



UNIVERSIDAD GALILEO

RISC-V

COOL Code Generation

9 de octubre de 2018

La idea central de este documento es explicar cada una de las partes que salen durante la generación de código en la última fase de su proyecto. Intentaremos explicar a gran detalle la función de cada una de estas secciones que deben implementar.

Vamos a compilar este pequeño archivo llamado *main.cl*, utilizando *coolc-rv* que instalaron con el material de RISC-V.

```
main.cl
class Main {
  out: IO <- new IO;

  main(): Object {
    out.out_int(2 + 3)
  };
};

class A {
};

class B {
};
```

Si compilamos el archivo tal y como se presenta en la figura anterior, se generará un archivo llamado *main.s* que si lo abren verán en las primeras líneas lo siguiente:

```
Global Data
# start of generated code
.data
.align 2
.globl class_nameTab
.globl Main_protObj
.globl Int_protObj
.globl String_protObj
.globl bool_const0
.globl bool_const1
.globl _int_tag
.globl _bool_tag
.globl _string_tag
```

Esta primera parte representa los símbolos globales que necesita el archivo ***trap-handler.s***, que es el código base que implementa el runtime de **COOL** para poder funcionar correctamente, es algo que ya está hecho por ustedes y lo pueden ir a revisar en ***CgenClassTable.java*** y específicamente el método ***codeGlobalData()***.

Las siguientes 6 líneas de código que genera también el método antes mencionado representan el *class tag* de tres clases básicas que ya conocen: **Int**, **Bool**, **String**.

Class tag of basic classes

```
_int_tag :
  .word    2
_bool_tag :
  .word    3
_string_tag :
  .word    4
```

Ustedes tienen que implementar alguna forma de asignar los *tags*, de tal manera que se les facilite implementar el codegen de cada ***Expression*** en ***cool-tree.java*** (esto sobre todo en *typcase* que utiliza bastante los *tags*). En este caso el compilador **coolc-rv** le asignó a **Int**: 2, a **Bool**: 3 y a **String**: 4.

Las siguientes 3 líneas de código generado representan una opción que el manejador de memoria utilizará (si está habilitada) para verificar que el estado del *heap* sea consistente cada vez que sea crea un objeto o se necesite reservar más espacio de memoria.

Memory Manager test no habilitado

```
.globl  _MemMgr_TEST
_MemMgr_TEST :
  .word    0
```

Si ustedes le pasan la opción **-t** al compilador **coolc-rv** habilitarán esta opción y se generará algo como lo siguiente:

Memory Manager test habilitado

```
.globl  _MemMgr_TEST
_MemMgr_TEST :
  .word    1
```

Esto también ya está hecho por ustedes y lo pueden encontrar también en ***CgenClassTable.java*** y específicamente en el método llamado ***codeMemoryManager()***.

Las siguientes líneas ya es algo en donde ustedes van a comenzar a hacer algunos *tweaks* para que se vea tal como el output generado por el compilador **coolc-rv**:

```
String constants
.word    -1
str_const13:
.word    4
.word    5
.word    String_dispTab
.word    int_const2
.byte    0
.align   2
.word    -1
str_const12:
.word    4
.word    5
.word    String_dispTab
.word    int_const3
.ascii   "B"
.byte    0
...
```

Este código generado para este caso que estamos analizando indica que hay 14 constantes String (del 0 al 13). Estas se generan automáticamente en el método que está en **CgenClassTable.java** llamado **codeConstants()**. Si vemos detalladamente cada uno de las líneas después del label *strconst_13* por ejemplo, estas representan un prototipo de objeto que explicaremos más adelante, la primera línea **.word 4** es el class tag, lo que indica que la clase **String** se le asignó el tag 4 (coincide perfectamente con el tag declarado más arriba donde estaba **_string_tag**), ustedes tienen que ver cómo asignar estos tags como se mencionó anteriormente porque cada clase tiene que tener un tag diferente, a ustedes les aparecerá **.word 0** si utilizan **./mycoolc** porque todavía no han asignado estos *tags* que mencionamos.

La tercera línea **.word String_dispTab** es un puntero a la tabla dispatch de la clase **String**, los dispatch tables los explicaremos más adelante. Si ustedes nuevamente utilizan **./mycoolc** les aparecerá solamente **.word** porque ustedes tienen que ir a modificar **StringSymbol** para lograr que aparezca la referencia hacia la dispatch table de String como se vió en el ejemplo.

Luego de las constantes **String** vienen las constantes **Int** que se ven algo así:

```
Int constants
.word    -1
int_const9:
.word    2
.word    4
.word    Int_dispTab
.word    7
.word    -1
int_const8:
.word    2
.word    4
.word    Int_dispTab
.word    13
.word    -1
...
```

Al igual que las constantes **String**, ustedes en la tercera línea después de la etiqueta `int_constx` tienen que modificar su código para que puedan tener la referencia `.word Int_dispTab ./mycoolc.` (*hint*: IntSymbol)

Las siguientes líneas representan las constantes **Bool** que son únicamente 2 *false* y *true* y se ven de la siguiente forma:

```
Bool constants
.word    -1
bool_const0:
.word    3
.word    4
.word    Bool_dispTab
.word    0
.word    -1
bool_const1:
.word    3
.word    4
.word    Bool_dispTab
.word    1
```

Y sí aquí también tiene que lograr que aparezca en la tercera línea después de `bool_constx` la referencia hacia el dispatch table de Bool. (*hint*: BoolConst).

A continuación verán el class name table, que aparece en las siguientes líneas de código generado:

Class name table	
class_nameTab :	
.word	str_const5
.word	str_const6
.word	str_const7
.word	str_const8
.word	str_const9
.word	str_const10
.word	str_const11
.word	str_const12

Esta es la tabla de nombre de clases, si revisan el código compilado, verán que **str_const5** corresponde a la clase **Object**, **str_const6** corresponde a la clase **IO**, y así sucesivamente, hasta llegar a **str_const12**, que corresponde a la clase **B**. El orden en que coloquen estas clases es muy importante para los siguientes pasos. Prácticamente el orden es el orden en el que asignaron los class tags.

Después de la tabla de nombre de clases, debemos implementar la tabla de objetos

Object table	
class_objTab :	
.word	Object_protObj
.word	Object_init
.word	IO_protObj
.word	IO_init
.word	Int_protObj
.word	Int_init
.word	Bool_protObj
.word	Bool_init
.word	String_protObj
.word	String_init
.word	Main_protObj
.word	Main_init
.word	A_protObj
.word	A_init
.word	B_protObj
.word	B_init

No hay mucho que explicar de esta tabla, cada vez que se declara un objeto utilizamos la etiqueta *protObj*, y cada vez que se instancia con *new*, utilizamos la etiqueta *init*, sin embargo, es muy importante que escriban estas etiquetas con el mismo orden con el que escribieron las etiquetas en la tabla de nombre de clase (primero **Object**, luego **IO**, así).

La siguiente sección que deben implementar, son las dispatch tables de cada una de las clases. Aquí también es muy importante que las escriban en el mismo orden que antes. Veamos las generadas del ejemplo.

Object dispatch table

```
Object_dispTab :
  .word    Object . abort
  .word    Object . type_name
  .word    Object . copy
```

IO dispatch table

```
IO_dispTab :
  .word    Object . abort
  .word    Object . type_name
  .word    Object . copy
  .word    IO . out_string
  .word    IO . out_int
  .word    IO . in_string
  .word    IO . in_int
```

Int dispatch table

```
Int_dispTab :
  .word    Object . abort
  .word    Object . type_name
  .word    Object . copy
```

Bool dispatch table

```
Bool_dispTab :
  .word    Object . abort
  .word    Object . type_name
  .word    Object . copy
```

String dispatch table

```
String_dispTab :
  .word    Object . abort
  .word    Object . type_name
  .word    Object . copy
  .word    String . length
  .word    String . concat
  .word    String . substr
```

Main dispatch table

```
Main_dispTab :
  .word    Object . abort
  .word    Object . type_name
  .word    Object . copy
  .word    Main . main
```

A dispatch table

```
A_dispTab :
  .word    Object . abort
  .word    Object . type_name
  .word    Object . copy
```

B dispatch table

```
B_dispTab :
  .word    Object . abort
  .word    Object . type_name
  .word    Object . copy
```

Revisando las tablas anteriores, pueden ver que el orden de cada uno de los métodos es el mismo, eso es muy importante también. Vean también que, por ejemplo, la clase **String** tiene al inicio los métodos heredados de la clase **Object**, y tienen el prefijo **Object**: **Object.abort**, **Object.type_name**, etc. Si una clase sobrescribe un método de su padre, por ejemplo, si **Main** sobrescribiera el método **copy** de la clase **Object**, entonces, en su dispatch table, debería estar, en vez de **Object.copy**, **Main.copy**, en la misma posición, y si alguna clase heredara de **Main**, pero no sobrescribiera el método **copy**, debería tener en su dispatch table el método **Main.copy**.

Veamos un ejemplo en donde se pueda ver claramente lo que se mencionó anteriormente.

main.cl

```
class Main {
  out: IO <- new IO;

  main(): Object {
    out.out_int(2 + 3)
  };
};

class A inherits Main {
  main() : Object {
    out.out_int(3 + 4)
  };
};

class B inherits A { };
```

Estas serian las dispatch tables de las clases **Main**, **A** y **B**:

Main A and B dispatch tables

```
Main_dispTab:
  .word    Object.abort
  .word    Object.type_name
  .word    Object.copy
  .word    Main.main
A_dispTab:
  .word    Object.abort
  .word    Object.type_name
  .word    Object.copy
  .word    A.main
B_dispTab:
  .word    Object.abort
  .word    Object.type_name
  .word    Object.copy
  .word    A.main
```

Noten como la clase **A** tiene el método **A.main** porque sobrescribe el método definido en **Main**, y como **B** hereda de **A**, pero no sobrescribe el método **main**, en su dispatch table escribimos **A.main**. Estos métodos deben estar siempre en el mismo orden; el método **main**, en este caso, debe ser siempre la cuarta entrada en las dispatch tables de todas las clases que hereden de **A**.

Ya vamos cerca de generar código, pero aún no llegamos, lo que toca es escribir el código de los **protObj**, es decir los prototipos de objeto. Imaginen que tenemos el siguiente código en **COOL**:

```
main.cl
class Main {
    out: IO ← new IO;

    main(): Object {
        out.out_int(2 + 3)
    };
};

class A inherits Main {
    out2: String;

    main() : Object {
        out.out_int(3 + 4)
    };
};

class B inherits A {
    out3: Int ← 4;
    out4: IO;
};
```

La clase **Main** tiene un campo, la clase **A** tiene dos campos (**out**, que hereda de **Main**, y **out2**), y la clase **B** tiene cuatro campos. Ahora, tomando esto en cuenta, veamos la codificación de sus **protObj**.

Object protObj

```
.word    -1
Object_protObj:
.word    0 # posicion en class_nameTab
.word    3 # cantidad de campos
.word    Object_dispTab # dispatch table
```

Ignoremos el `.word -1` (aún así, deben agregarlo). Enfoquemonos en `.word 0`, este 0, representa la posición en la tabla de nombres de clase que definieron anteriormente, o simplemente el class tag. Recuerden que **Object** estaba en la posición 0. El siguiente valor; `.word 3` representa la cantidad de campos. La clase **Object**, como tal, no tiene campos, pero debemos guardar de ella su nombre, la cantidad de campos (si, como un campo), y su dispatch table, así que por eso debemos poner un 3 en ese lugar. Luego, la etiqueta de su dispatch table, y listo.

Main protObj

```
.word    -1
Main_protObj:
.word    5 # posicion en class_nameTab
.word    4 # cantidad de campos
.word    Main_dispTab # dispatch table
.word    0 # valor inicial de out
```

El 5, representa la posición en la tabla de nombres de clases de la clase **Main**, el 4 representa la cantidad de campos, aparte de los 3 que ya mencionamos, la clase **Main** tiene un campo out. Después, tenemos la dispatch table de la clase **Main**, y finalmente, el `.word 0` representa el valor inicial del campo out. Si los campos pertenecen a las clases **Int**, **Bool** o **String**, los inicializaremos como la constante 0 para la clase **Int**, el string vacío para la clase **String**, y falso para la clase **Bool**. Si el campo no pertenece a ninguna de estas clases, lo inicializaremos como 0, para reponsetar que es *void*.

A protObj

```
.word    -1
A_protObj:
.word    6
.word    5
.word    A_dispTab
.word    0
.word    str_const13
```

El 6, como ya sabemos, representa la posición en la tabla de nombres de clases, el 5, representa la cantidad de campos que tiene la clase **A**, luego tenemos la dispatch table de la clase, el 0 que sigue, es el valor inicial de la variable out, y vemos que después tenemos `str_const13`, que, si buscamos más arriba, hace referencia al string vacío, esto es porque out2 es de tipo **String**.

B protObj	
.word	-1
B_protObj:	
.word	7
.word	7
.word	B_dispTab
.word	0
.word	str_const13
.word	int_const3
.word	0

Para la clase **B** sigue siendo lo mismo; el primer 7 representa la posición en la tabla de nombres de clase, el segundo 7 representa la cantidad de campos (los 3 ya establecidos y los 4 que declaramos). Luego tenemos la dispatch table de la clase **B**, seguido de los valores iniciales de sus campos. Noten que out3 es de tipo **Int**, y si revisamos en la etiqueta `int_const3`, veremos que esta constante representa el valor 0.

Finalmente, luego de codificar todos los **protObj** (aquí solo vimos los ejemplos de algunos, ustedes deben codificarlos todos), ya podemos pegar el último fragmento de código que se genera automáticamente:

Global text	
.globl	heap_start
heap_start:	
.word	0
.text	
.globl	Main_init
.globl	Int_init
.globl	String_init
.globl	Bool_init
.globl	Main.main

Esta porción de código ya está hecha por ustedes y pueden ir a revisarlo en *Cgen-ClassTable.java* en el método *codeGlobalText()*.

Si se dan cuenta, aún no hemos empezado a codificar las instrucciones, toda esta parte es para preparar al simulador para inicializar objetos, y aquí podemos ver la complejidad de un lenguaje de este tipo, contra uno como C. Pero siguiendo con el tema, la última sección que deben codificar, antes de empezar a traducir instrucciones en **COOL** a *assembler*, son los **inits**.

Main init

```

Main_init:
    addi    sp    sp    -12
    sw      tp    12(sp)
    sw      s0    8(sp)
    sw      ra    4(sp)
    addi    tp    sp    4
    mv      s0    a0
    jal     Object_init
    la      a0    IO_protObj
    jal     Object.copy
    jal     IO_init
    sw      a0    12(s0)
    mv      a0    s0
    lw      tp    12(sp)
    lw      s0    8(sp)
    lw      ra    4(sp)
    addi    sp    sp    12
    jr      ra
  
```

Como usaremos *sp*, *tp*¹ y *s0*, debemos guardarlos en el *stack*. Esto se hace en las primeras 4 instrucciones de **Main_init**. En la siguiente instrucción, hacemos que *tp* quede en el top del *stack*, para saber donde esta nuestra referencia, ya que *sp* cambia constantemente. Y ahora, movemos a *a0* hacia *s0*, esto se hace porque en *a0* tenemos la dirección final del objeto, y se sobrescribiera, así que la pasamos a *s0* para preservarla. *jal Object_init* se hace porque debemos inicializar los campos que tiene **Object**, que es el padre directo de **Main**, antes de poder inicializar los campos propios de **Main**. Cuando regresa de *Object_init*, tenemos los campos inicializados por **Object** en *s0* (aunque también los tendremos en *a0*), y ya podemos inicializar los campos propios de **Main**. Carga a *a0* el *IO_protObj*, y salta a *Object.copy*

¹Se utiliza *tp* en vez de *fp* para el frame pointer, porque *fp* y *s0* en RISC-V son lo mismo.

para realizar una copia de él. `Object.copy` se encuentra en el *traphandler.s*, así que no se preocupen de eso. Luego, tenemos la copia en `a0`, pero como en `Main` inicializamos `out` como un new `IO`, debemos saltar a `IO_init`. Cuando regresa, lo guardamos en la posición 12 de `s0` (porque es el cuarto campo), y como no hay nada más que hacer, regresamos la dirección del objeto (`s0`) a `a0`, restauramos el stack, y regresamos. Noten que estos últimos pasos (`la a0 IO_protObj`, `jal Object.copy`, `jal IO_init`) representan hacer el codegen del nodo `new` en *cool-tree.java*

B init

```

B_init:
    addi    sp    sp    -12
    sw      tp    12(sp)
    sw      s0    8(sp)
    sw      ra    4(sp)
    addi    tp    sp    4
    mv      s0    a0
    jal     A_init
    la      a0    int_const2
    sw      a0    20(s0)
    mv      a0    s0
    lw      tp    12(sp)
    lw      s0    8(sp)
    lw      ra    4(sp)
    addi    sp    sp    12
    jr      ra

```

Para el init de **B** es algo parecido, guardamos `sp`, `tp` y `s0` al stack. En la siguiente instrucción, hacemos que `tp` quede en el top del stack, como en `Main_init` para saber donde esta nuestra referencia. Y ahora, movemos hacia `s0`, `a0`, para no perderla, y saltamos a `A_init` (padre directo de **B** es **A**) que inicializa los campos que **B** hereda de **A**. Cuando regresa de `A_init`, tenemos los campos inicializados por **A** en `s0` (y también los tendremos en `a0`). Los campos que inicializó **A** fueron la posición 0, 4, 8, 12 y 16, y ahora debemos inicializar el campo 20 y 24 (la clase **B** tiene 7 campos; los 4 declarados y los 3 que establecimos aquí). Cargamos a `a0` `int_const2`, que, si buscamos en las constantes, tiene como valor el 4 que se le asigna, y lo guardamos en la posición 20 de `s0` (porque es el sexto campo). Vean que el séptimo campo se declara, pero no se inicializa, así que no hacemos nada (no lo inicializamos), regresamos la dirección del objeto (`s0`) a `a0`, restauramos el stack, y regresamos.

Algo importante que tienen que tener claro es que el activation record de los *inits* tiene la siguiente forma:

locals
old tp
old s0
old ra

Cuadro 1: tp apunta a la posición de `old ra` que es el top del stack.

Luego de codificar todos los **inits**, finalmente pueden empezar a codificar cada uno de los métodos declarados (son parecidos a los inits) tomando en cuenta que el activation record es diferente porque ahora si se permiten parámetros:

formals
locals
old tp
old s0
old ra

Cuadro 2: tp apunta a la posición de `old ra` que es el top del stack.

¡Listo! han terminado su compilador. El equipo de CC4 espera que se hayan divertido en esta travesía que duró un semestre completo y sobre todo que hayan aprendido bastante, es un conocimiento poderoso (si lo absorbieron y no se engañaron a ustedes mismos, SÍ a ustedes mismos no a nosotros) que pocos allá afuera tienen...

The great successful men of the world have used their imaginations, they think ahead and create their mental picture, and then go to work materializing that picture in all its details, filling in here, adding a little there, altering this a bit and that bit, but steadily building, steadily building.

— Robert Collier