Teoría de Lenguajes y Autómatas

Proyecto Especial
Parte II: Backend

23/11/2023

Docentes

Mario Agustín Golmar
Ana Maria Arias Roig

Miembros del grupo

Jose Burgos - jburgos@itba.edu.ar - 61525
Santiago Rivas -srivas@itba.edu.ar - 61007
Franco Panighini - fpanighini@itba.edu.ar - 61258
Thomás Busso - tbussozungri@itba.edu.ar - 62519

Tabla de contenidos

Repositorio

https://github.com/Santiago-Rivas/TLA-TPE.git

Idea subyacente y objetivos del lenguaje

La idea es desarrollar un lenguaje no tipado que permita crear y visualizar circuitos eléctricos definiendo sus componentes básicos como baterías/pilas, cables, resistencias, capacitores, inductores, luces, etc. El lenguaje deberá generar un archivo de Látex con el circuito definido previamente.

El objetivo de este lenguaje es que lo usen tanto estudiantes como profesores para modelar diferentes circuitos.

Consideraciones realizadas

Debido a las características de nuestro lenguaje, no es necesario la implementación de ciclos “for” , ”while” o “if” , ya que no le vimos demasiada utilidad a la hora de crear mallas.

Los componentes se conectan en serie a menos que se indique lo contrario mediante la palabra “parallel”, en donde se indica entre paréntesis y separados con corchetes los dos bloques de componentes en serie, que se conectan en paralelo.

Por ejemplo en el siguiente código:

```
start mesh
    parallel (
        [
            Battery
            Led
            Capacitor
        ],
        [
            Resistor
        ]
    )
    Battery
end mesh
```

Aquí hay 2 bloques en serie:

- El 1° tiene una batería, un led y un capacitor.
- El 2° con un resistor.

Luego estos 2 bloques se interconectan en paralelo entre sí.

Finalmente estos 2 bloques en paralelo se conectan en serie con la batería

En cuanto al alcance de las variables, en nuestro lenguaje se definen todas las variables al principio, puesto que se trata de representar un circuito, por lo que carece de sentido definir nuevas variables dentro de un bloque “parallel”. Como consecuencia, todas las variables tienen el mismo alcance.

Si se ingresan números extremadamente grandes para los valores de los componentes, por ejemplo un valor de resistencia de $4*10^{20}$ ohm, se reemplaza con el valor “-1”.

Desarrollo

Para el desarrollo del lenguaje se hizo uso de Flex y Bison, herramientas vistas en clase y provistas por la cátedra.

Lo primero que hicimos fue definir las palabras propias del lenguaje en el archivo “flex-patterns.l”. Luego definimos el alfabeto y los símbolos no terminales en el

archivo “bison-grammar.y”. En este archivo también definimos la gramática de nuestro lenguaje.

Para almacenar las variables, decidimos usar un hashmap donde la clave es el nombre de la variable y el valor es una struct llamada “Comp” que definimos, donde se almacena el componente asociado a esa variable, junto con todos sus metadatos como son el tipo de componente, el color del mismo y los parámetros. Los parámetros hacen referencia al valor nominal del componente junto a su unidad de medida.

```
typedef struct {
    char * identifier;
    Component * component;
} Comp;
```

Dificultades encontradas

Al momento de representar los componentes, tuvimos problemas al modelar particularmente el transistor, por lo que decidimos eliminarlo de la lista de componentes ya que no logramos que se representara de manera correcta.

Otra dificultad encontrada fue la traducción espacial de nuestro lenguaje al lenguaje de la librería de LaTex. En particular, el modo de manejar el anidamiento de circuitos en paralelo en LaTex fue complicado. La librería para graficar los circuitos utiliza una grilla con coordenadas (x,y) para generar cada componente en el lugar deseado y para los cables que unen a los componentes, lo cual significó que tuvimos que construir el circuito manualmente siguiendo el “árbol” generado por nuestro lenguaje.

Benchmarking

Tiempo de compilación en WSL: 0.879 seg
Tiempo de compilación en Linux: 0.679 seg
Tiempo de compilación en Windows: 0.511 seg

Tiempo de correr todos los tests en WSL:0.084 seg
Tiempo de correr todos los tests en Linux: 0.043 seg
Tiempo de correr todos los tests en Windows: 0.077 seg

Futuras extensiones

Nos interesó, desde el comienzo, la posibilidad de que se puedan hacer cálculos a partir de los valores dados. Por ejemplo, calcular V utilizando R.I de nuestro circuito.

Otra extensión que nos pareció interesante, era la posibilidad de que el compilador detecte si un circuito con valores cumple con las leyes de la física. Es decir, que además de tener una malla válida, también tenga valores coherentes para sus componentes.

Referencias y bibliografía

Utilizamos la librería de LaTex:

PDFLaTex (se puede utilizar cualquier compilador de LaTex):

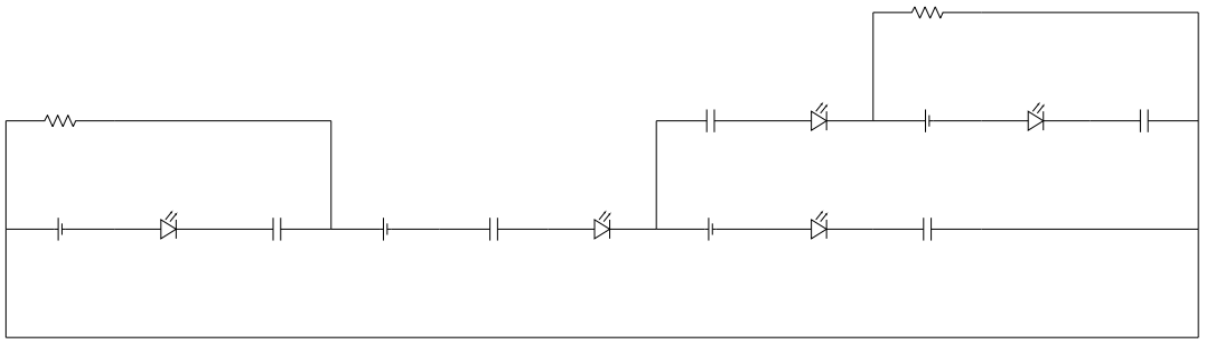
Anexo

En esta sección se adjuntan imágenes de las salidas esperadas de los archivos de prueba.

Test: 03-parallel_series_extra_nested
Código:

```
start mesh
    parallel (
        [
            Battery
            Led
            Capacitor
        ],
        [
            Resistor
        ]
    )
    Battery
    Capacitor
    Led
    parallel (
        [
            Battery
            Led
            Capacitor
        ],
        [
            Capacitor
            Led
            parallel (
                [
                    Battery
                    Led
                    Capacitor
                ],
                [
                    Resistor
                ]
            )
        ]
    )
end mesh
```

Salida:

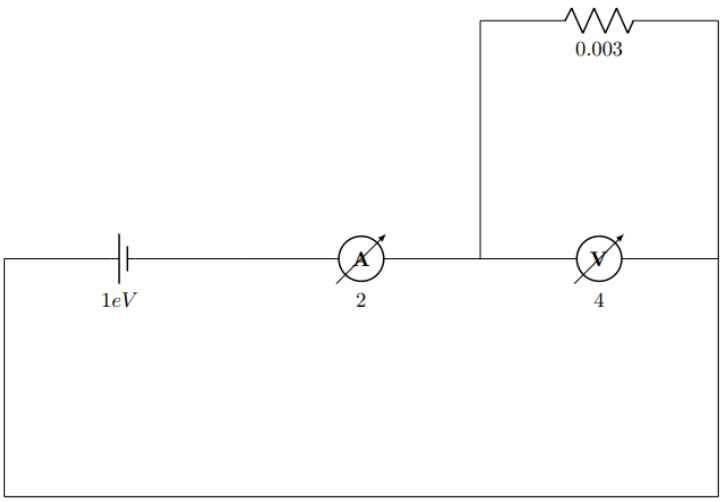


Test: 16-variables

Código:

```
my_bat = Battery({1,"eV"})
my_vm = Voltmeter(4)
my_am = Ammeter(2)
my_res = Resistor(0.003)
start mesh
    my_bat
    my_am
    parallel(
    [
        my_vm
    ],
    [
        my_res
    ]
    )
end mesh
```

Salida:

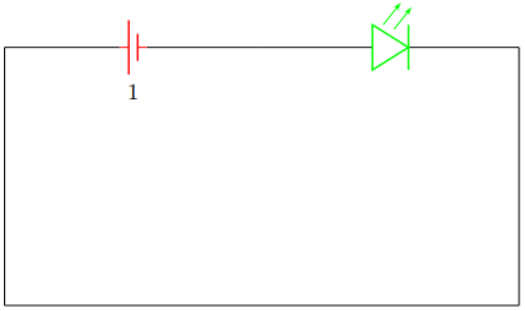


Test: 17-color_varialbes

Código:

```
bat = red Battery(1)
led = Led
start mesh
    bat
    green led
end mesh
```

Salida:



Test:15-multicomponent_TP1_example2

Código:

```
my_light = Led
battery1 = Battery({2, "V"})

start mesh
   battery1
   Ammeter({2, "mA"})
   my_light
   red battery1
end mesh
```

Salida: