Trabajo Práctico Especial

Automatas, Teoría de Lenguajes y Compiladores

Integrantes:

- Bilevich, Andres Leonardo 59108
- Margossian, Gabriel Viken 59130
- Mónaco, Matías Damian 59102
- Lin, Scott 59339

Índice

Idea subyacente y objetivo del lenguaje.

Nuestro lenguaje está diseñado para poder generar simulaciones físicas entre cuerpos. Estos cuerpos pueden interactuar entre ellos, utilizando características como la densidad, tamaño y la constante gravitacional de un espacio. Con estos parámetros se utilizan ecuaciones físicas para calcular la atracción y las fuerzas producidas entre dichos cuerpos y visualizar esto en la pantalla.

Con este lenguaje se podrían simular sistemas solares estables o inestables y colisiones inelásticas entre cuerpos a distintas velocidades. También se puede observar cómo se atraen entre sí y cómo los distintos parámetros de cada cuerpo afectan el comportamiento del sistema. Todos los objetos que se encuentran en el espacio afectan a los otros en mayor o menor medida, dependiendo de variables como la distancia en la que se encuentran, la densidad de estos objetos, etc. Cabe destacar que la atracción entre cuerpos depende de la constante gravitacional que puede ser modificada (si esta se pone en 0 los cuerpos dejan de atraerse entre sí pero aún colisionan) y también que las distancias son medidas en pixeles.

El lenguaje permite también que se modifiquen las configuraciones del espacio donde se encuentran los cuerpos para poder generar distintos entornos personalizados para las distintas simulaciones que se quieran realizar.

Consideraciones realizadas (no previstas en el enunciado).

Para poder crear este simulador se eligió el motor gráfico de HTML Canvas, que nos permite dibujar formas simples y animarlas para generar la ilusión del movimiento de los distintos cuerpos. Este motor corre a base de js en una pestaña del buscador, es por eso que la salida de nuestro compilador es un archivo “index.html” dónde embebemos el código JS. junto con una template de página estándar y el código necesario para correr el simulador.

Descripción del desarrollo del TP.

Empezamos analizando las posibles ideas de lenguajes que podíamos desarrollar, después de discutir distintas ideas, nos decantamos por la actual. Comenzamos desarrollando el código JS para tener la base de lo que necesitamos para hacer la simulación. Con esto hecho, se empezó desarrollando el lexer y el parser para que puedan traducir simples símbolos como números, cadenas, bloques condicionales (if), bloques while. Luego trabajamos en definir el lenguaje que queríamos crear, se decidió hacerlo tipado para simplificar las verificaciones de tipos, por ejemplo, en operaciones aritméticas. Una vez que tuvimos todo definido, empezamos a modelar una gramática que permita generar dicho lenguaje, esto nos ayudó mucho a la hora de escribir el parser del lenguaje. Con todo lo anterior hecho, se adaptó el parser básico que teníamos para que pudiera traducir el código de nuestro lenguaje a código JS. Después de tener varios problemas durante el desarrollo del parser y una consulta con el profesor, se decidió reestructurar la gramática del lenguaje para facilitar el desarrollo,se hicieron cambios como definir una zona de declaración de variables para evitar problemas con el orden de las reducciones durante el parseo. Con estos cambios hechos, se trabajó en agregar funcionalidad al lenguaje.

Descripción de la gramática.

A continuación se muestra la definición de nuestra gramática:

```
<start> ::=
        lambda
      | <variable_definitions> MAIN '(' ')' '{' <statement_list> '}'

<variable_definitions> ::=
        lambda
      | <variable_definitions> <variable_definition>

<variable_definition> ::=
        TYPE_NUM NAME
      | TYPE_STR NAME
      | TYPE_BOOL NAME
      | TYPE_NUM NAME '[' ']'
      | TYPE_STR NAME '[' ']'
      | TYPE_BOOL NAME '[' ']'

<statement_list> ::=
        lambda
      | <statement_list> <statement>

<statement> ::=
      <block_statement>
      | <expression_statement>

<block_statement> ::=
    | <if_statement>
    | <while_statement>
    | <for_statement>

<expression_statement> ::=
      <system>
    | <config>
    | <print>
    | NAME '[' <exp> ']' OPEQ <exp>
    | NAME '[' <exp> ']'  '=' <exp>
    | NAME '=' <exp>
    | NAME OPEQ <exp>
    | NAME '=' <arr_init>
```

```
<if_statement> ::=
       IF '(' <exp> ')' '{' <statement_list> '}'
     | IF '(' <exp> ')' '{' <statement_list> '}' ELSE '{'
<statement_list> '}'

<while_statement> ::=
     WHILE '(' <exp> ')' '{' <statement_list> '}'

<for_statement> ::=
     FOR  '(' <expression_statement> ';' <exp> ';'
<expression_statement>')' '{' <statement_list> '}'

<exp> ::=
     <exp> '+' <exp>
   | <exp> '-' <exp>
   | <exp> '/' <exp>
   | <exp> '*' <exp>
   | <exp> '%' <exp>
   | '-' <exp>
   | '(' <exp> ')'
   | <exp> AND <exp>
   | <exp> OR <exp>
   | NOT <exp>
   | <exp> COMPARATION <exp>
   | NAME '[' <exp> ']'
   | NAME
   | NUM_FUNCT_1 '(' <exp> ')'
   | NUM_FUNCT_2 '(' <exp>  ',' <exp> ')'
   | INTEGER
   | FLOAT
   | QSTRING
   | TRUE_TK
   | FALSE_TK
   | READ_STR '(' <exp> ')'
   | READ_NUM '(' <exp> ')'
   | GET_HEIGHT '(' ')'
   | GET_WIDTH '(' ')'

<arr_init> ::= '[' <arr_item> ']'

<arr_item>::=
        <exp>
     |  <exp> ',' <arr_item>
```

```
<system> ::= SYSTEM_TOKEN '.' <system_action>

<system_action> ::=
  ADDBODY '(' <exp> ',' <exp> ',' <exp> ',' <exp> ',' <exp> ',' <exp>
',' <exp> ')'

<config> ::= CONFIG_TOKEN '.' <config_action>

<config_action> ::=
       GRAVITY_CONF '(' <exp> ')'
     | BOUNCE_CONF '(' <exp> ')'
     | TRAIL_CONF '(' <exp> ')'

<print> ::= PRINT '(' <exp> ')'
```

*Esto también se encuentra en el archivo "grammar.bnf" pero lo dejamos aca tambien por comodidad.

Documentación del lenguaje.

A continuación se encuentra un breve resumen de todo lo que se puede hacer en el lenguaje

Declaración de variables simples:

- Para declarar una variable numérica se debe escribir en una línea de la forma:
  `num varName`
- Para declarar una variable de tipo String se debe escribir en una línea de la forma:
  `str varName`
- Para declarar una variable de tipo booleano se debe escribir en una línea de la forma:
  `bool varName`
  Donde varName es el nombre de la variable definida.

Declaración de arrays:

Los arrays deben estar declarados de la forma

`type varName[]`

Donde type puede ser cualquier tipo de variable simple (num, str o bool) y varName el nombre de la variable definida

Todas las variables que se quieran utilizar en el programa deben estar declaradas al inicio del código. (antes del main)

Operaciones entre expresiones:

Las operaciones que soporta nuestro lenguaje son:

- **Suma (+)** entre valores numéricos, entre cadenas de caracteres y entre cadenas de caracteres y valores numéricos.
- **Diferencia (-)** entre valores numéricos
- **División (/)** entre valores numéricos
- **Producto (*)** entre valores numéricos
- **Módulo (%)** entre valores numéricos

cabe destacar que también se acepta las expresiones

- número += número
- número -= número
- número /= número
- número *= número
- string += string o número

Inicialización y modificación de variables:

Todas las variables que se deseen visualizar o modificar deberán estar previamente declaradas fuera del main (al inicio del código) y deberán ser inicializadas dentro del main. Para inicializar cualquier variable correctamente, deberá ingresarse un valor correspondiente a su tipo. Ejemplos de cada caso a continuación:

```
num numVar
str strVar
bool boolVar

num arrNumVar[]
str arrStrVar[]
bool arrBoolVar[]

main(){
      numVar = 3
      strVar = "Hola Mundo"
      boolVar = true
      arrNumVar = [1,2,3]
      arrStrVar = ["Nombre","Apellido"]
      arrBoolVar = [true,false,true]

      arrNumVar[0] = 10
      arrStrVar[2] = "Edad"
      arrBoolVar = [true,true,true,true]
}
```

Expresiones booleanas:

Las distintas expresiones soportadas son:

- True y false
- Operaciones lógicas:
    - Not (!)
    - And (&&)

    - Or (||)
- Comparación entre números y cadenas:
    - mayor (>)
    - mayor o igual (>=),
    - menor (<)
    - menor o igual (<=)
    - igualdad (==)
    - diferencia (!=)

Bloque condicional if/else

El bloque condicional if recibe una expresión booleana y se puede escribir con o sin un else. Por ejemplo:

```
bool boolVar
main(){
      boolVar = true
      if(boolVar){
            print("variable is true")
      }
      else{
            print("Variable is false")
      }

      if(!boolVar){
            print("variable is false")
      }
}
```

Bloque while:

El bloque while recibe una expresión booleana. Por ejemplo:

```
num x
main(){
      x = 3
      while(x >= 0){
            print("val is: " + x)
            x = x - 1;
      }
}
```

Bloque for:

El bloque for recibe una expresión numérica, una expresión booleana y los incrementos que se tienen que realizar en cada iteración. Por ejemplo:

```
num x
num cant
```

```
main(){
      cant = 10
      for(x = 0 ; x < cant ; x+=1){
            print("Iteration: " + x)
      }
}
```

Opciones para obtener las dimensiones de la pantalla:

- Para obtener el alto de la pantalla se puede utilizar la función:
  ```
  num heightVar = getHeight()
  ```
- Para obtener el ancho de la pantalla se puede utilizar la función:
  ```
  num widthVar = getWidth()
  ```

Funciones matemáticas:

- Función piso:
  ```
  num x= flor(3.5)
  ```
- Función techo:
  ```
  num x= ceil(3.5)
  ```
- Raíz cuadrada
  ```
  num x= sqrt(4)
  ```
- Potencia
  ```
  num x= pow(2,3) // = 2^3
  ```

Cambios de configuración del sistema:

- Si se desea cambiar la gravedad del sistema de la simulación se puede hacer:
  ```
  Config.gravityConstant(num numVal)
  ```

  Donde numVal es el valor de la constante gravitatoria del sistema. Si se desea deshabilitar la atracción gravitatoria, numVal debe ser igual a cero. Por default es 0.001

- Si se desea que los cuerpos no puedan rebotar con el borde de la pantalla:
  ```
  Config.worldBorderBounce(bool boolVal)
  ```

  Donde boolVal puede ser true y false dependiendo de si se quiere habilitar o deshabilitar esta opción. Por default es true

- Si se desea que los cuerpos de la simulación dejen visible el rastro por el que pasan:
  ```
  Config.enableBodyTrail(bool boolVal)
  ```

  Donde boolVal puede ser true y false dependiendo si se quiere habilitar o deshabilitar esta opción. Por default es false.

Creación de cuerpos:

Para crear un cuerpo en el plano se deben pasar las variables:

- Posición inicial en x (en pixeles)
- Posición inicial en y (en pixeles)
- Masa
- Radio (en pixeles)
- Velocidad inicial en x (en pixeles)
- Velocidad inicial en y (en píxeles)
- Color (Un string en forma hexadecimal. Por ejemplo “#123ABC”)

De la siguiente manera:

```
System.addBody(num posX, num posY, num masa, num radio, num velX, num velY, str color)
```

Opciones de input y output:

Para input, se utilizan las funciones **readStr()** y **readNum()**, la primera permite leer una cadena de caracteres y la segunda permite leer un número. Ambas funciones, se traducen en un window.prompt de JS. Para output se utiliza la función **print()** que recibe un string y lo muestra en pantalla, esto se traduce a un alert() de JS, Por ejemplo:

```
num numVar
str strVar

main(){
      numVar = readNum(“Ingrese un número”)
      strVar = readStr(“Ingrese una palabra”)
      print(“Se ingresó el número: ” + numVar)
      print(“Se ingresó la cadena: ” + strVar)
}
```

Dificultades encontradas en el desarrollo del TP.

Encontrar bugs en el código

Una de las partes más complicadas del desarrollo del trabajo práctico fue la de encontrar bugs. Generalmente, nos dabamos cuenta que había un error cuando el analizador sintáctico indicaba un syntax error. Encontrar y debuggear el origen del problema a menudo llevaba varias horas.

Pensar como queríamos hacer el lenguaje

El lenguaje que creamos pasó por varias iteraciones. Cada miembro tenía una imagen mental distinta de como debía estar estructurado el lenguaje por lo que ponernos de acuerdo entre todos para encontrar la forma más clara de transmitir las distintas acciones consumió bastante tiempo.

Problemas declarando y modificando variables

Inicialmente en el diseño de nuestro lenguaje teníamos pensado que se pudiera inicializar y modificar variables en cualquier lugar del código. Esto nos llevó a un problema debido a que a veces, al hacerse el análisis sintáctico podía ocurrir que encontrara una variable que haya sido declarada en el código pero el analizador no haya parseado su declaración aún; por lo que retornaba un error. Consultando este problema se nos sugirió tomar una decisión entre dos posibilidades: Hacer un análisis en dos pasadas donde en la primera se guardaran las variables y sus tipos y en la segunda se hiciera la traducción a JavaScript. O convertir la gramática para que contuviera una sección separada del resto del código donde se declaren todas las variables, al igual que ocurría en las versiones más primitivas de C. Observando la finalidad de nuestro lenguaje y la complejidad que tenía cada opción decidimos optar por modificar nuestra gramática y hacer que todas las variables del código se declaren al inicio del programa.
Adicionalmente, para resolver el problema de que un usuario le ponga cómo nombre a su variable el mismo que nosotros utilizamos en nuestras variables en Canvas, se optó por agregar el prefijo “u_” a las variables creadas por el usuario para que no haya conflicto. De esta manera, todas las variables del usuario se distinguen en el código JavaScript por su prefijo.

Futuras extensiones

Función para generar que un cuerpo orbite de forma estable a otro:

La idea de esta funcionalidad era poder tener una función que modifique ciertas variables iniciales de un objeto para que este orbite de forma circular a otro.
Para hacer esta funcionalidad, investigamos las ecuaciones para mecánica de órbitas circulares. El principal problema es que la constante gravitacional universal se encuentra en unidades $m^3/(kg*s^2)$. Adaptar estas unidades para que funcionen en pixeles y obtener una constante que se adecúe a nuestros sistemas para obtener valores iniciales que generen una órbita estable es una tarea extremadamente difícil.

Posibilidad de ingresar objetos estáticos inamovibles al espacio:

Para agregar esta funcionalidad, principalmente consistiría en crear un tipo de objeto que no pueda moverse de su posición inicial, que las reglas de colisión sean elásticas y que no afecten ni sean afectados por las reglas gravitacionales del sistema. Además, en cada frame se tendría que chequear la colisión de cada uno de los objetos normales con los inamovibles. Hacer todo esto no es del todo difícil pero tendríamos que agregar varias funciones adicionales al proyecto para permitir esta funcionalidad.

Objetos con distintos tipos de formas:

En este momento, el programa solo genera objetos circulares. Se podría extender para que se pudieran generar también distintos tipos de polígonos. Sin embargo, html canvas no ofrece funcionalidad nativa para crear estas formas geométricas por lo que cada una deberían ser generadas a mano. No sería muy difícil pero podría ser laborioso y largo.

Modificaciones a los tipos de impacto entre objetos:

Se podría agregar una opción para cambiar la forma de colisión entre los cuerpos como elástico, elástico o inelástico. A nivel físico, consiste en modificar el valor del coeficiente de restitución de las ecuaciones de estos choques para variar entre una y otra. Sin embargo, al intentar variar este valor mientras se aplica atracción gravitacional, se llegaba en muchos casos a resultados caóticos para el motor por lo que es una tarea difícil de hacer.

Testeos

Corrimos Strace y Valgrind para hacer un análisis de bugs y manejo de memoria y obtuvimos los siguientes resultados:

```
scott@scott-VirtualBox:~/TLA/TP_TLA/compiler$ CC=clang scan-build -disable-checker deadcode.DeadStores -o /tmp/out  make
scan-build: Using '/usr/lib/llvm-6.0/bin/clang' for static analysis
yacc -d -v parser.y
lex lexer.l
gcc -Wall -pedantic -std=c99 -Wextra -D_POSIX_C_SOURCE=200809L -std=c99 -Wno-unused-function -c lex.yy.c y.tab.c -lm
gcc -Wall -pedantic -std=c99 -Wextra -D_POSIX_C_SOURCE=200809L -std=c99 -Wno-unused-function -o compile lex.yy.o y.tab.o -lfl -lm
scan-build: Removing directory '/tmp/out/2020-11-27-125307-7656-1' because it contains no reports.
scan-build: No bugs found.
```

```
khalagard@khalagard-VirtualBox:/media/sf_git/TP_TLA/compiler$ valgrind --leak-check=full --show-leak-kinds=all --tool=memcheck -s ./compile ./sample_program1.phy
==7098== Memcheck, a memory error detector
==7098== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7098== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==7098== Command: ./compile ./sample_program1.phy
==7098==
==7098==
==7098== HEAP SUMMARY:
==7098==     in use at exit: 0 bytes in 0 blocks
==7098==   total heap usage: 281 allocs, 281 frees, 61,238 bytes allocated
==7098==
==7098== All heap blocks were freed -- no leaks are possible
==7098==
==7098== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Como verán, no se encontraron bugs ni memory leaks (lo mismo sucede con todos los programas de prueba).

Referencias

- Lex & Yacc - Levine
- https://www.w3schools.com/html/html5_canvas.asp
- https://developer.mozilla.org/es/docs/Web/HTML/Canvas
- https://es.wikipedia.org/wiki/Choque_inel%C3%A1stico
- https://es.wikipedia.org/wiki/Ley_de_gravitaci%C3%B3n_universal