Trabajo práctico especial
Autómatas, Teoría de Lenguajes y Compiladores

JAAA-lang
un lenguaje de múltiples lenguajes de salida

José Martín Torreguitar - legajo 57519

Agustín Dammiano - legajo 57702

Alan Donoso Naumczuk - legajo 57583

Agustín Emilio Izaguirre - legajo 57774

Índice

Introducción

En el presente documento se destacan las ventajas y desventajas del lenguaje jaaa-lang o simplemente jaaa, así como su motivación, objetivo y los problemas encontrados a lo largo de su realización junto con las decisiones tomadas frente a estos.

Objetivo

Jaaa-lang tiene como objetivo ser un lenguaje que le provee al usuario la capacidad de elegir entre eficiencia y portabilidad ¿Cómo? La solución es la de darle al usuario la posibilidad de elegir el lenguaje al cual el compilador convertirá lo escrito en Jaaa. Los lenguajes elegidos son c, si es que el usuario decide priorizar la eficiencia, o java, si es que decide priorizar la portabilidad.
Desde el punto de vista del usuario esto es tan simple como pasar el argumento -b al compilar, lo cual hará que el lenguaje de salida sea java, de lo contrario el lenguaje de salida será c.
Además de ofrecer eficiencia y portabilidad busca facilitar el manejo de memoria en la utilización de Strings, comparado con el manejo de memoria requerido al usar un lenguaje de bajo nivel como C.

Descripción del lenguaje

Pensando en el demográfico que utilizaría este lenguaje, esto es, gente ya familiarizada con las capacidades de ambos lenguajes de salida, se optó por una sintaxis similar a la de c/java pero simplificada. Para lograr esto se removieron los paréntesis en las expresiones, se reemplazaron los corchetes al principio/final de los ciclos y los condicionales por delimitadores escritos, por ejemplo:

```
while number != 0 do
                number = number - 1
loop
```

ejemplo de if:

```
if iseven == 1 do
                print "Is even" ;
else do
```

```
                    print "Is odd" ;
        end
```

Se eliminaron los ‘;’ al final de las sentencias y, finalmente, el tipo de las variables se obvia, este es reconocido en base al valor al cual se iguala la variable, luego de lo cual el tipo de la variable no podrá cambiar. Puesto que no es necesario especificar el tipo de la variable en su declaración se removieron del lenguaje las declaraciones que no asignan a la variable un valor.
Los ‘;’ son utilizados, sin embargo, para denotar el final de una lista de argumentos para una función, como lo es la función print o read.
También permite realizar operaciones relacionales con Strings (>, <, <=, >=, !=, ==) directamente sin necesidad de llamar a funciones auxiliares,como sucede en ambos lenguajes de salida.
Por último también facilita la declaración de strings para los usuarios familiarizados con C ya que su declaración es directa.

Ejemplo de declaración de String:
nombre = “string asociado”

GETTING STARTED JAAA

En esta sección detallaremos la sintaxis del lenguaje.

Empezaremos describiendo cómo debe ser la declaración de cada tipo de dato.
Para declarar una variable hace falta un identificador del mismo, los identificadores soportados por el lenguaje son los formados únicamente por letras del alfabeto inglés(desde la a hasta la z) en minúscula.

Como se explicó en la sección anterior JAAA es un lenguaje fuertemente tipado, pero a la vez no requiere la especificación de un tipo en su declaración. Por lo que explicaremos cómo debe ser la declaración de cada tipo de dato.
enteros: Para los tipos enteros

Con respecto a las formas en las que el programa puede comunicarse con su usuario (entrada y salida), nuestro lenguaje provee dos funciones internas, una que permite ofrecer una salida (print) y otra que permite ofrecer entrada de input(read), que pasaremos a describir a continuación

- Enteros: su valor debe estar formado únicamente por dígitos (0,1,2,3,4,5,6,7,8,9)
  Ejemplo:
  ```
  var = 10
  ```

- Flotantes: su valor debe estar formado únicamente por dígitos y un único punto(‘.’) con la restricción de que el punto no puede estar ni al principio ni al final de la secuencia de dígitos.
  Ejemplo:
  ```
  var = 10.0
  ```

- Booleanos: su valor puede ser o bien true o bien false
  Ejemplo:
  ```
  var = true
  var = false
  ```

- Strings: su valor deben ser únicamente caracteres ASCII con la restricción de que deben empezar y terminar con comillas dobles(“)
  Ejemplo:
  ```
  var = “1 string variable”
  ```

El lenguaje también soporta constantes cuya declaración es muy similar a las de las variables y pueden ser de los mismos tipos que las variables.
Para declarar una constante se debe especificar antes del nombre de la constante la palabra const.

Ejemplos:

```
const newconstant = 10
const stringconstant = “first constant”
```

Tanto las declaraciones de variables como las de constantes pueden aparecer en cualquier parte del código salvo dentro de cualquier bloque condicional o bloque de repetición y son válidas desde su declaración hasta el final del programa

Ahora empezaremos a describir las operaciones posibles que se pueden realizar con estos tipos de datos.

Operaciones Lógicas

Las operaciones lógicas soportadas son AND, OR y NOT y su sintaxis es idéntica a la de los lenguajes de salida (JAVA y C). Cada una de esta debe estar acompañada de una o más expresiones.

Estas operaciones solo pueden aparecer del lado derecho en una asignación o declaración, tanto de una variable como de una constante o en la condición de un bloque condicional o un bloque de repetición.

Una expresión puede ser una variable, un valor de cualquiera de los detallados en la declaración de variables, o operaciones aritméticas, relacionales y/o lógicas de los mismos donde además pueden aparecer paréntesis para dar prioridad a alguna expresión

- AND: es un operador binario(opera entre dos expresiones) y su formato es expresión1 && expresión2, o bien (expresión1 && expresión2) con la restricción de que ni la expresión1 ni la expresión2 sean del tipo string, cualquier otro tipo de dato es aceptado, en los enteros y flotantes el 0 es el único falso y los demás son verdaderos, su valor de retorno es booleano y representa el AND lógico
  Ejemplos:

```
true && false
10 && (true && true)
(false && 15)
```

- OR: es un operador binario(opera entre dos expresiones) y su formato es expresión1 || expresión2, o bien (expresión1 || expresión2) con la restricción de que ni la expresión1 ni la expresión2 sean del tipo string, cualquier otro tipo de dato es aceptado, en los enteros y flotantes el 0 es el único falso y los demás son verdaderos, su valor de retorno es booleano y representa el OR lógico
  Ejemplos:

```
true || false
10 || (true || true)
(false || 15)
```

- NOT: es un operador unario(una única expresión) y su formato es !expresión, o bien (!expresión1) con la restricción de que la expresión no sea del tipo string, cualquier otro tipo de dato es aceptado, en los enteros y flotantes el 0 es el único falso y los demás son verdaderos, su valor de retorno es booleano y representa el NOT logico
  Ejemplos:

```
!true
! false
!10
```

  Ejemplo de expresión más compleja:

```
((10 + 4) || true) && (!1)
```

Operaciones Aritméticas

Las operaciones aritméticas soportadas son SUMA, PRODUCTO, RESTA , DIVISIÓN y NEGACIÓN. Su sintaxis es idéntica a la de los lenguajes de salida (JAVA y C). Cada una de esta debe estar acompañada de una o más expresiones.

Estas operaciones solo pueden aparecer del lado derecho en una asignación o declaración, tanto de una variable como de una constante o en la condición de un bloque condicional o un bloque de repetición.

- SUMA: es un operador binario(opera entre dos expresiones) y su formato es expresión1 + expresión2, o bien (expresión1 + expresión2) con la restricción de que ni la expresión1 ni la expresión2 sean del tipo string, cualquier otro tipo de dato es aceptado, en los booleanos el true es tomado como 1 y el false como 0. Su valor de retorno es el más específico de los tipos involucrados en la expresión, donde el más específico es el flotante, luego sigue el entero y por último el booleano. Sin embargo si ambas expresiones son booleanas el valor de retorno es entero.
  Ejemplos:

```
true + false
(10 + 4)
5 + 10.0
```

- RESTA: es un operador binario(opera entre dos expresiones) y su formato es
  expresión1 - expresión2, o bien (expresión1 - expresión2) con la restricción de que ni la expresión1 ni la expresión2 sean del tipo string, cualquier otro tipo de dato es aceptado, en los booleanos el true es tomado como 1 y el false como 0. Su valor de retorno es el más específico de los tipos involucrados en la expresión, donde el más específico es el flotante, luego sigue el entero y por último el booleano. Sin embargo si ambas expresiones son booleanas el valor de retorno es entero.
  Ejemplos:

```
true - false
(10 - 4)
5 -10.0
```

- PRODUCTO: es un operador binario(opera entre dos expresiones) y su formato es expresión1 * expresión2, o bien (expresión1*expresión2) con la restricción de que ni la expresión1 ni la expresión2 sean del tipo string, cualquier otro tipo de dato es aceptado, en los booleanos el true es tomado

como 1 y el false como 0. Su valor de retorno es el más específico de los tipos involucrados en la expresión, donde el más específico es el flotante, luego sigue el entero y por último el booleano. Sin embargo si ambas expresiones son booleanas el valor de retorno es entero.
Ejemplos:

```
true * false
(10 * 4)
5  * 10.0
```

- DIVISIÓN: es un operador binario(opera entre dos expresiones) y su formato es expresión1 / expresión2, o bien (expresión1/expresión2) con la restricción de que ni la expresión1 ni la expresión2 sean del tipo string, cualquier otro tipo de dato es aceptado, en los booleanos el true es tomado como 1 y el false como 0. Su valor de retorno es el más específico de los tipos involucrados en la expresión, donde el más específico es el flotante, luego sigue el entero y por último el booleano. Sin embargo si ambas expresiones son booleanas el valor de retorno es entero.
  Ejemplos:

```
true / false
(10 / 4)
5  / 10.0
```

- NEGACIÓN: es un operador unario y su formato es -expresión, o bien (- expresión) , con la restricción de que la expresión no sea del tipo string, cualquier otro tipo de dato es aceptado, en los booleanos el true es tomado como 1 y el false como 0. Su valor de retorno es el más específico de los tipos involucrados en la expresión, donde el más específico es el flotante, luego sigue el entero y por último el booleano. Sin embargo si ambas expresiones son booleanas el valor de retorno es entero.
  Ejemplos:

```
-true
(-10 )
-5.0
```

Operaciones Relacionales

Las operaciones relacionales soportadas son >,<, <=, >=, == y !=. Su sintaxis es idéntica a la de los lenguajes de salida (JAVA y C). Cada una de esta debe estar acompañada de dos expresiones.

Estas operaciones solo pueden aparecer del lado derecho en una asignación o declaración, tanto de una variable como de una constante o en la condición de un bloque condicional o un bloque de repetición.

El formato de una operación relacional es el siguiente:
expresión OPERADOR expresión, o bien (expresión OPERADOR expresión), donde OPERADOR puede ser: ‘<’, ‘<=’, ‘>’, ‘>=’, ‘!=’, ‘==’.,
Estas operaciones aceptan todos los tipos de datos, pero tienen la restricción de que si alguna de las dos expresiones es del tipo string para que sea válida la otra tambien deberia serlo.

El valor de retorno de estas expresiones es booleano.

Si alguna expresión es booleana, se toma a true como 1 y a false como 0 y la semántica de la misma es la misma que en la matemática, en cambio si ambas expresiones son de tipo string los operadores adquieren un significado que no es el habitual:

> : Devuelve true si el primer string se encontrará después del segundo en un diccionario. Sino retorna false.

>= : Devuelve true si el primer string se encontrará después del segundo en un diccionario, o bien que sean iguales. Sino retorna false.

< : Devuelve true si el primer string se encontrara antes del segundo en un diccionario. Sino retorna false.

<= : Devuelve true si el primer string se encontrara antes del segundo en un diccionario, o bien que sean iguales. Si no retorna false.

!= : Devuelve true si ambos strings son distintos. Si no retorna false.

== : Devuelve true si ambos strings son iguales. Si no retorna false.

Ejemplos:

```
“primer string” < “segundo string”
10 > 5
true != false
“string” == “string”
100 >= 100
15 <= 5
```

Comentarios

No se aclaró aún, pero la sintaxis ignora tanto espacios como saltos de línea. Salvo los incluidos en un string.

Únicamente soporta comentarios unilinea y comienzan con "//" de ahí hasta el final de la línea es ignorado por el lenguaje. Este tipo de comentario es compatible con los comentarios unilinea de los lenguajes de salida

Ejemplo:
// esto es un comentario en JAAA

Bloques Condicionales

Nuestro lenguaje soporta un único tipo de bloque condicional el IF, pero tiene varias variantes. Estos bloques pueden aparecer en cualquier parte salvo en la condición de otro bloque o en una asignación o declaración.

- IF: el formato del if es if condicion DO bloque_de_sentencias END, en la condición se espera una expresión y entre el DO y el END una lista de sentencias a ejecutar en caso de que la condición sea verdadera.
  Ejemplo:

```
if 4 > 2 do
   if 3 > 2 do
     print "dentro del if de adentro";
  end
  println "dentro del if de afuera";
end
```

  Como se ignoran los saltos de linea el codigo exhibido en el ejemplo podría aparecer en una única línea.

- IF-ELSE-IF: el formato es if condicion do bloque_de_sentencias else if condicion do bloque_de_sentencias end
  Tanto la condición como el bloque de sentencias siguen teniendo que cumplir lo mismo que en el if.
  Ejemplo:

```
a = 10
b = 5
if a > b do
     println a, "es mayor que", b;
```

```
else if b>a do
     println b,”es mayor que”, a;
end
```

Luego del primer if pueden venir tantos else if como se quiera.

Como se ignoran los saltos de linea el codigo exhibido en el ejemplo podría aparecer en una única línea.

- IF-ELSE: el formato es if condicion do bloque_de_sentencias else do bloque_de_sentencias end
  Tanto la condición como el bloque de sentencias siguen teniendo que cumplir lo mismo que en el if.
  Ejemplo:

```
a = 10
b = 5
if a > b do
     println a, “es mayor que”, b;
else do
     println b,”es mayor o igual a”, a;
end
```

  Como se ignoran los saltos de linea el codigo exhibido en el ejemplo podría aparecer en una única línea.

- IF-ELSE-IF-ELSE: el formato es if condicion do bloque_de_sentencias else if condicion do bloque_de_sentencias else do bloque_de_sentencias end
  Tanto la condición como el bloque de sentencias siguen teniendo que cumplir lo mismo que en el if.
  Ejemplo:

```
a = 10
b = 5
if a > b do
     println a, “es mayor que”, b;
else if b == a do
     println a, “es igual a”,b;
else do
     println b,”es mayor que”, a;
end
```

  Entre el if y el else pueden aparecer tantos else if como se quiera.

Como se ignoran los saltos de linea el codigo exhibido en el ejemplo podría aparecer en una única línea.

Bloques de ciclos repetitivos

El lenguaje soporta cuatro tipos de ciclos repetitivos: while, do-while, until y do-until. Estos pueden aparecer en cualquier parte excepto en una condición, asignación o declaración.

- **Ciclo while**
  La sintaxis del ciclo while es la siguiente:

```
while condition do
     statements
loop
```

  Siendo un ejemplo de un programa que imprime en pantalla los números del 1 al 10 el siguiente:

```
i = 1
while i <= 10 do
     print i;
loop
```

- **Ciclo do-while**
  La sintaxis del ciclo do-while es la siguiente:

```
do
     statements
loop while condition
```

  Siendo un ejemplo de un programa que lee caracteres de entrada estándar hasta leer EOF el siguiente:

```
const eof = -1
c         =  0
do
     c = read;
loop while c != eof
```

- **Ciclo until**

La sintaxis del ciclo until es la siguiente:

```
until condition do
     statements
loop
```

Siendo un ejemplo de un programa que imprime en pantalla los números del 1 al 10 el siguiente:

```
i = 1
until i > 10 do
     print i;
loop
```

- **Ciclo do-until**
  La sintaxis del ciclo do-until es la siguiente:

```
do
     statements
loop until condition
```

Siendo un ejemplo de un programa que lee caracteres de entrada estándar hasta leer EOF el siguiente:

```
const eof = -1
c         =  0
do
     c = read;
loop until c == eof
```

Es importante destacar que la elección de las palabras claves **do** y **loop**, para delimitar el bloque de statements a realizarse dentro del ciclo, en lugar de llaves o cualquier otra alternativa fue basada en la naturalidad del código al leerlo.

Por ejemplo, si tenemos el siguiente código de un ciclo do-while:

```
counter = 0
do
     statements
     counter = counter + 1
loop while counter < 5
```

Se puede leer de forma considerablemente clara como “Do *statements*, loop while counter is less than five” y creemos que esto aporta mucha claridad al código, y es incluso simple de entender para alguien que recién comienza a programar.

Print

Es una función interna provista por el lenguaje para permitirle al programa mostrarle un mensaje al usuario, tiene dos variantes print y println.

El formato de ambas variantes es el siguiente,

```
print TEXTO;    println TEXTO;
```

o bien

```
print TEXTO, TEXTO;    println TEXTO, TEXTO;
```

donde TEXTO es o bien un string(siempre entre comillas dobles “) o una variable, pueden aparecer tantos strings o textos como se quiera siempre y cuando se separen con coma(‘,’).

- print: Imprime el o los textos que recibe como parámetro concatenados y separados por espacios.
  Ejemplos:

```
a = “hola”
print “variable a vale:”, a;
print a;
print “es un texto”;
```

- println: Imprime el o los textos que recibe como parámetro concatenados y separados por espacios agregando al final un salto de línea.
  Ejemplos:

```
a = “hola”
println “variable a vale:”, a;
println a;
println “es un texto”;
```

Read

Es una función interna del lenguaje que le permite al programa recibir datos del usuario. El formato es el siguiente:

```
read;
```

o bien

```
read ENTERO characters;
```

o bien

```
read VARIABLE characters;
```

Cabe destacar que la variable debe contener un entero. Por ejemplo:

```
length = 10
str = read length characters;
```

Este código lee 10 caracteres del usuario y los guarda en la variable str.

Para el caso de read sin parámetros por default lee un único carácter.
Ejemplos:

Ejemplo 1:
caracter = read;

Ejemplo2:
read;

En el ejemplo 1 se asigna el carácter leído a la variable caracter.

En el ejemplo 2 se lee un caracter pero no se asigna a ninguna variable, por lo que es equivalente a ignorar un carácter.

Exit

Es otra función interna del lenguaje que se utiliza para terminar con la ejecución del mismo análogo al exit() de c o al System.exit() de JAVA, su formato es simplemente exit.
Ejemplo:

```
exit
```

Gramática

statement_list: **statement**
| statement statement_list
;

statement: **NAME '=' expression**
| CONST NAME '=' expression
| printExpression

```
                | conditional
                | exit_statement
                | while_loop
                | readExpression
                | NAME '=' readExpression


expression:     expression '+' expression
                | expression '-' expression
                | expression '*' expression
                | expression '/' expression
                | '-' expression %prec UMINUS
                | '(' expression ')'
                | expression '<' expression
                | expression LTOET expression
                | expression '>' expression
                | expression GTOET expression
                | expression ET expression
                | expression NET expression
                | expression AND expression
                | expression OR expression
                | NOT expression
                | FLOAT
                | INTEGER
                | STRING
                | BOOL
                | NAME
                ;


printExpression: PRINT_TEXT text ';'
                | PRINT_TEXT_NEW_LINE text ';'
                ;


      text:     STRING ',' text
                | NAME ',' text
                | STRING
                | NAME
                ;
```

```
while_loop:        WHILE expression DO statement_list LOOP
                   | UNTIL expression DO statement_list LOOP
                   | do_loop_statement
                   ;

do_loop_statement: DO statement_list LOOP final_condition
                   ;

final_condition:   WHILE expression
                   | UNTIL expression
                   ;


exit_statement:    EXIT
                   ;


conditional:       IF expression DO statement_list END
                   | IF expression DO statement_list else_block
                   ;


else_block:        ELSE IF expression DO statement_list END
                   | ELSE IF expression DO statement_list else_block
                   | ELSE DO statement_list END
                   ;


readExpression:    READ_TEXT ';'
                   | READ_TEXT NAME CHAR ';'
                   | READ_TEXT INTEGER CHAR ';'
                   ;
```

En la gramatica expuesta las palabras en mayuscula son los TOKEN o simbolos terminales de la gramatica, cuyo nombre trata de ser autodescriptivo y las palabras en minuscula son los simbolos no terminales de la gramatica, cada producion esta indicada por ‘:’ y entre dos producciones del mismo no terminal se separan con ‘|’

Explicaremos brevemente a continuación el sentido de cada no terminal.

statement-list: símbolo inicial de la gramática, tiene dos producciones, las cuales pueden ser una única sentencia (statement) o una sentencia seguida de una statement-list.

statement: símbolo que representa tanto las declaraciones de variables, de constantes, los ciclos, los condicionales, las expresiones de escritura (print) o lectura (read) o una declaración de salida (exit_statement).

expression: la unidad lógica del lenguaje la cual permite todas las operaciones (aritméticas, lógicas y relacionales), como valores de cada tipo de datos y nombres de variables.

printExpression: expresión para declarar escritura a salida estándar.

text: el texto a ingresar para la función print (puede consistir de tanto texto como números y variables o listas de estos.

while_loop: representa un ciclo while común, un ciclo until (se realiza hasta que la condición se cumpla) o un do-loop-statement.

do_loop_statement: realiza la acción indicada luego del do la cual puede estar seguida por una condición until o while.

conditional: permite realizar ifs o ifs seguidos o no de un else_block.

else_block: permite seguir los condicionales con else o else ifs.

read_expression: expresión para declarar lectura de entrada estandard, puede indicar cuantos caracteres leer o no en cuyo caso lee un único caracter.

Limitaciones y restricciones

JAAA solo reconoce 4 tipos de datos(enteros, booleans, flotantes y string) y para el caso de los string hay restricciones especiales. Estas son que los strings no pueden ser parte de operaciones lógicas (NOT, OR , AND) ni de operaciones aritméticas (SUMA, PRODUCTO, RESTA ,DIVISIÓN y NEGACIÓN).

Las variables no pueden declararse dentro de ningún bloque de repetición ni de ningún bloque condicional, debido a problemas que ocurrían con el scope de las variables.

El usuario no puede crear funciones.

Hay pérdidas de memoria debido a que no toda la memoria alocada puede liberarse.

Las variables y constantes se guardan en una lista simplemente encadenada por lo que cada vez que se utiliza o declara una nueva variable o constante puede llegar a tener que recorrerse toda la lista (O(N)) y no es tan eficiente como se quisiera.

Dificultades encontradas en el desarrollo del TP

Al desarrollar la gramática surgieron algunos conflictos shift/reduce que pudimos entender y resolver gracias a las conocimientos adquiridos en la materia, especialmente en la última parte donde veíamos cómo se producían los conflictos y cómo evitarlos.

En algunos casos tuvimos que factorizar y en otros agregar un delimitador para que el siguiente no coincida con alguno de los siguientes del que se reduce como en el caso del read.

También fue complicado cambiar un poco la forma de pensar, debido a que en YACC no podemos elegir en qué momento ejecutar la acción asignada a la producción

Otra dificultad que tuvimos fue como hacer para incluir los statement dentro de los bloques(tanto de repetición como los condicionales) ya que si hacíamos la traducción directamente los statement aparecerian antes de que nos dieramos cuenta que estábamos dentro de un bloque. Para este problema implementamos una estructura que servía como un AST (Abstract Syntax Tree) para de esta manera recorriendolo poder generar los statements en el orden esperado.

Utilización del Compilador

Para utilizar facilitar el uso del compilador de JAAA al usuario se elaboró un script en bash que simplifica en gran medida su utilización. Los requisitos son tener instalado gcc, java, lex, yacc y make en la computadora donde se ejecute.

El script se llama jaaa y esta en la carpeta raiz del proyecto, para compilar un archivo lo único que debe hacerse es ejecutar ese script pasandole un parametro si quiere especificar la salida como bytecode de JAVA y el path del archivo que se quiere compilar.

El script se encarga de verificar que la cantidad de parámetros y el orden de los mismos sean válidos y si el archivo a compilar existe y tiene la extensión requerida (“.jaaa”). Luego borra los archivos generados si hace falta (make clean) y genera los nuevos archivos necesarios para compilar el programa. Finalmente genera un ejecutable en la carpeta raíz con el nombre del archivo que se le pasó.

Aclaraciones sobre el uso del compilador, en primer hay dos posibles llamados al script (para lenguaje de salida C y JAVA), para que el lenguaje de salida sea C debe ejecutarse el comando ./jaaa path_del_archivo (ya que es la opción default del compilador) y para que el lenguaje de salida sea java debe ejecutarse el comando ./jaaa -b path_del_archivo (es importante respetar el orden de los parámetros ya que en caso contrario se tomara como invalido).

Otra aclaración es que si por alguna razón el usuario elimina el archivo ejecutable compiler en el directorio JAAA-compiler, para que el script funcione correctamente deberá ir a la carpeta JAAA-compiler y ejecutar manualmente make clean, luego puede seguir utilizando el script normalmente.

Por último si se utilizó el parámetro -b la salida será un archivo con el bytecode de JAVA, pero como Java requiere que sus archivos tengan el mismo nombre que la clase principal y que los nombres de las clases empiezan en mayúscula y nosotros no queríamos imponer esa restricción en la nomenclatura de los archivos .jaaa,se optó por generar siempre un Main.class y un script en bash con el nombre del archivo cuya única instrucción es ejecutar el bytecode Main.class.Por esta razón tanto el script como el Main.class deben estar en la misma carpeta y si se quiere transferir el ejecutable a otra persona u otra máquina debe tenerse en cuenta que el ejecutable es Main.class y no el script con el nombre del archivo. También por esta razón si luego de compilar un programa se compila otro el Main.class corresponderá al bytecode del último programa compilado, por lo que si se quiere guardar el anterior debe moverse el Main.class a otra carpeta que se quiera.

Se decidió utilizar este script que ejecute el Main.class para que sea más fácil para el usuario ejecutar su código y consistente con cómo debe hacer cuando la salida es en C, aunque para lograr eso se oculta un poco la realidad de que en realidad el ejecutable es el Main.class y no el nombre del archivo como en C.

Ejemplos de uso:
Para los ejemplos vamos a suponer que existe un programa sintácticamente y semánticamente correcto en JAAA llamado programa.jaaa

Para lenguaje de salida en C:

1. **Asignar permiso de ejecución al script si no lo tiene:**
   chmod +x jaaa
2. **Generar el ejecutable**:
   ./jaaa programa.jaaa
3. **Ejecutar el programa:**
   ./programa

Para lenguaje de salida en JAVA:

4. **Asignar permiso de ejecución al script si no lo tiene:**
   chmod +x jaaa
5. **Generar el ejecutable**:
   ./jaaa -b programa.jaaa
6. **Ejecutar el programa:**
   ./programa
   o bien manualmente:
   java Main.class

Futuras extensiones

- Mejorar la eficiencia del programa guardando las variables en una hash table, haciendo su compilación más eficiente. La complejidad de esta extensión es el desarrollo de la hash table en C.

- Poder declarar variables dentro de un bloque condicional o un bloque de repetición. Para esto necesitamos identificar cada posible scope (posiblemente con un número) y en el momento de la declaración de la variable o constante asignarle ese scope. Luego en la utilización de cualquier variable o constante deberíamos verificar si el scope de la variable incluye al scope actual. La complejidad de esta extensión radica más que nada en la implementación de la función que determina la inclusión de los scopes, ya que asignarle un scope a una variable es simplemente agregar un campo con su scope y para determinar el scope del bloque es agregar un campo a cada bloque con su scope.

- Agregar nuevos tipos de datos como arreglos. La complejidad de esta extensión es la de diseñar la sintaxis necesaria para el mismo y luego añadir la gramática que permita su utilización, ya que tanto C como Java poseen arreglos su traducción no debería ser muy compleja, salvo que se quiera

permitir arreglos cuyos elementos son de distintos tipos en cuyo caso la complejidad aumenta.

- Agregar un modo intérprete. Esta es una extensión que no amplía el poder lenguaje sino que le da más utilidad al usuario, debido a que muchas veces uno quiere probar algo y no quiere tener que poner a hacerse el código y luego compilarlo para recien ahi ejecutarlo. Para ello se ejecutará el script que diseñamos para correr el compilador con el parámetro -i. La complejidad de esta extensión es terminar de añadir la nueva funcionalidad del lenguaje al modo intérprete para que ejecute la acción en vez de traducirla, pues al principio habíamos hecho la funcionalidad básica como intérprete y puede verse en el código lineas comentadas con el texto modo intérprete, para utilizar en la futura extensión.
- Generar un ejecutable en bytecode de JAVA con el nombre del archivo pasado como entrada al compilador con la primer letra en mayúscula en caso de que el mismo sea en minúscula. La complejidad de esta extensión consiste en que el script que llama al compilador del programa pase también como parámetro el nombre del archivo para que al momento de armar el codigo en java el nombre de la clase principal sea el nombre del archivo con la primera letra en mayúscula si esta estuviera en minúscula originalmente.

- Por último si se quisiera (aunque no era la idea original con la que se penso el lenguaje) podríamos transformar JAAA en un lenguaje con tipado dinámico.
  Como no utilizamos identificadores para el tipo de las variables, sino que el tipo de las mismas es determinado por el valor que se les asigna en su declaración. La complejidad de la extensión consiste en verificar en cada asignación si el tipo de la variable coincide con el tipo del valor que se quiere asignar y si no coincide, al traducirlo crear una nueva variable en el lenguaje de salida con algún identificador no admitido en nuestro lenguaje(para evitar colisiones de nombre) y registrar que cuando en nuestro lenguaje se hable de la variable con ese nombre en realidad se refiere a la nueva variable creada con el nuevo tipo.
  Para agregar esta funcionalidad en el modo intérprete es más fácil ya que directamente podríamos asignar el nuevo tipo de datos a la variable sin problemas.

Conclusiones

La realización de este trabajo práctico permitió la apreciación de los conceptos teórico prácticos aprendidos en clase, dándoles a estos una aplicación del mundo real lo cual también resultó en una solidificación de estos.

La utilización de las herramientas proporcionadas fue satisfactoria por lo que se piensa en agregarle mayores funcionalidades al lenguaje en el futuro de ser posible.

El construir nuestro propio lenguaje también resultó en la posibilidad de una mejor comprensión del funcionamiento subyacente de otros lenguajes y como estos derivan sus gramáticas.

Referencias

1. Aho, Lam, Sethi, Ullman (2008), Compiladores, Principios, Técnicas y Herramientas, Addison, Wesley.(El Libro del Dragón).

2. John R.Levine, Tony Mason, Doug Brown (1992), lex & yacc, O'Reilly