo	A <= B

Operadores Lógicos

Operador	Descripción	Ejemplo
and	Operador Lógico AND. Evalúa si ambos operadores tienen valores distintos a cero.	A and B
or	Operador Lógico OR. Evalúa si alguno de los operadores tiene valores distintos a cero.	A or B
not	Operador Lógico NOT. Niega el valor del operador.	not A

Delimitadores y Comentarios

Los comentarios multilínea en Citronella utilizan la siguiente sintaxis:

```
#- Comentario -#
```

El carácter delimitador en Citronella es el siguiente:

`\n (Ascii 20).`

Bloques Condicionales

Los bloques condicionales en Citronella utilizan la siguiente sintaxis:

```
expresión booleana
? declaración 1
  declaración 2
  ...
  declaración n

! declaración 1 bis
  declaración 2 bis
  ...
  declaración n bis
```

Ciclos

Los ciclos en Citronella utilizan la siguiente sintaxis:

```
repeat declaración 1
  declaración 2
  ...
  declaración n
 while expresión booleana
```

Lectura de Entrada Estándar

La lectura de la entrada estándar en Citronella utiliza la siguiente sintaxis:

```
read
```

Escritura a Salida Estándar

La escritura a la salida estándar en Citronella utiliza la siguiente sintaxis:

```
show varName
show "texto"
show expresión booleana
show expresión numérica
```

Dificultades Encontradas en el Desarrollo

Comportamiento no esperado de Citronella

En una determinada instancia del desarrollo se observó que la gramática no se comportaba de la forma esperada. Luego de un tiempo de análisis se descubrió que la gramática, en su diseño original contenía múltiples conflictos de tipo Reducción-Desplazamiento los cuales YACC intentaba resolver internamente.
Una vez descubierto el origen del error se factorizaron algunas reducciones para reducir el número de conflictos lo cual solucionó el comportamiento no esperado existente.

Asignación de tipos inválidos

En una primera versión Citronella permitía la siguiente secuencia de instrucciones:

```
num a = 3
a = "Texto"
```

Esta ejecución es sintácticamente válida por lo cual el compilador no reportaba ningún error al analizarla.
Sin embargo Citronella en definitiva se compila a C y el código C generado no era válido debido al conflicto de tipos de datos.
La solución a este problema fue utilizar un Mapa que almacena el nombre de las variables junto con su tipo al momento de declaración. Se evalúa el contenido del mapa al hacer asignaciones, esto nos permite detectar errores de tipo de datos en el compilador de Citronella previo a la generación de código C.

Futuras Extensiones

Reflecciones

El equipo se encuentra satisfecho con el lenguaje diseñado, consideramos que los objetivos planteados se lograron con éxito. No obstante Citronella contiene varias limitaciones que lo alejan de los lenguajes imperativos de uso cotidiano.

Limitaciones

Asignación de una constante negativa

Si bien Citronella permite almacenar números negativos dentro de las variables, debido al parseo la instrucción:

```
num A = -3
```

se considera un error de sintaxis. Si bien esto no resulta en una limitación de la potencia del lenguaje, resulta en una mala experiencia de uso para el programador.

Bloques Condicionales

Los bloques condicionales en Citronella tienen una sintaxis restrictiva por lo cual el lenguaje obliga a programadores a tener un bloque de evaluación positiva (if) seguido obligatoriamente por un bloque de evaluación negativa (else).

Estructuras de Datos

Citronella contiene los tres tipos de variables básicos definidos anteriormente pero no tiene forma de definir estructuras de datos auxiliares o customizadas orientadas a las necesidades del programador.

Extensiones

El principal objetivo a futuro es corregir las limitaciones identificadas en el análisis anterior. Una vez terminado esto podremos agregar nuevos tipos de datos como vectores y estructuras y funciones que garanticen sincronicidad.

Benchmarking

Se pretende comparar la velocidad de ejecución de la devolución del factorial de 6.

Implementación en Citronella

```
begin

num x = 5
num y = 6

repeat y = y . x
  x = x - 1
while x > 0

show "El factorial de 6 es: "
show y

end
```

Compilación

```
./Citronella_compile Factorial_example.Cit factorialCit
```

Implementación en C++

```
#include <iostream>
using namespace std;

int main() {
  int n = 6;
  long long factorial = 1;

  for (int i = 1; i <= n; ++i) {
          factorial *= i;
  }

  cout << "El factorial de 6 es: " << factorial;
  return 0;
}
```

Compilación

```
g++ factorial.cpp -o factorialC++
```

Implementación en Python

```
#!/usr/bin/env python

num = 6
factorial = 1

for i in range(1,num+1):
        factorial = factorial*i
print("El factorial de 6 es ",factorial)
```

Para la ejecución de los benchmarks se utilizó el comando time de GNU sobre el siguiente sistema:

Sistema Operativo: Ubuntu 16.04.5 LTS 64-bit
Procesador: Intel® Core™ i5-8250U CPU @ 1.60GHz × 8

Lenguaje	Tiempo Usuario	Tiempo Sistema	Tiempo Real	Uso de CPU
Citronella	0,00s	0,00s	0,002	81%
C++	0,00s	0,00s	0,003	85%
Python	0,01s	0,00s	0,014	96%

Referencias

- *“lex & yacc”* de *John R Levine, Tony Mason & Doug Brown*
- *https://github.com/petewarden/c_hashmap*