Autómatas, Teoría de Lenguajes y Compiladores 72.39

Trabajo Práctico Especial 2022 2C

Grupo 6
Arnott, Ian - Leg. 61267
De Simone, Franco - Leg. 61100
Mattiussi, Agustín Hernán - Leg. 61361
Sasso, Julián Martín - Leg. 61535

Tabla de contenidos

Idea

Desarrollar un lenguaje que permita construir y manipular autómatas finitos determinísticos y evaluar la aceptación de palabras en los lenguajes de los mismos. Este lenguaje contará con dos secciones: una de declaración y una de ejecución.

En la primera sección, se podrán definir uno o más autómatas, estados de los mismos y qué símbolos se pueden consumir para transicionar entre estados. Cabe destacar que los símbolos no estarán limitados a caracteres individuales, sino que pueden ser composiciones de estos. Cualquier transición no definida para un símbolo, caerá en un estado trampa.

En cuanto a la segunda sección, se podrá entregar a los autómatas definidos una palabra o una lista de estas para su verificación, o editarlos para probar nuevas posibilidades. Esto es, añadir o quitar transiciones, unir autómatas, añadir o quitar estados, etc.

La salida de un programa en este lenguaje será una lista de pares “palabra-booleano” donde se indique para cada palabra brindada si cierto autómata la acepta o no.

Sintaxis

- Comentarios: Se encierran entre “/*” y “*/”. Ejemplo:

```
/* Este es un Comentario */
```

- Los “literales” (no variables) se escriben entre comillas. Ejemplo:

```
abc  /* variable llamada abc */
"abc" /* string "abc" */

sym[] array = {"a", "b", abc, "abc"};
```

- Toda declaración o sentencia debe terminar en “;”.

Tipos de Dato

Símbolo

```
sym nombre = "valor";
```

Representa los símbolos del autómata. El valor no tiene por qué ser un caracter único, puede ser un conjunto.

Ejemplo

```
sym ejemplo = "Avbbc43df";
```

Estado

```
sta nombre = "valor";
```

Representa los estados del autómata. El valor no tiene por qué ser un caracter único, puede ser un conjunto.

Ejemplo

```
sta ejemplo = "q0";
```

Transición

```
trn nombre = {ESTADO_ORIGEN, ESTADO_DESTINO, simboloConsumido};
```

Representa una transición entre el estado "*ESTADO_ORIGEN*" y "*ESTADO_DESTINO*" consumiendo el símbolo "*símbolo consumido".*

Ejemplo

```
trn ejemplo1 = {"q0", "E2", "a"};

sta EST1 = "Estadote";
sym unSimbolo = "abcdef";
trn ejemplo2 = {EST1, "Estadito", unSimbolo};
```

Array

```
sym[] arraySym = {"valor1", "valor2", ..., "valorN"};

sta[] arraySta = {"valor1", "valor2", ..., "valorN"};
```

```
trn[] arrayTrn = {{"eo1", "ed1", "a1"}, ... , {"eoN", "edN", "aN"}};
```

Los arrays representan vectores de algún tipo de dato con sus componentes separadas por coma y encerradas entre llaves.

En nuestro lenguaje sólo existen los tipos simples “símbolo” y “estado”, por lo que los arrays sólo deberían representar vectores de estos tipos. Cabe destacar, por separado (no puede haber un array con tipos mezclados).

Ejemplo

```
sym[] ejemplo1 = {"a", "bb", "c"};
trn[] ejemplo2 = {{"E1", "E2", "a"},
                  {"E2", "E1", "a"},
                  {"E1", "E1", "b"}};
```

Adicionalmente, una instancia de alguno de estos tipos creada previamente, puede añadirse al arreglo.

Ejemplo:

```
sym miSimbolo = "abc123";
sym[] ejemplo2= {"x", "y", "15", miSimbolo, "bbb"};

sta ESTADO = "E3";
sta[] ejemplo3 = {"E1", "E2", ESTADO};

sta EST1 = "Estadote";
sym unSimbolo = "abcdef";
trn unaTransicion = {EST1, "Estadito", unSimbolo};
trn[] ejemplo4 = {unaTransicion, {EST1, EST1, "p"}};
```

Autómata

```
dfa myAuto = {
        arrayEstados,
        arraySímbolos,
        estadoInicial,
        estadosFinales,
        funcionDeTransición
     };
```

Representa un autómata. Igual que un autómata real, consiste de 5 componentes:

- (sta[]) **Conjunto de Estados**: Lista de estados de la gramática (nodos del autómata). Es un arreglo de estados SIN REPETIDOS.
- (sym[]) **Conjunto de Símbolos:** Lista de símbolos terminales del lenguaje (alfabeto). Es un arreglo de símbolos SIN REPETIDOS.
- (sta) **Estado Inicial:** Punto de entrada del autómata.
- (sta[]) **Array de Estados Finales**: Define un arreglo de estados en los que el autómata podría aceptar palabras. Deben ser un subconjunto del Array de Estados. Por ende, tampoco puede hbaer estados repetidos.
- (trn[]) **Función de Transición**: Representa las transiciones entre estados del autómata, junto con el símbolo consumido en cada caso. Consiste en una lista de transiciones SIN REPETIDOS.

  ```
  {ESTADO_ORIGEN,  ESTADO_DESTINO,  simboloConsumido}
  ```

  Obviamente, los estados deben pertenecer al array de estados declarado para el autómata y lo análogo para los símbolos.

Ejemplo

```
/* Declaramos las variables previamente */
sta initial = "q0";
sta final = "qf";
sta[] states = {inicial, final};
sta[]  finalStates = {final};
sym nums = "123456789";
sym[] symbols = {"a", "bb", nums};
trn[] transitions = {
          {inicial, final, "a"},
          {final, inicial, nums},
          {final, final, "bb"}
};

/* Declaramos el automata */
dfa myAuto = {
        states,
        symbols,
        initial,
        finalStates,
        transitions
 };
```

Aclaración: El orden de los parámetros del autómata es estricto.

Aclaración: Los parámetros del autómata deben ser variables declaradas previamente.

Aclaración: Los valores no definidos de la función de transición, serán llevados a un estado trampa artificial.

Aclaración: Todas las variables son finales, excepto los autómatas

Prestaciones

Entregar Palabras a un Autómata

```
automata CHECK arrayDeSimbolos;
```

Se le entrega la palabra “palabra” al autómata para que éste compruebe si la acepta o no en su lenguaje. En esta implementación, una palabra es un arreglo de símbolos. A diferencia del arreglo de símbolos entregado al autómata en su definición, este acepta repetidos.

Ejemplo

```
dfa myAuto = {...};
sym[] palabra = {"h", "o", "l", "a", "mundo"};  /* Notemos que "mundo" es un solo simbolo */
myAuto CHECK palabra;
```

¿Por qué no entregar al autómata directamente strings y complicarnos con sym[]? Porque la primera opción puede generar ambigüedad en cuando a la
transición a aplicar. Supongamos que hubiésemos definido los siguientes símbolos y transiciones:

trn[] *transiciones* = {{*E1*, *E2*, “a”}, {*E2*, *E3*, “b”}, {*E1*, *E4*, “ab”}, …};
dfa *auto* = {…, *E1*, …, *transiciones*}; /* E1: Estado inicial */

Al realizar la operación ‘*auto* CHECK “abc”’, ¿Con qué transición comenzamos? ¿Con la que consume “a” para moverse a *E2* y luego aplicar la que consume “b” para ir a *E3*? ¿O con la que consume “ab” para moverse directamente a *E4*?

Si bien resulta en un lenguaje más “verboso”, esta ambigüedad se soluciona entregando al autómata un sym[]:
OPCION 1) entrego {”a”, “b”, “c”}
OPCION 2) entrego {”ab”, “c”}

Complementar un Autómata

```
dfa automataComplementado = !automataInicial;
```

Esta operación devuelve un autómata complemento del operando. Es decir, con estado iniciales y finales intercambiados.

Ejemplo

```
dfa notMyAuto = !myAuto;
myAuto = !myAuto;
```

Agregar un Estado a un Autómata

```
dfa automata2 = ADD estado TO automata1;
```

Se añade un nuevo estado al autómata. Inicialmente, toda transición de este nuevo estado se lleva al estado trampa. Luego se puede cambiar.

Ejemplo

```
myAuto2 = ADD "nuevoEstado" TO myAuto1;
sta unEstado = "estadito";
myAuto1 = ADD unEstado TO myAuto1;
```

Agregar un Transición a un Autómata

```
dfa automata2 = ADD transicion TO automata1;
```

Se añade una nueva transición entre dos estados de un autómata. Obviamente, la transición a añadir debe usar estados y símbolos del autómata.

Esta operación falla si el autómata resultante fuera no determinístico. Es decir, si se intenta agregar una transición de un estado a otro con cierto símbolo ya existiendo una transición desde el estado origen que consume ese símbolo. Esto no se tiene en cuenta cuando el estado que “colisiona” es el estado trampa artificial agregado para el autómata (simplemente se “pisa” esa transición).

Ejemplo

```
myAuto2 = ADD {E1, E2, "a"} TO myAuto1;
myAuto1 = ADD {E1, E2, "b"} TO myAuto1;
```

Remover un estado de un autómata

```
dfa automata2 = REM estado FROM automata;
```

Se elimina el estado “estado” del automata. No se puede eliminar el estado inicial. Si se elimina un estado que deja al autómata como grafo no conexo, sigue siendo un único autómata, pero con estados inaccesibles. Si se eliminan todos los estados finales, el autómata ya no aceptará ninguna palabra.

Ejemplo

```
dfa automata2 = REM q0 FROM myAuto1;
```

Unir Dos Autómatas

```
dfa newAuto = auto1 JOIN auto2 {"estado_auto1", "estado_auto2", "símbolo"};
```

Se unen dos autómatas, creando uno nuevo. Notemos que se añade una transición particular, que una un estado del primer autómata con un estado del segundo. El nuevo autómata tendrá:

- **Estados**: Unión de los conjuntos de estados de ambos autómatas. (si existen nombres repetidos, se produce un error).
- **Símbolos**: Unión de los conjuntos de símbolos de ambos autómatas.
- **Estado Inicial**: El estado inicial del primer autómata.
- **Estados Finales**: Unión de los conjuntos de estados finales de ambos autómatas.
- **Función de Transición**: Se respeta la de cada autómata y cualquier símbolo que no tenga transición desde algún estado, se lleva a un estado trampa. Se agrega la nueva transición, que idealmente uniría los estados deseados del primer y segundo autómata, pero no es obligatorio (resultaría en un autómata con dos secciones no conexas).

  Ejemplo

```
dfa newAuto = auto1 JOIN auto2 {E1, G3, "a"};
```

Impresión en Pantalla

```
PRINT "Texto a Imprimir";
```

Permite imprimir en salida estándar el texto pasado como parámetro a la función. A la cadena a imprimir, se le concatena “#“ al principio. Esto es por si algún programa que utilice la salida de un programa DFA-Factory como entrada, quisiera ignorar las impresiones por pantalla.

Salida

La salida de un programa escrito en DFA-Factory, es una lista de ternas “autómata-palabra:booleano” que indica para cada operación CHECK si el autómata correspondiente aceptó o no la palabra.

Adicionalmente, con la función PRINT se puede imprimir cualquier cadena por salida estándar.

Ejemplo:

```
auto1-{"h", "o", "l", "a"}:true
auto2-{"h", "o", "l", "a"}:false
auto3-{"aa", "a"}:true
# Acá complementé auto3 !!!
auto3-{"aa", "a"}:false
```

Dificultades a la hora de desarrollar el proyecto

El desarrollo del proyecto no fue una tarea sencilla para el equipo. La primera parte de este, es decir el planteo de la idea y los casos de uso no presentó mayores dificultades. Ahora bien, con el comienzo del desarrollo de la parte del *frontend* del proyecto (el analizador léxico de *Flex* y el analizador sintáctico de *Bison*) comenzaron a surgir las primeras dificultades, varias cosas que pensamos en la primera entrega fueron modificadas con el objetivo de un mejor y más eficiente desarrollo. El desarrollo de la gramática en *Bison* tampoco fue simple ya que si bien lo pensamos de una manera inicialmente, a la hora de compilarlo y probar los *tests* eran constantes los conflictos *SHIFT-REDUCE* los cuales debían ser solucionados para un efectivo de una gramática *LALR(1).*

Una vez completada esta entrega con éxito, fue hora de comenzar a desarrollar la parte de *backend* donde el equipo se topó con dificultades como agregar los tokens a la tabla de símbolos, generar las estructuras para almacenar cada tipo de dato y luego pensar sobre como recorrer el árbol de sintaxis abstracta construido desde el analizador sintáctico y así generar el código encargado de producir la salida de los programas desarrollados con este lenguaje.

Futuras extensiones y/o modificaciones

Con vistas a futuro, este compilador podría modificarse para manejar diferentes tipos de autómatas. Actualmente solo soporta operaciones y declaraciones de autómatas finitos determinísticos, pero en el futuro esta implementación podría extenderse para manejar autómatas finitos no determinísticos e incluso hasta autómatas de pila.

Por otra parte, como extensión también se podría considerar usar otro tipo de estructura de datos, como por ejemplo un hasmap, con el objetivo de optimizar la complejidad de las operaciones.

Conclusión

En resumen, si bien como se mencionó anteriormente este trabajo no fue sencillo, a la vez este presentó un desafio muy interesante para el equipo, Al dividirlo en 3 entregas, algunas cosas que pensamos con inicialmente luego tuvieron que ser modificadas para desarrollar la parte de *frontend*, mismo caso con la transición de *front* a *backend*. Como se fueron presentando las diferentes dificultades mencionadas el equipo tuvo que utilizar toda su capacidad para aplicar los conocimientos aprendidos en la materia e intentar resolverlas y así conseguir realizar la versión final este proyecto.