UNIVERSIDAD NACIONAL
AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Proyecto final: Compilador Jelly

ASIGNATURA

Compiladores

PROFESORA

Lourdes del Carmen González Huesca

AYUDANTES

Fausto David Hernández Jasso
Juan Alfonso Garduño Solís
Juan Pablo Yamamoto Zazueta

ALUMNOS

Carlos Emilio Castañon Maldonado
Dafne Bonilla Reyes
José Camilo García Ponce
Jorge Daniel Velasco García

Índice

1. Introducción

Mediante este reporte se plantea el desarrollo del proyecto final de la materia de compiladores cuyo objetivo será mostrar de forma descriptiva los pasos seguidos para desarrollar las extensiones del *compilador* que se ha estado llevando a cabo en prácticas anteriores tales como la gramática del lenguaje `Jelly`, agregando traducciones de ASA a Java, algunos ejercicios de prueba y las instrucciones para ejecutarlo.

2. Desarrollo

Los elementos fundamentales y que son bases para el proyecto son en gran medida las 6 prácticas realizadas a lo largo del curso, las cuales incluyen el desarrollo de los siguientes componentes necesarios para el compilador:

- Analizador léxico
- Sintaxis abstracta
- Tabla de símbolos
- Analizador sintáctico
- Sistema de tipos

Después de esto, agregamos el código necesario para realizar los ejercicios del proyecto, que serán explicados en las siguientes secciones.

3. Gramática

```
Programa -> FuncionMain
          | FuncionMain Procedimientos
          | VarGlob FuncionMain
          | VarGlob FuncionMain Procedimientos

VarGlob -> Asignaciones

Asignaciones -> AsignacionEspecial
              | AsignacionEspecial Asignaciones

FuncionMain -> "main" "{" Lineas "}"

Procedimientos -> Metodo
                | Funcion
                | Metodo Procedimientos
                | Funcion Procedimientos

Funcion -> Identificador "(" ")" "{" Lineas "}"
         | Identificador "(" Parametros ")" "{" Lineas "}"

Metodo -> Identificador "(" ")" ":" Tipo "{" "return" Expresion "}"
        | Identificador "(" ")" ":" Tipo "{" Lineas "return" Expresion "}"
        | Identificador "(" Parametros ")" ":" Tipo "{" "return" Expresion "}"
        | Identificador "(" Parametros ")" ":" Tipo "{" Lineas "return" Expresion "}"

Parametros -> DeclaracionNormal
            | DeclaracionNormal "," Parametros

If -> "if" Expresion "{" Lineas "}"
    | "if" "(" Expresion ")" Linea "else" Linea
    | "if" "(" Expresion ")" "{" Lineas "}" "else" "{" Lineas "}"

While -> "while" Expresion "{" Lineas"}"

Print -> "println" "(" Expresion ")"

Lineas -> Linea
        | Linea Lineas

Linea -> While
       | If
       | Declaracion
       | Print
       | Asignacion
       | Expresion

Declaracion -> DeclaracionNormal
             | DeclaracionMultiple

DeclaracionNormal -> Identificador ":" Tipo

DeclaracionMultiple -> Tipo Identificadores
```

```
Asignacion -> AsignacionNormal
            | AsignacionEspecial
            | DeclaracionMultiple

AsignacionNormal -> Identificador "=" Expresion
                  | ArrPos "=" Expresion

AsignacionEspecial -> AsignacionTipo
                    | AsignacionArr

AsignacionTipo -> Identificador ":" Tipo "=" Expresion

AsignacionArr -> Identificador ":" Tipo "=" "{" Expresiones "}"

AsignacionMultiple -> AsignacionVarias "=" Expresion

AsignacionVarias -> Identificador "=" Identificador
                  | Identificador "=" AsignacionVarias

Expresiones -> Expresion
            | Expresion "," Expresiones

Expresion -> "(" Expresion ")"
            | Longitud
            | Identificador
            | ArrPos
            | Numero
            | Booleano
            | Cadena
            | UsoProc
            | Expresion "?" Expresion ":" Expresion
            | "!" Expresion
            | Expresion "++"
            | Expresion "--"
            | Expresion "= =" Expresion
            | Expresion "! =" Expresion
            | Expresion "> =" Expresion
            | Expresion ">" Expresion
            | Expresion "< =" Expresion
            | Expresion "<" Expresion
            | Expresion "|" Expresion
            | Expresion "&" Expresion
            | Expresion "+=" Expresion
            | Expresion "-=" Expresion
            | Expresion "+" Expresion
            | Expresion "-" Expresion
            | Expresion "*" Expresion
            | Expresion "/" Expresion
            | Expresion "%" Expresion

Longitud -> "len" "(" Identificador ")"
```

```
UsoProc -> Identificador "(" ")"
         | Identificador "(" Expresiones ")"

Identificadores -> Identificador
                 | Identificador "," Identificadores

Identificador -> "[a-z][a-zA-Z0-9_]*"

ArrPos -> Identificador "[" Expresion "]"

Tipo -> "int"
      | "bool"
      | "string"
      | "int" "[" "]"
      | "bool" "[" "]"
      | "string" "[" "]"

Numero -> "[0-9]+"

Booleano -> "True"
          | "False"

Cadena -> "\" any-string [^\"] any-string "\"
```

4. Instrucciones

A continuación, explicaremos las instrucciones de uso de nuestro proyecto:

1. Creamos el ejecutable:

```
batman@equipoAzul:🦇 /Proyecto$ chmod +x script.sh
```

2. Ejecutamos:

```
batman@equipoAzul:🦇 /Proyecto$ ./script.sh archivo.jly
```

➢ Ejemplo de uso:

```
batman@equipoAzul:🦇 /Proyecto$ chmod +x script.sh
batman@equipoAzul:🦇 /Proyecto$ ./script.sh ejemplos/in.jly
 4 shift/reduce conflicts
 Archivo generado: In.java
```

5. Ejercicios

5.1. Traducción del Árbol de Sintaxis Abstracta a Java

Este ejercicio fue resuelto en el archivo `transform.rkt` del proyecto usando la función `traducir-programa`:

```
(define (traducir-programa ir clase) ... )
```

La cual recibe una representación intermedia del programa en el lenguaje Jelly hecho en `nanopass` y el nombre de la clase en `Java`. Al final, devuelve el texto de como sería en `Java`, este texto lo obtiene mediante otras funciones como `traducir-metodo`, entre otras.

Así, `traducir-programa` primero revisa que el archivo recibido si tenga la extensión `.jly` como en la función `validar-archivo`, después generamos el nombre de la clase de `Java` quitando la terminación `.jly`, volviendo la primera letra del archivo en mayúscula y quitando la dirección del archivo. Además, usamos `compilar-archivo` para guardar el archivo y generar todo el proceso:

```
(define (compilar-archivo nombre) ... )
```

Luego, con la función `ranme-var-archivo`, obtenemos el árbol sintáctico del programa (con las variables ya renombradas):

```
(define (ranme-var-archivo nombre) ... )
```

Después, con esta función obtenemos la tabla de símbolos del archivo:

```
(define (symbol-table-sin ir) ... )
```

A continuación, usando `type-check` función, revisamos que tenga un tipado válido mediante el árbol sintáctico y la tabla de símbolos.

```
(define (type-check ir) ... )
```

Posteriormente, usamos `traducir-programa` para tener la versión del texto en formato `Java` pasándole el nuevo nombre del programa y por último, escribimos en un nuevo archivo el código de `Java`:

```
    (call-with-output-file (string-append nuevo-nombre ".java")
        (lambda (out)
            (display prueba4 out))
        #:exists 'replace)
```

Por lo tanto, es así como funciona nuestra traducción.

5.1.1. Ejemplo de Uso

```
batman@equipoAzul:~/Proyecto$ chmod +x script.sh
batman@equipoAzul:~/Proyecto$ ./script.sh ejemplos/in.jly
 4 shift/reduce conflicts
 Archivo generado: In.java
```

- **Archivo original en `Jelly`:**

```
// Comentario 1

main{
    zzzz:int = 0
    i:int = zzzz++
    zzzz += 1
    zzz:int = 1
    r:int = gdc(i,zzz)
}

{- Comentario 2
   Comentario 2 -}

gdc(var1:int, var2:int): int{
    while var1 != 0 {
        if (var1 < var2) var2 = var2 - var1
        else var1 = var1 - var2
    }
    b:int = 10
    return b
}

sort(a:int[]){
    i:int = 0
    n:int = len(a)

    while i < n {
        j:int = i
        while j > 0 {
            if a[j-1] > a[j] {
                swap:int = a[j]
                a[j] = a[j-1]
                a[j-1] = swap
            }
            j--
        }
        i++
    }
}
```

- **Archivo generado en `Java`:**

```
public class In {

public static void main(String[] args) {
int variable_bonita_2 = 0;
int variable_bonita_1 = variable_bonita_2 = variable_bonita_2 + 1;
variable_bonita_2 = variable_bonita_2 + 1;
int variable_bonita_0 = 1;
int variable_bonita_3 = gdc(variable_bonita_1, variable_bonita_0);
}
```

```
public static int gdc(int variable_bonita_4, int variable_bonita_6) {
while (variable_bonita_4 != 0) {
if (variable_bonita_4 < variable_bonita_6) {
variable_bonita_6 = variable_bonita_6 - variable_bonita_4;
} else {
variable_bonita_4 = variable_bonita_4 - variable_bonita_6;
};
};
int variable_bonita_5 = 10;
return variable_bonita_5;
}

public static void sort(int[] variable_bonita_10) {
int variable_bonita_9 = 0;
int variable_bonita_7 = variable_bonita_10.length;
while (variable_bonita_9 < variable_bonita_7) {
int variable_bonita_11 = variable_bonita_9;
while (variable_bonita_11 > 0) {
if (variable_bonita_10[variable_bonita_11 - 1] > variable_bonita_10[variable_bonita_11]) {
int variable_bonita_8 = variable_bonita_10[variable_bonita_11];
variable_bonita_10[variable_bonita_11] = variable_bonita_10[variable_bonita_11 - 1];
variable_bonita_10[variable_bonita_11 - 1] = variable_bonita_8;
} else {
};
variable_bonita_11 = variable_bonita_11 - 1;
};
variable_bonita_9 = variable_bonita_9 + 1;
};
}

}
```

5.2. Script de Jelly desde la Línea de Comandos

Este ejercicio fue resuelto en el archivo `script.sh` del proyecto, cuyo contenido es:

```
#!/bin/bash

# Verificar si se ha proporcionado algún argumento
if [ $# -eq 0 ]; then
    echo "Uso: $0 <ruta-al-archivo.jly>"
    exit 1
fi
# Verificar si se han proporcionado múltiples argumentos
if [ $# -gt 1 ]; then
    echo "Uso: $0 <ruta-al-archivo.jly>"
    echo "Se proporcionaron multiples argumentos.
    Solo se permite un archivo .jly."
    exit 1
fi
# Ejecutar el script Racket con el argumento proporcionado
racket transform.rkt "$1"
```

A continuación explicaremos como funciona. Primero escribimos:

```
#!/bin/bash
```

Lo anterior se encarga de indicar que el script debe ser ejecutado usando bash. Después tenemos:

```
if [ $# -eq 0 ]; then
    echo "Uso: $0 <ruta-al-archivo.jly>"
    exit 1
fi
```

Esto se encarga de comprobar si se proporcionó algún argumento y en caso de que no se hayan pasado argumentos, lanza un mensaje de como debe usarse y termina ejecución con `exit 1`. Posteriormente usamos el siguiente `if`:

```
if [ $# -gt 1 ]; then
    echo "Uso: $0 <ruta-al-archivo.jly>"
    echo "Se proporcionaron multiples argumentos.
    Solo se permite un archivo .jly."
    exit 1
fi
```

El `if` se encarga de comprobar si se ha introducido más de un argumento, en caso de que se pasen más argumentos de los establecidos, desplegamos un mensaje con el uso del script y terminamos ejecución.

```
racket transform.rkt "$1"
```

La línea anterior se encarga de indicar que se ejecute el archivo `transform.rkt` con el archivo `.jly` que le pasamos como parámetro. Nótese como es que la parte encargada de lo anterior dentro de `transform.rkt`, es:

```
(command-line
    #:program "transform.rkt"
    #:args (nombre-archivo)
    "Compilar archivo especificado"
    (compilar-archivo nombre-archivo))
```

La cual realiza la comunicación entre el `bash` y las operaciones a realizar por `transform.rkt`.

5.2.1. Ejemplo de Uso

```
batman@equipoAzul:/Proyecto$ chmod +x script.sh
batman@equipoAzul:/Proyecto$ ./script.sh ejemplos/dos.jly
 4 shift/reduce conflicts
 Archivo generado: dos.java
```

- **Archivo original en Jelly:**

```
resultado:bool = True

main {
    b:bool = impar(60)
}

// Funcion impar xd
impar(num:int):bool{
    if (num%2 ==0) {resultado = False}
    return resultado
}
```

- **Archivo generado en Java:**

```
public class Dos {

    public static boolean variable_bonita_0 = true;

    public static void main(String[] args) {
        boolean variable_bonita_1 = impar(60);
    }

    public static boolean impar(int variable_bonita_3) {
        if (variable_bonita_3 % 2 == 0) {
        variable_bonita_0 = true;
        } else {
        };
        return variable_bonita_0;
    }
}
```

5.3. Extensión del Lenguaje para Permitir Cadenas al Igual que `println(e)`

Este ejercicio fue resuelto modificando nuestras implementaciones en los archivos:

- ✪ lexer.rkt
- ✪ parser.rkt
- ✪ symbolTable-renameVar.rkt
- ✪ syntaxtTree.rkt
- ✪ types.rkt
- ✪ transform.rkt

En el archivo lexer.rkt agregamos su token (PRINT) dentro de:

```
(define-empty-tokens vacios ( ...
                              RETURN    ; palabra reservada return
                              PRINT     ; palabra reservada print
                              LENGTH    ; palabra reservada length
                               ... ))   ; fin de archivo
```

Además de que agregamos su palabra reservada (println) dentro de lo siguiente:

```
(define jelly-lex
    (lexer
        ...
        ["return"  (token-RETURN)] ; Retorno
        ["println"   (token-PRINT)] ; Impresion
        ["len"  (token-LENGTH)] ; Longitud de un arreglo
        ... ))
```

En el archivo parser.rkt agregamos su estructura (printj):

```
...
(define-struct ifj (condicion lineasthen lineaselse) #:transparent)
(define-struct printj (expresion) #:transparent)
(define-struct declaracion (nombre tipo) #:transparent)
...
```

Además de agregarlo al parser:

```
(define jelly-parser
    (parser
            ...
            [printj
                [(PRINT OPENP expresion CLOSEP) (printj $3)]]
            [lineas
                [(linea) (list $1)]
                [(linea lineas) (list* $1 $2)]]
            [linea
                [(while) $1]
                [(ifj) $1]
                [(printj) $1]
                [(declaracion) $1]
                [(asignacion) $1]
                [(expresion) $1]]
                ... ))
```

Después, en el archivo `symbolTable-renameVar.rkt` lo agregamos dentro de la definición del lenguaje de Jelly:

```
(define-language jelly
    ...
    (Linea (ln)
        (whilejly e (ln* ...)) ; while expresion (linea* ...)
        (ifjly e (ln1* ...) (ln2* ...)) ; if expresion (linea* ...) (linea* ...)
        (printjly e) ; print expresion
        dec ; declaracion
        e ; expresion
    )
    ...
)
```

Posteriormente, lo agregamos en la siguiente función para poder tener su variable de una línea:

```
(define (vars-linea ir vs)
    (nanopass-case (jelly Linea) ir
        ...
        [(printjly ,[vars-expr : e vs -> e1]) vs]
        ...))
```

Una vez hecho lo anterior procedemos a agregarlo en nuestro método de renombramiento de variables.

```
(define-pass ranme-var : jelly (ir) -> jelly ()
    ...
    (Linea : Linea (ir h) -> Linea ()
        ...
        [(printjly ,e)
            (let ([e-1 (Expr e h)])
                `(printjly ,e-1))]
        ...
    ...))
```

Luego, lo agregamos en nuestro método de obtención de la tabla de símbolos de una sola línea:

```
(define (symbol-table-linea ir st)
    (nanopass-case (jelly Linea) ir
        ...
        [(printjly ,[symbol-table-expr : e st -> e1]) st]
        ...
    ...))
```

A continuación, en el archivo `syntaxtTree.rkt` lo agregamos dentro del árbol de sintaxis:

```
(define (syntax-tree prog)
    (match prog
        ...
        [(printj expresion) (string-append "(printjly " (syntax-tree expresion) ")")]
        ...
    ...))
```

Subsecuentemente, en el archivo types.rkt lo agregamos dentro del verificador de tipos de línea:

```
(define (type-check-linea ir st)
    (nanopass-case (jelly Linea) ir
        ...
        [(printjly ,[type-check-expr : e st -> t1])
            (if (or (equal? t1 'int) (equal? t1 'bool) (equal? t1 'string))
            'UNIT (error
            "Error en type-check-linea-print con " t1))]
        ...
    ...))
```

Finalmente, en el archivo transform.rkt lo agregamos dentro de la traducción de una línea:

```
(define (traducir-linea ir)
    (nanopass-case (jelly Linea) ir
        ...
        [(printjly ,[traducir-expr : e -> e1])
            (string-append "System.out.println(" e1 ")")]
        ...
    ...))
```

Para agregar el tipo string, se realizaron algunos cambios (se fueron trabajando durante las practicas), estas modificaciones no fueron muy complicadas, debido a que solo es tener otro tipo (como int y bool)

- lexer.rkt: En este archivo agregamos la forma para poder tener un token que contenga cadenas y además la forma de reconocer cadenas, que son caracteres entre comillas. Además agregamos tokens para el tipo string y arreglo de strings.
- parser.rkt: En este archivo modificamos la gramática para usar los nuevos tokens y tomar las cadenas como expresiones.
- syntaxtTree.rkt: En este archivo las modificaciones son muy pocas, solo poner las lineas de código necesario para que funcione como los otros tipos.
- symbolTable-renameVar.rkt: Aquí modificamos el lenguaje en nanopass, para que en las constantes se incluyan a las cadenas y que tipos tenga a string y stringarr.
- types.rkt: Modificamos como se revisa el tipo de algunas relacionadas con arreglos, con el fin de aceptar arreglos de cadenas. No hubo tantas modificaciones debido a que las cadenas no tiene muchas operaciones.
- transform.rkt: Solo modificamos como se traducen las constantes.

5.3.1. Ejemplo de Uso

```
batman@equipoAzul:~/Proyecto$ chmod +x script.sh
batman@equipoAzul:~/Proyecto$ ./script.sh ejemplos/string1.jly
 4 shift/reduce conflicts
 Archivo generado: string1.java
```

- **Archivo original en `Jelly`:**

```
yeah:string = "ave4"
main{
    ave_g4:int[] = {1,3,4,5}
    a:string = "pato"
    b_2:string[] = {"ave", "ave2", "ave3", yeah}
    dummyVar:bool
    println(a)
}
```

- **Archivo generado en `Java`:**

```
public class String1 {

public static String variable_bonita_0 = "ave4";

public static void main(String[] args) {
int[] variable_bonita_2 = {1, 3, 4, 5};
String variable_bonita_1 = "pato";
String[] variable_bonita_3 = {"ave", "ave2", "ave3", variable_bonita_0};
boolean variable_bonita_5;
System.out.println(variable_bonita_1);
}

}
```

5.4. Definición y Uso de Variables Globales

Este ejercicio fue resuelto (a lo largo de las prácticas) modificando los archivos:

- `parser.rkt`
- `symbolTable-renameVar.rkt`
- `transform.rkt`
- `syntaxtTree.rkt`
- `types.rkt`

- `parser.rkt:` En este archivo modificamos la gramática para que ahora un programa tenga la posibilidad de tener variables globales, la cuales son asignaciones.

  Nota: Nuestras variables globales solo son asignaciones, no permitimos declaraciones de arreglos, esto debido a que en el proceso de traducción a código de `Java` se generaron algunas complicaciones por la manera en que funcionan los arreglos en `Java`.

- `syntaxtTree.rkt:` En este archivo hicimos pequeñas modificaciones a como se muestra el árbol sintáctico del programa.

- `symbolTable-renameVar.rkt:` Este archivo fue el que más modificaciones tuvo, esto debido a que al generar las variables de un programa primero tenemos que obtener las variables de las variables globales, luego estos conjuntos pasárselos al `main` y procedimientos para que agreguen sus variables, en la parte de renombrar no hubo cambios, ya que todo se resolvió con el cambio anterior.

- `types.rkt:` Aquí solo revisamos que las variables globales tengan tipado correcto.

- `transform.rkt:` En este archivo modificamos que al traducir las variables globales se traducen como una lista de asignaciones, solo que agregamos el prefijo `public static`.

5.4.1. Ejemplo de Uso

```
batman@equipoAzul:🦇 /Proyecto$ ./script.sh ejemplos/b.jly
 4 shift/reduce conflicts
 Archivo generado: B.java
```

- **Archivo original en `Jelly`:**

```
numero:int = 99
c:bool = True
numerito:int[] = {1,2}

main{
    println("algo")
    println(c)
    println(1+3)
    cadena:string = "pato"
    aux:int = 77
    aux = 78
    z:int[]
    p:int
    p = 1
    q:int = 0
    q:int = 1
    y:int[] = {1,2,3}
    z:int[] = {1,2}
    y:int[] = {1,4+1}
    g:int = y[0]
    fact(1, !True)
    j:int = True ? 1 : 0
    g = len(y)
    cadena:string = "pato2"
}
// metodo
fact(num:int, b:bool):int{
    algo:int = 0
    if (algo < 1) {
        algo = 1+1
    } else {
        algo = 2+2
    }
    algo = 2+2
    aux:int = 14
    aux = 15
    return numero
}

a(num:int){
    while (num > 3) {
        if (1 < 2) num = 3 - 1
        else num = 4 - 1
    }
}

d(){
    if 1>2 {a:int = 2+2}
}
```

✪ **Archivo generado en Java:**

```java
public class B {
    public static int variable_bonita_10 = 99;
    public static boolean variable_bonita_8 = true;
    public static int[] variable_bonita_9 = {1, 2};
    public static void main(String[] args) {
        System.out.println("algo");
        System.out.println(variable_bonita_8);
        System.out.println(1 + 3);
        String variable_bonita_19 = "pato";
        int variable_bonita_12 = 77;
        variable_bonita_12 = 78;
        int[] variable_bonita_13;
        int variable_bonita_15;
        variable_bonita_15 = 1;
        int variable_bonita_11 = 0;
        variable_bonita_11 = 1;
        int[] variable_bonita_17 = {1, 2, 3};
        variable_bonita_13 = new int[]{1, 2};
        variable_bonita_17 = new int[]{1, 4 + 1};
        int variable_bonita_18 = variable_bonita_17[0];
        fact(1, !true);
        int variable_bonita_14 = true ? 1 : 0;
        variable_bonita_18 = variable_bonita_17.length;
        variable_bonita_19 = "pato2"; }
    public static int fact(int variable_bonita_24, boolean variable_bonita_23) {
        int variable_bonita_20 = 0;
        if (variable_bonita_20 < 1) {
            variable_bonita_20 = 1 + 1;
        } else {
            variable_bonita_20 = 2 + 2;
        };
        variable_bonita_20 = 2 + 2;
        int variable_bonita_21 = 14;
        variable_bonita_21 = 15;
        return variable_bonita_10;
    }
    public static void a(int variable_bonita_25) {
        while (variable_bonita_25 > 3) {
            if (1 < 2) {
                variable_bonita_25 = 3 - 1;
            } else {
                variable_bonita_25 = 4 - 1;
            };
        };
    }
    public static void d() {
        if (1 > 2) {
            int variable_bonita_26 = 2 + 2;
        } else {
        };
    }
}
```

6. Archivos

Los archivos que conforman a nuestro compilador son:

1. `lexer.rkt:` Definición de tokens(lexemas) del lenguaje.
2. `parser.rkt:` Definición de la gramática del lenguaje y parseo del código entrante.
3. `types.rkt:` Realiza la verificación de las variables que tengan el tipado correcto.
4. `syntaxtTree.rkt:` Construye el árbol de sintaxis.
5. `symbolTable-renameVar.rkt:` Realiza el renombramiento de variables y construir la tabla de símbolos con el respectivo cambio.
6. `transform.rkt:` Traduce el árbol de sintaxis abstracta a código de Java, el cual es guardado en un archivo.