Autómatas, Teoría de Lenguajes y Compiladores
Trabajo Práctico Especial- BPMN compiler
Entrega Final

Profesores: Ana María Arias Roig

Agustín Golmar

Agustín Benvenuto

Alumnas:

Baron, María Mercedes	61187
Occhipinti, Abril	61159
Ortu, Agustina Sol	61548
Rossi, Victoria	61131

Año: 2022

Índice

Repositorio:

- https://github.com/mechibaron/TP-BPMN-Compiler (branch master)

Introducción:

El objetivo principal del proyecto es desarrollar un lenguaje que permita crear diagramas de estructura BPMN a partir de comandos user-friendly para el usuario.

BPMN, Business Process Model and Notation, es una notación gráfica para el modelado de procesos.

La idea del lenguaje modelado es que acepte una cantidad finita y predefinida de comandos para luego transformar cada uno en un símbolo aceptado por la notación BPMN. Una vez finalizada la inserción de componentes, generará el diagrama a través de dichos símbolos. De esta manera, cualquier usuario (con conocimientos en programación o no) puede utilizar el proyecto de manera sencilla y rápida logrando el objetivo que desee dentro de los límites del lenguaje.

Consideraciones:

A la hora de querer correr el programa, el usuario debe tener en cuenta que necesita tener previamente instalado **dot.** Para descargar dot visite https://graphviz.org/download/ y siga las instrucciones.

Esto se debe a que el proyecto genera un archivo out.dot. Luego, se deberá ejecutar en una consola la siguiente línea:

```
dot -Tsvg out.dot  -o out.svg
```

Dicha línea traducirá el archivo .dot a un archivo svg el cual posteriormente se puede abrir y visualizar el grafo BPMN realizado por el compilador. Puede visualizar la representación de un grafo ejemplo aquí.

Además, se tuvo en cuenta varias consideraciones en el lenguaje. Estas limitaciones fueron llevadas a cabo dado que la notación BPMN tiene extensas posibilidades de aplicación. Por lo tanto, no posee la variedad de tipos de actividades, artefactos y compuertas.

Asimismo, no está permitido que se creen “usuarios” y se asignen a diferentes actividades, eventos, etc.

Adicionalmente, el lenguaje permite que los artefactos se vinculen tanto entre pools como dentro de las mismas y entre otro tipo de símbolos.

Descripción:

El lenguaje modelado consta de varias herramientas para crearlo.

Por un lado, están los eventos, los cuales son estados que nos indican que en el proceso de negocio algo importante ha ocurrido. De éstos hay tres tipos: los de inicio, que desencadenan el flujo de secuencia de un proceso. Los eventos intermedios, que interrumpen temporalmente el flujo de secuencia y los eventos finales quienes terminan con dicho flujo.

Los eventos que han ocurrido son condiciones generadas por una causa externa, éstos pueden ser eventos de inicio o intermedios. Los eventos desencadenados son condiciones generadas por el propio proceso y pueden ser eventos intermedios o finales.

Por otro lado, también permite ingresar actividades, las cuales se utilizan para representar los pasos individuales del proceso en los diagramas de BPMN.

Asimismo, se pueden crear compuertas, éstas son utilizadas para los procesos no lineales, es decir, para los que consisten en divisiones y fusiones.

Además, BPMN también consta de conectores, los cuales conectarán los flujos de proceso.

Adicionalmente, permite diferenciar los diferentes participantes del proceso. Estos se encuentran divididos en piscinas y carriles.

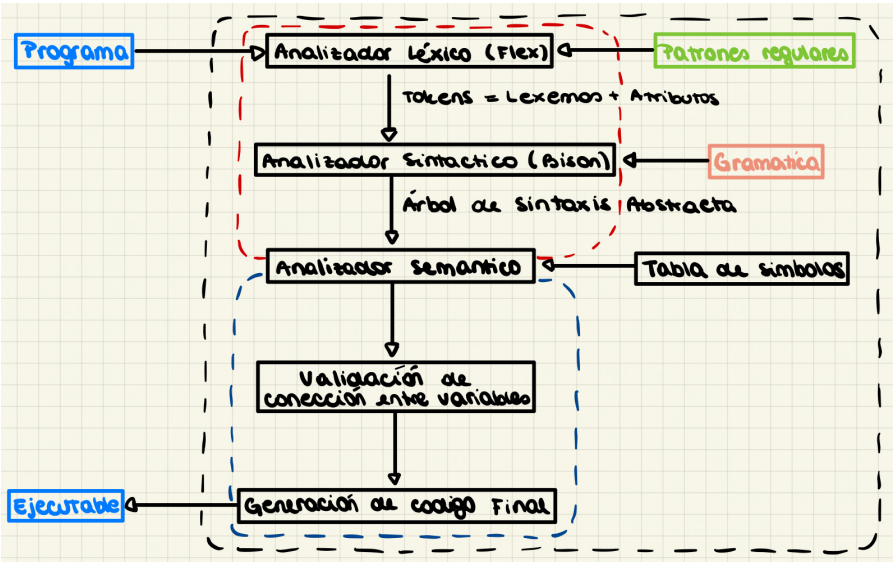
Por último, se encuentran los artefactos para permitir un diseño del proceso aún más claro.

La gramática creada para este lenguaje utiliza todos estos objetos (variables de tipo event, activity, gateway, connection, artifact, pool o lane) conectándolos entre sí mediante asociación, flujo de secuencia o flujos de mensaje.

Desarrollo del proyecto:

Frontend:

Para la primera entrega se definió la arquitectura básica del compilador la cual se modificó posteriormente en la segunda entrega dado que la validación de conexión entre variables se tuvo que realizar antes de la generación de código. Esto fue así pues nuestro lenguaje acepta que primero se defina la conexión y luego se definan las variables. La arquitectura junto con sus componentes quedaron dispuestas de la siguiente manera:



Adicionalmente, se definieron patrones y expresiones regulares utilizados por el analizador léxico con Flex. Dentro de los patrones reutilizables, se definió el fin de línea, el espacio en blanco, los tipos que se pueden tener tanto de eventos como de artefactos y por último, las variables. Por otro lado, se determinaron expresiones fijas que permite y requiere el lenguaje, como por ejemplo, las palabras start, end, graph, pool, lane, etc.

Además se desarrolló la gramática del proyecto y se trasladó a Bison. Fue definida de la siguiente manera:

G = < S_{NT}, S_T, P, S_I >

donde

$S_{NT} = \{program, graph, pool, lane, create, createp, expression, gateway, set, connect\}$

S_T = {*START, END, GRAPH_ID, LANE, CREATE, EVENT, ACTIVITY, GATEWAY, CONNECT, CONNECT_TO, CURLY_BRACES_OPEN, CURLY_BRACES_CLOSE, ARTIFACT, AS, TO, SET*}

$S_I = \{program\}$

P = {
program → graph
graph → *START GRAPH_ID NAME* pool *END GRAPH_ID* |
START GRAPH_ID NAME create *END GRAPH_ID*
pool → *START POOL NAME* lane createp *END POOL* |
START POOL NAME lane createp *END POOL* pool
lane → *START LANE NAME* create *END LANE* lane |
START LANE create *END LANE* lane |
START LANE lane END LANE lane |
START LANE NAME lane END LANE lane |
λ
create → expression | expression create
createp → create | ***λ***
expression → *CREATE EVENT EVENT_TYPE NAME AS VAR* |
CREATE ACTIVITY NAME AS VAR |
CREATE ARTIFACT ARTIFACT_TYPE NAME AS VAR |
gateway |
connect
gateway → *CREATE WATEWAY NAME CURLLY_BRACES_OPEN* set
CURLY_BRACES_CLOSE AS VAR
set → *SET NAME CONNECT TO VAR* |
SET NAME CONNECT TO VAR set
connect → *CONNECT VAR TO VAR*
}

Luego, para conectarlo con el backend las funciones para cada producción fueron llevadas a cabo.

Backend:

Para esta entrega, se definieron las estructuras necesarias para construir el árbol de sintaxis abstracta, que luego será recorrido por la función Generator() definida en generator.c

También se construyó una tabla de símbolos, donde se guarda el nombre y tipo de las variables para validar que no existan dos variables con el mismo nombre y que las éstas estén efectivamente creadas. Para la misma, se implementó una lista encadenada con un nodo el cual contiene los datos que se guardarán y un next. Esta implementación simplificó la validación dado que se lograba tan solo con recorrer los nodos de la lista.

Luego de construir el AST, pero antes de generar el código, se realiza una etapa de validación donde se chequea que las variables (eventos/actividades/artefactos/gateways) a conectar estén creadas y que los eventos finales no generen conexiones nuevas. Este proceso se puede ver en validateProgram.c. Realizar esta validación en una etapa posterior a generar el AST permite al usuario escribir conexiones antes de crear las variables. Si alguna de las variables a conectar no está creada, entonces se retorna -1 y no se pasa a la etapa de generación de código.

Al momento de generar el código, se recorre el AST para luego crear un archivo out.dot el cual contiene las instrucciones necesarias, en lenguaje DOT, para construir un archivo.svg. Este archivo finalmente tiene la representación del BPMN.

Dificultades encontradas:

En un comienzo del proyecto hubieron dificultades a la hora de seleccionar de manera inteligente el lenguaje de programación a elegir.

Al tener incorporado lenguajes de programación más complejos, no se lograba abrir la idea de pensamiento de manera tal para llegar a uno de nuestros objetivos principales del proyecto, que cualquier tipo de usuario pueda utilizarlo (con conocimientos previos de programación o no).

Por otro lado, a la hora de realizar la gramática hubieron varios impedimentos para armarla correctamente. Esto fue debido a la cantidad de casos y los tipos de las variables. Algunas medidas fueron tomadas para poder realizarla. Éstas fueron limitar un poco más el lenguaje de lo que estaba previsto (por ejemplo, la creación de subprocesos y tareas dentro de una actividad), pero manteniendo siempre el objetivo del lenguaje. De esta manera, se logró una definición correcta sin recursividad a izquierda, ni con conflictos de reduce/reduce o shift/reduce.

Además, al momento de crear el árbol, él mismo se estaba creando de forma incorrecta dado que no se estaban agregando nodos correctamente. Esto sucedía dado que el árbol se crea de abajo para arriba mientras que en un principio se estaba cortando con null una ramificación del árbol. Este error surgía dado que en bison, cuando en la gramática había un no terminal a veces se devolvía null. Consecuentemente, se solucionó devolviendo una función que crea la estructura correcta y la retorna.

Futuras extensiones y/o modificaciones:

Como se mencionó en Consideraciones, la implementación del lenguaje tuvo que ser reducida dado la amplia variedad de objetos y variantes que se pueden realizar y el tiempo que tuvimos para realizar el proyecto. Por lo tanto, en un futuro, el lenguaje se puede modificar para que acepte:

- Variedades de símbolos de eventos como “Símbolo de mensaje”, “Símbolo de temporizador”, “Símbolo de error”, etc.
- Tipos a las actividades y sus características, como que una tarea sea representada con un rectángulo de línea contigua, un subproceso sea representado con un rectángulo con línea punteada, etc.
- Tipos de compuertas, como que sean exclusivas, paralelas, inclusivas, etc.
- Distintas representaciones de los artefactos. Es decir, que cada uno tenga un símbolo identificador.
- Limitar a que los artefactos de tipo Mensaje solamente se puedan conectar entre distintas piscinas.

Ejemplo:

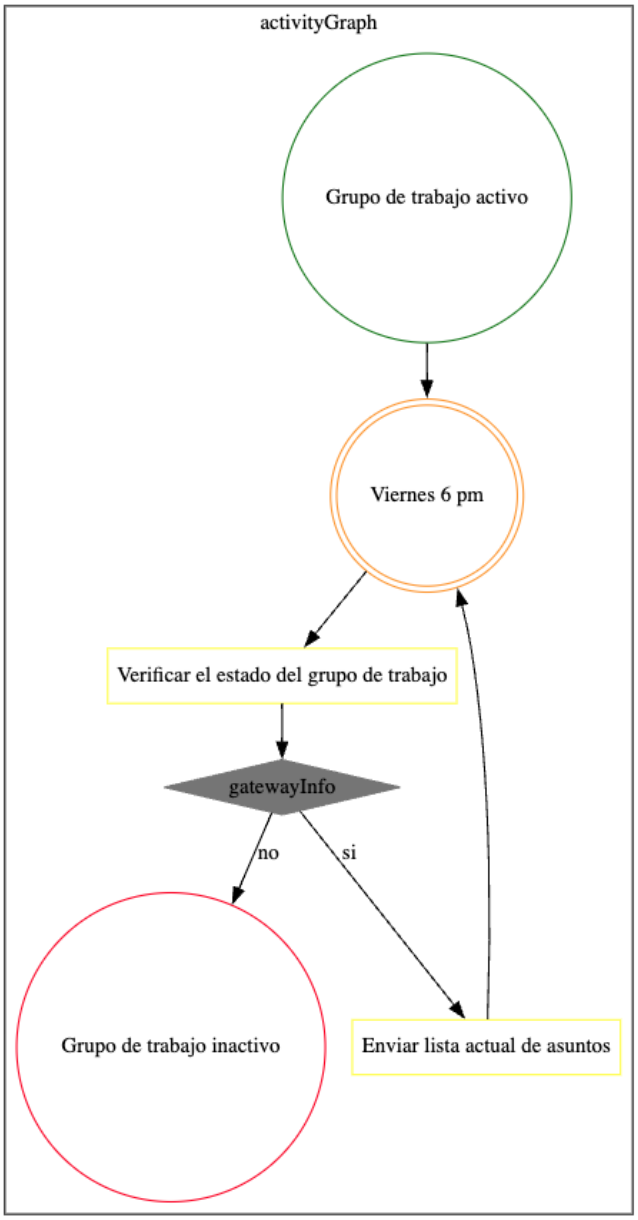
Código del usuario:

```
start graph "activityGraph"
    create event initial "Grupo de trabajo activo" as $first
    create event intermediate "Viernes 6 pm" as $clock
    create event final "Grupo de trabajo inactivo" as $last
    create activity "Verificar el estado del grupo de trabajo" as $activity1
    create activity "Enviar lista actual de asuntos" as $activity2
    create gateway "gatewayInfo" {
        set "si" connect to $activity2
        set "no" connect to $last
    } as $gate
    connect $first to $clock
    connect $clock to $activity1
    connect $activity1 to $gate
    connect $activity2 to $clock
end graph
```

Código generado por el compilador:

```
digraph {
subgraph cluster_0 {
label="activityGraph"
labelloc="t"
    first [label="Grupo de trabajo activo" , shape=circle, color=green]
    clock [label="Viernes 6 pm" , shape=doublecircle, color=orange]
    last [label="Grupo de trabajo inactivo" , shape=circle, color=red]
    activity1 [label="Verificar el estado del grupo de trabajo" , shape=rectangle, color=yellow]
    activity2 [label="Enviar lista actual de asuntos" , shape=rectangle, color=yellow]
    gate [label="gatewayInfo" shape=diamond,  color=gray, style=filled]
gate->activity2 [label="si"]
gate->last [label="no"]
    first->clock
    clock->activity1
    activity1->gate
    activity2->clock
}
}
```

Grafo generado por dot:



Bibliografía:

https://graphviz.org/
https://www.gbtec.com/es/recursos/bpmn/
https://www.lucidchart.com/pages/es/bpmn-bpmn-20-tutorial