Cgen

Equipo CCIV

Ciencias de la Computacion IV, Universidad Galileo,Guatemala

16 de octubre de 2016

Resumen

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

Vamos a compilar este pequeño archivo llamado **main.cl**:

```
Class Main{
        out : IO <- new IO;
        main() : Object{
                out.out_int(2 + 3)
        };
};

Class A{

};

Class B{

};
```

Si lo compilamos con los archivos tal y como nos los dan, obtenemos el siguiente resultado en **main.s**:

```
# 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
```

Si mi conocimiento no me falla, estas son etiquetas globales que utiliza el simulador para funcionar correctamente. Se generan automaticamente, así que para esta parte no es necesario hacer nada.

La siguiente porciÓn de codigo que se genera es esta:

```
_int_tag:
        .word   0
_bool_tag:
        .word   0
_string_tag:
        .word   0
```

Aqui deben hacer una pequeña modificación: las tags deben tener un valor diferente para que el procesador pueda diferenciarlas. Algo como esto:

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

Seguido de mas código que se genera automaticamente, para el manejo de memoria. Aqui no debemos tocar nada:

```
        .globl  _MemMgr_INITIALIZER
_MemMgr_INITIALIZER:
        .word   _NoGC_Init
        .globl  _MemMgr_COLLECTOR
_MemMgr_COLLECTOR:
        .word   _NoGC_Collect
        .globl  _MemMgr_TEST
_MemMgr_TEST:
        .word   0
        .word   -1
```

Bajamos un poco mas y encontraremos ese código:

```
        .word   -1
str_const13:
        .word   0
        .word   5
        .word
        .word   int_const2
        .byte   0
        .align  2
        .word   -1
str_const12:
        .word   0
        .word   5
        .word
        .word   int_const3
        .ascii  "B"
        .byte   0
        .align  2
        .word   -1
str_const11:
        .word   0
        .word   5
        .word
        .word   int_const3
        .ascii  "A"
        .byte   0
        .align  2
```

Tenemos 13 constantes al parecer. Estas se generan automaticamente; veremos que representa cada una de estas lineas:

```
        .word   -1
str_const13:
        .word   0
        .word   5
        .word
        .word   int_const2
        .byte   0
        .align  2
```

No toquen la primer linea, La segunda linea es el numero de la constante. La tercera linea **.word 0** es el valor que le asignamos a la string_tag anteriormente, asi que en el código que ustedes generen, deben cambiar este 0 a un 4. La siguiente linea **.word 5** no la toquen, despues vean que solo dice **.word** y no tiene ningun valor. Aqui deben escribir la label de la **String_dispTab**. Las siguientes lineas tampoco deben tocarlas, asi que el código final debe ser el siguiente:

```
        .word   -1
str_const13:
        .word   4
        .word   5
        .word   String_dispTab
        .word   int_const2
        .byte   0
        .align  2
```

Noten que la primer constante: **str_const13** no tiene una seccion **.ascii**, y las demas si. Esto representa que **string_const13** es el **String** vacio.

Veamos otro ejemplo:

```
#codigo original
        .word   -1
str_const7:
        .word   0
        .word   5
        .word
        .word   int_const1
        .ascii  "Int"
        .byte   0
        .align  2
```

```
#codigo final
        .word   -1
str_const7:
        .word   4
        .word   5
        .word   String_dispTab
        .word   int_const1
        .ascii  "Int"
        .byte   0
        .align  2
```

Vean que en el **.ascii**, de esta constante, esta el texto que representa, asi que, por lo que podemos deducir, **str_const7** hace referencia al nombre de la clase **Int**. Si revisan, encontraran los nombres de las demas clases.

Y segimos avanzando... despues de todas las **str_const** tenemos las **int_const**:

```
        #codigo original
        .word   -1
int_const9:
        .word   0
        .word   4
        .word
        .word   15
        .word   -1
int_const8:
        .word   0
        .word   4
        .word
        .word   13
```

Para estas constantes, al igual que para las **str_const**, unicamente debemos modificar la tercer linea colocando el numero de la **int_tag**, y la quinta linea colocando el nombre de la **Int_dispTab**

```
        #codigo final
        .word   -1
int_const9:
        .word   2
        .word   4
        .word   Int_dispTab
        .word   15
        .word   -1
int_const8:
        .word   2
        .word   4
        .word   Int_dispTab
        .word   13
```

En las **int_const**, el valor de la constante es el valor en la ultima linea **.word**, para **int_const9**, su valor es 15, y para **int_const8**, su valor es 13.

Finalmente, tenemos las **bool_const**, en las que tenemos que modificar exactamente las mismas dos lineas:

```
        #codigo original
        .word   -1
bool_const0:
        .word   0
        .word   4
        .word
        .word   0

        .word   -1
bool_const1:
        .word   0
        .word   4
        .word
        .word   1
```

```
        #codigo final
        .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
```

Los valores de estas constantes estan támbien en la ultima linea **.word**. **bool_const0** tiene como valor 0, que representa un **false** y **bool_const1** tiene como valor 1, que representa un **true**

La ultima porción de codigo que se les proporciona es esta:

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

Estas son las etiquetas para inicializar estos objetos, noten que no esta **IO_init**, de esta inicializacion se encarga la libreria **trap.handler**, asi que no nos preocupemos por eso.

Sin embargo, aun hay mucho codigo que debemos implementar, asi que sigamos con la explicación. Esta ultima porción no va en este lugar, antes debemos implementar lo siguiente:

```
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, veran que **str_const5** corresponde a la clase **Object**, **str_const6** corresponde a la clase **IO**, y asi sucecivamente, 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.

Despues de la tabla de nombre de clases, debemos implementar la tabla de objetos:

```
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 en el nombre en que escribieron las etiquetas en la tabla de nombre de clase (primero **Object**, luego **IO**, y así).

La siguiente sección que deben implementar, son las **dispatch tables** de cada una de las clases. Aquí támbien es muy importante que las escriban en el mismo orden que antes. Veamos como deben escribirlas:

```
#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 tambien. Vean también que, por ejemplo, la clase **String** tiene al inicio los metodos heredados de la clase **Object**, y tienen el prefijo **Object**: **Object.abort**, **Object.type_name**, etc. Si una clase sobreescribe un método de su padre, por ejemplo, si **Main** sobreescribiera el método **copy** de la clase **Object**, entonces, en su **dispatch table**, deberia estar, en vez de **Object.copy**, **Main.copy**, en la misma posición, y si alguna clase heredara de **Main**, pero no sobreescribiera el metodo **Copy**, deberia tener en su **dispatch table** el metodo **Main.copy**.

Veamos un ejemplo de como funcionaría. Consideremos el siguiente código a en **COOL**:

```
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_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 metodo **A.main** porque sobreescribe el método definido en **Main**, y como **B** hereda de **A**, pero no sobreescribe el método **Main**, en su **dispatch table** escribimos **A.main**. Estos métodos deben estar siempre en el mismo orden; el metodo **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**. Imaginen que tenenemos el siguiente código en **COOL**:

```
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**:

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

Ignoremos el **.word -1** (aun así, deben agregarlo). Enfoquemonos en **.word 0**, este 0, representa la posición en la tabla de nombres de clase que definieron anteriormente. 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 ahí. Luego, la etiqueta de su **dispatch table**, y listo.

Ahora con la clase **Main**:

```
        .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**. Despúes, 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 vacio para la clase **String**, y falso para la clase **Bool**. Si el campo no pertenece a ninguna de estas clases, lo incializaremos como 0, para reprensetar algo similar a **null**.

Veamos ahora la clase **A**:

```
        .word   -1
A_protObj:
        .word   6 #posicion en class_nameTab
        .word   5 #cantidad de parametros
        .word   A_dispTab #dispatch table
        .word   0 #valor inicial de out
        .word   str_const10 #valor inicial de out2
```

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 despues tenemos **str_const10**, que, si buscamos mas arriba, hace referencia al string vacio. Esto es porque **out2** es de tipo **String**.

Veamos un ultimo ejemplo con la clase **B**:

```
        .word   -1
B_protObj:
        .word   7 #posicion en class_nameTab
        .word   7 #cantidad de campos
        .word   B_dispTab #dispatch table
        .word   0 # valor inicial de out
        .word   str_const10 #valor inicial de out2
        .word   int_const3 #valor inicial de out3
        .word   0 #valor inicial de out4
```

Y nos damos cuenta que es 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.

Y 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 nos dan desde el inicio:

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

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

Init de la clase **Main**:

```
Main_init:
        addiu   $sp $sp -12
        sw      $fp 12($sp)
        sw      $s0 8($sp)
        sw      $ra 4($sp)
        addiu   $fp $sp 4
        move     $s0 $a0
        jal     Object_init
        la      $a0 IO_protObj
        jal     Object.copy
        jal     IO_init
        sw      $a0 12($s0)
        move     $a0 $s0
        lw      $fp 12($sp)
        lw      $s0 8($sp)
        lw      $ra 4($sp)
        addiu   $sp $sp 12
        jr      $ra
```

Como usaremos $sp, $fp y $s0, debemos guardarlos al stack. Esto se hace en las primeras 4 instrucciones de **Main_init**. En la siguiente instrucción, hacemos que $fp quede en la punta del stack, para saber donde esta nuestra referencia, ya que $sp varia constantemente. Y ahora, movemos a $s0, $a0, esto se hace porque en $a0 tenemos la dirección final del objeto, y se sobreescribira, asi que la pasamos a $s0 para preservarla. **jal Object_init** se hace porque debemos inicializar los campos que tiene **Object**, que es el padre de **Main**, antes de poder inicializar los campos propios de **Main**. Cuando regresa de **Object_init**, tenemos los campo inicializados por **Object** en $s0 (aunque támbien los tendremos en $a0) , y ya podemos inicializar los campos propios de **Main**. Carga a $a0 el **IO_protObj**, y salta a **Object.copy** para realizar una copia de él. **Object.copy** se encuentra en el **trap.handler**, asi 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 posicion 12 de $s0 (porque es el cuarto campo), y como no hay nada mas que hacer, regresamos la dirección del objeto ($s0) a $a0, restauramos el stack, y regresamo.

Init de la clase **B**:

```
B_init:
        addiu   $sp $sp -12
        sw      $fp 12($sp)
        sw      $s0 8($sp)
        sw      $ra 4($sp)
        addiu   $fp $sp 4
        move    $s0 $a0
        jal     A_init
        la      $a0 int_const2
        sw      $a0 20($s0)
        move    $a0 $s0
        lw      $fp 12($sp)
        lw      $s0 8($sp)
        lw      $ra 4($sp)
        addiu   $sp $sp 12
        jr      $ra
```

Guardamos $sp, $fp y $s0 al stack. En la siguiente instrucción, hacemos que $fp quede en la punta, como en **Main_init** para saber donde esta nuestra referencia. Y ahora, movemos a $s0, $a0, para no perderla, y saltamos a **A_init** que inicializa los campos que **B** hereda de **A**. Cuando regresa de **A_init**, tenemos los campo inicializados por **A** en $s0 (y támbien los tendremos en $a0). Los campos que inicializo **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 posicion 20 de $s0 (porque es el sexto campo). Vean que el septimo campo se declara, pero no se inicializa, asi que no hacemos nada (no lo inicializamos), regresamos la dirección del objeto ($s0) a $a0, restauramos el stack, y regresamo.

Luego de codificar todos los **inits**, finalmente pueden empezar a codificar cada uno de los métodos declarados, y listo! han terminado su compilador.