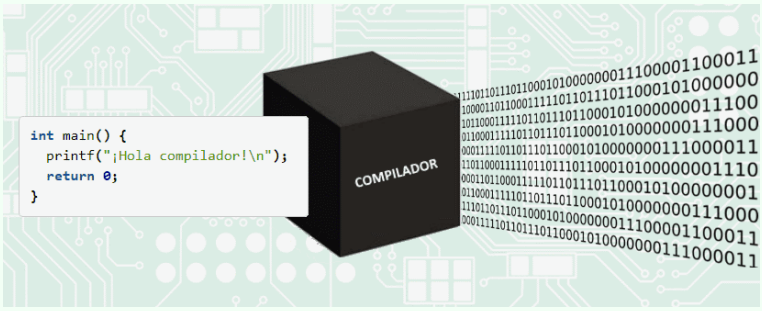
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE CIENCIAS, 2022-2
Compiladores 2023-2



Proyecto

Compilador de Jelly a Java

PROFESOR:

Lourdes del Carmen González Huesca

AYUDANTES:

Braulio Aaron Santiago Carrillo

Ma. Fernanda Mendoza Castillo

Juan Alfonso Garduño Solís

Lobitos Compilados:

Ortega Garcia Alejandra - 420002495

Pedro Mendez Jose Manuel - 315073120

Ramirez Gutierrez Oscar - 419004183

Villanueva Garcia Israel - 317052147

Aguilera Moreno Adrián - 421005200

Gutierrez Medina Sebastian Alejandro - 318287021

12 de junio de 2023

1. Desarrollo del proyecto

Además del ejercicio obligatorio, el de realizar la compilación completa de `Jelly` a `Java`, para el proyecto decidimos realizar una extensión del lenguaje para que se admita el ciclo `for` y también se agrego un `script` que nos permite ejecutar el código compilado al lenguaje objetivo.

La gramática de nuestro proyecto es la siguiente:

```
(define-language jelly
  (terminals
   (constante    (c))
   (id           (i))
   (operador     (op))
   (tipo         (t)))

  (Programa (p)
            (programa m)
            (programa m (cp* ...)))

  (Main (m)
        (main (s* ...)))

  (Cuerpo (cp)
          met
          func)

  (Funcion (func)
           (funcion i (d* ...) (s* ...)))

  (Metodo (met)
          (metodo i (d* ...) t (s* ... s1)))

  (Sentencia (s)
             (if-s e0 (s1* ...) (s2* ...))
             (while e0 (s1* ...))
             (= e s)                        ; ASIGNACION
             (return e)
             d
             e)

  (Decla (d)
         (: i t))

  (Expr (e)
        (length i)               ; LENGTH
        (i (e* ...))             ; LLAMADA
        (arr-estruc e* ...)      ; ESTRUCTURA DEL ARREGLO
        (if-c e0 e1 e2)          ; IF CORTO / OPERADOR TERNARIO
        (get-elem i e1)          ; ACCEDER A POSICION DEL ARREGLO
        (op e0 e1)
        (op e0)
        c
        i))
```

2. Compilador de Jelly a Java.

Para traducir a *Java* nuestro lenguaje, usamos *nanopass-case* para analizar cada uno de los patrones del lenguaje *Jelly* para poder redefinir la sintaxis que teníamos en *Jelly* a su correspondiente traducción en *Java*, esto nos sirve porque los detalles dependían de cada tipo de sentencia y/o patrón dados por los lenguajes.

Algunos de los detalles más importantes considerados al momento de la traducción:

- La inclusión del carácter especiales-necesarias en java: el **;** para delimitar cada instrucción.
- Uso esporádico del **"\n"** pese a que no estamos hablando de un lenguaje indentado.
- La traducción del nombre de la clase en `Java` a partir del nombre del archivo de entrada es un enfoque pragmático para mantener una correspondencia uno a uno entre los programas de origen y destino.
    - `Java` tiene ciertas convenciones de nomenclatura, como el hecho de que las clases deben comenzar con una letra mayúscula y que los nombres de las variables y métodos deben comenzar con una letra minúscula. Para asegurarse de que el código generado se adhiera a estas convenciones nos encargamos de que el nombre de cada clase empiece con una mayúscula y también hemos tenido que garantizar que las funciones de `Jelly` se traduzcan adecuadamente a métodos estáticos en `Java`, lo que puede incluir asegurarse de que todas las dependencias de variables estén correctamente manejadas.
- Nombre diferente para los tipo booleanos, en *Java* es boolean.
- La gestión de los arreglos en *Java* es especialmente delicada, ya que *Java* tiene una semántica diferente para los arreglos en comparación con muchos otros lenguajes. Esto se maneja a través de una tabla hash que rastrea los arreglos declarados en el programa.

La implementación de la solución se encuentra en el archivo `proyecto.rkt`. En la implementación dada encontramos la función `get-rep-java` encargada de traducir un programa de Jelly en un programa de Java, esta la ocuparemos para ejecutar todo con ayuda de nuestro script.

La función realiza lo siguiente:

1. Usa `aceptado?` en el archivo de entrada para verificar si es un programa válido en Jelly.
2. Si el archivo es aceptado, entonces renombra las variables del programa utilizando la función `rename-var`.
3. Crea una tabla de tipos usando la función `tipos-programa`, que genera información sobre los tipos de las variables en el programa.
4. Realiza una comprobación de tipos en el programa renombrado utilizando la función `type-check`
5. Finalmente, se traduce el programa al código Java utilizando la función `java-programa`.

En la última línea del script `proyecto.rkt` se encarga de tomar el primer argumento de la línea de comando y el nombre de la clase previamente obtenido, los argumentos de la línea de comandos son pasados como un vector, y en Racket los índices de los vectores empiezan en 0, por lo que:

```
(display (get-rep-java (vector-ref (current-command-line-arguments) 0) nombre-clase))
```

Solo recuperará el primer argumento y el nombre de la clase, luego los pasa a la función get-rep-java para iniciar el proceso de `get-rep-java`.

3. Extensión del lenguaje para que admita el ciclo for.

El **for** implementado es azucar sintáctica del la estructura while creada con anterioridad.

La implementación del **for** requirió de la modificación del archivo `lexer.rkt` para agregar el `token-FOR` y símbolo `for`. También fue necesario modificar el archivo `parser.rkt` para anexar la regla que deriva el `for` a una lista que contiene una de las declaraciones que tiene cómo argumento el `for` y la forma de la estructura `while` que ya teniamos en el parser.

A grandes rasgos, una estructura cómo

```
main() {
    for(i:int=0, i<3, i++) {i + 4}
}
```

se traduce a una lista que contiene la declaración de `i`, el `while` con expresión de argumento igual a la segunda expresión de entrada en el `for`, y el cuerpo del `while` creado será una lista con el cuerpo del `for` y la tercer entrada en el mismo. Véase el ejemplo anterior cómo ejemplo en el parser.

4. Script para ejecutar el código compilado al lenguaje objetivo.

Tomando en cuenta lo denotado en la sección 1.1, ahora tenemos que el script `proyecto.sh` realiza lo siguiente:

1. Extrae el nombre base del archivo de entrada, excluyendo la extensión, usando el argumento `$1` que es el primer argumento que se pasa al script `proyecto.sh`. Por lo tanto, si el script se ejecuta con un archivo llamado `ejemplo.txt`, `filename` sería `ejemplo` y `extension` sería `txt`.
2. Convierte la primera letra del nombre del archivo (sin la extensión) a mayúscula. Utiliza el comando `sed` para esta operación de sustitución.
3. Ejecuta un script de Racket llamado `proyecto.rkt` con dos argumentos: el archivo original (`$1`) que usaremos para llamar así a nuestra clase en `Java` y el nombre del archivo con la primera letra en mayúscula (`$filename_capitalized`). La salida de este script de Racket, que se espera que sea código Java, se redirige a un archivo cuyo nombre es el nombre del archivo original con la primera letra en mayúscula y la extensión `.java`.
4. Finalmente, intenta compilar el archivo Java resultante con el nombre del archivo original con la primera letra en mayúscula usando el compilador Java, `javac`.

5. Instrucciones para ejecutar el código.

1. Primero, darle los permisos de ejecución a `proyecto.sh` y `proyecto.rkt`:

   ```
   chmod +x proyecto.sh proyecto.rkt
   ```

2. Luego, ejecutar el script:

   ```
   ./proyecto.sh <ruta-archivo>
   ```

   Por ejemplo:

   ```
   ./proyecto.sh ./pruebas/lp.jly
   ```

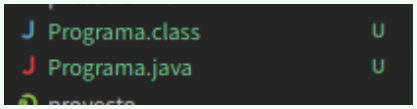
6. Ejemplos para cada ejercicio seleccionado

1. Ejercicio obligatorio:

```
kiriko@GLaDOS:~/Documentos/Compiladores/proyecto-lobitos-compilados$ racket
Welcome to Racket v8.8 [cs].
> (enter! "proyecto.rkt")
88 shift/reduce conflicts
"public class Programa { \n public static void main (String[] args) { \n int[] arr; \n arr
 = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}; \n in
t b; \n b = binary_search (arr, 11); \n } \n static int binary_search (int[] a, int x) { \
n int n; \n n = a.length; \n int l; \n l = 0; \n int r; \n r = n - 1; \n while (l < r) { \
n int m; \n m = l + r / 2; \n if (a[m] < x) { \n l = m + 1; \n } \n else { \n r = m; \n };
 \n }return l; \n } \n }"
"proyecto.rkt">
```

2. Ejercicio del script para ejecutar el código.

```
kiriko@GLaDOS:~/Documentos/Compiladores/proyecto-lobitos-compilados$ ./proyecto.sh ./pruebas/sort.txt
88 shift/reduce conflicts
```

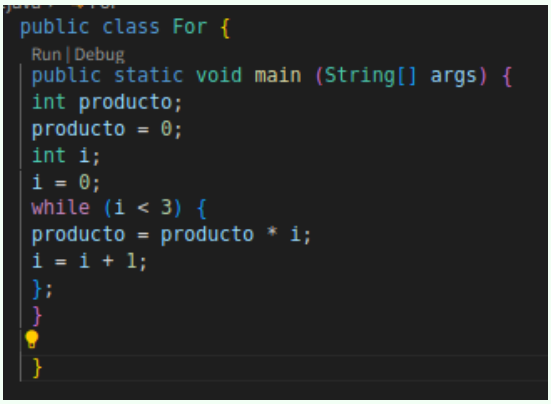



```
public class Programa {
 Run | Debug
 public static void main (String[] args) {
 int[] b;
 b = new int[]{1, 2, 3};
 sort (b);
 }
 static void sort (int[] a) {
 int i;
 i = 0;
 int n;
 n = a.length;
 while (i < n) {
 int j;
 j = i;
 while (j > 0) {
 if (a[j - 1] > a[j]) {
 int swap;
 swap = a[j];
 a[j] = a[j - 1];
 a[j - 1] = swap;
 }
 else {
 ;
 };
 j = j - 1;
 };
 i = i + 1;
 };
 }
```

3. Ejercicio For:

```
kiriko@GLaDOS:~/Documentos/Compiladores/proyecto-lobitos-compilados$ ./proyecto.sh ./pruebas/for
88 shift/reduce conflicts
kiriko@GLaDOS:~/Documentos/Compiladores/proyecto-lobitos-compilados$
```

```
main() {
  producto :int = 0
  for(i:int=0, i<3, i++) {
  producto = producto * i
  }
}
```



```
public class For {
  Run | Debug
  public static void main (String[] args) {
  int producto;
  producto = 0;
  int i;
  i = 0;
  while (i < 3) {
  producto = producto * i;
  i = i + 1;
  };
  }
  }
```

En general tenemos mas ejemplos para probar nuestro compilador en la carpeta de ”pruebas”.