

# Taller de Diseño de Software - Compiladores

Integrantes del grupo: Ezequiel Depetris  
Gaston Massimino

A continuación se presentan consideraciones tomadas en cuenta, cambios y decisiones de diseño en cada una de las etapas llevadas a cabo en el desarrollo del compilador:

- **Analizador Sintáctico:** Se respetó la gramática como fue presentada en un principio pero se decidió agregar un no terminal extra al cual llamamos *lambda* y es utilizado para representar el caso vacío. Otra decisión no de diseño sino que viene dada por leer el mail con las condiciones un poco más tarde, fue usar la extensión *.plum* para nuestros test. No tienen nada que ver con el diseño sino que solo fue por no leer el mail a tiempo donde ya se presentaba un formato para los archivos, y no querer cambiar la extensión de 54 archivos.
- **Analizador Semántico:** Para esta etapa como primera medida se corrigieron detalles de la entrega pasada. Se le agregó comentarios a todos los test, donde se explica el propósito del mismo (que es lo que se quiere testear). Algunos test se modificaron en detalles mínimos, además de contar con otros nuevos. Se realizó una refactorización del código entregado en la etapa anterior. El método *main* ya no forma parte del archivo *Parser.cup*. Se cambió la implementación de la declaración de atributos, se dejó de lado el modelo *type* seguido de una lista de *id's* para pasar a tener una lista de elementos de la forma *type id*. Se usaron varias implementaciones del patrón Visitor, entre las cuales mencionamos las siguientes:
  - ASTVisitor: interfaz que implementan todos los visitor's
  - PrintAST: Realiza una pasada por el árbol imprimiendo su estructura
  - DeclarationChecker: Recorre el árbol decorando todas las expresiones con sus correspondientes tipos
  - TypeChecker: Realiza el chequeo de tipos de métodos, variables, parámetros y expresiones
  - MainChecker: Revisa que el programa cuente con una clase *main*, la cual contiene un método *main*, y chequea que no halla clases repetidas
  - CycleChecker: Recorre el árbol en búsqueda de expresiones *break* y *continue* para revisar que solo se encuentren dentro de un ciclo
  - ReturnChecker: Recorre el árbol buscando sentencias *return* para verificar que estén dentro del cuerpo de un método

Por ultimo, se decidió implementar una clase *Error* con todos los métodos utilizados para lanzar o identificar los errores del programa por linea de comandos. Hasta el momento solo se informa de manera muy sencilla si hay algún error de tipos o errores semánticos en las estructuras, dando una ubicación de linea y columna del error encontrado.

- **Generador de código intermedio:** Al comenzar esta entrega se incorporaron los test provistos por los profesores, lo que hizo salir a luz algunos errores que tenían que ver con la etapa pasada. Entonces como primera tarea, se revisaron las entregas anteriores para eliminar todo tipo de error. Luego de esto se prosiguió a tomar un código fuente de nuestro lenguaje *plum* y se lo transformo a un *pseudo assembler* que se asemeja a un código assembler de tres direcciones, por medio de un nuevo visitor implementado en esta etapa(*IntermediateCode*). Como el resultado final va a ser un código assembler de tres direcciones, se decidió escribir este *pseudo assembler* “lo mas parecido” a lo que seria el código assembler final. Es decir, optamos por la convención de generar sentencias que se construyen de izquierda hacia derecha, por lo que se encontraran valores null en los operados de mas a la derecha y jamas al medio.

Otra consideración o convención que se siguió es de no crear variables temporales para almacenar resultados que en assembler ya quedarían almacenados en los flag, por ejemplo **cmp var1 var2 null**, donde el resultado de la comparación queda seteado en un flag y no haría falta crear un temporal y tener algo así **cmp var1 var2 t0**. Por ultimo, se consideró que trabajar desde el lenguaje *Java* es mas fácil que *Assembler*, por esto es que se trato de crear la mayor cantidad de sentencias de pseudo assembler en esta etapa con el fin de hacer una traducción casi textual en la próxima etapa, y esto también explica un poco las decisiones tomadas y convenciones respetadas.

- **Generador de código objeto para enteros:** Dada la lista de sentencias generadas en la etapa anterior, se implementó una clase llamada *AsmGenerator*, la cual toma dicha lista y para cada sentencia le genera una instrucción en código assembler correspondiente. Dicho código assembler se guarda en un archivo con nombre **program.s**. Una vez que se contó con un programa assembler, se tuvo que volver a tocar el código de la etapa anterior ya que saltaron a la luz muchos errores en cuanto a la forma de escribir el assembler, saltos, valores y cosas propias del lenguaje. Con el código de la etapa anterior funcionando, para poder ejecutar los programas asm, se le dio un *offset* a las variables temporales (t0), pero esto no lo hicimos en *AsmGenerator*, sino que regresamos al *DeclarationChacker* y algunos otros se setearon en *IntermediateCode*. Hasta ese momento se generaba código para programas que solo tengan una clase *Main* y un método *main*. Cuando esto funcione correctamente,

se implementó lo faltante para esta etapa, tener variables y métodos en el método main, pertenecientes a otras clases. Generar código para objetos presento una dificultad para acceder a atributos de tipo arreglos de un objeto, por lo que no se implementó porque era un caso similar a tener objetos con atributos que sean objetos, y dicho caso no se tomo para la realización de este compilador ya que no contamos con un heap para crear objetos dinámicamente. Lo que se destaca en esta etapa es la llamada a métodos, ya que se presentaron problemas con los parámetros, se solucionó con un código genérico independientemente del numero de parámetros y de si se cuenta con una llamada de método anidada. La solución fue salvar todos los parámetros, es decir, guardarlos en la pila para no perderlos de los registros. Se sabe que una forma eficiente de hacer esto seria visitar el método y solo salvar los parámetros si adentro contiene otro método anidado que tome parámetros.

- **Optimizador:** Con el compilador funcionando bajo los requerimientos propuestos al inicio de la materia, como ultima medida se optimizaron dos aspectos del mismo. Lo primero fue evitar lo que se llama *propagación de constantes*, es decir, dejar de generar variables temporales con offset para las constantes, tratarlas como una expresión a resolver y utilizar ya el valor final de la misma, si se le quiere llamar asi. Lineas de código de esta manera **num = 2+2+2+2+2** son vistas por el compilador de la siguiente manera **num = 10**, entonces se evita realizar 4 sumas con todo lo que eso implica(definir 5 variables temporales para cada una de las constantes, luego 4 temporales mas para los 4 resultados de la suma y realizar 4 operaciones de suma con sus correspondientes movimientos).

```
1 //testing a program with empty class
2 class main{
3     void printInt(integer a) extern;
4
5     void main(){
6         integer number1;
7
8         number1 = 2*3*4*3+2+45+24*34+1+2*34*4+5*6+3;
9
10        printInt(number1);
11
12        return;
13    }
14 }
```

(codigo utilizado para probar la optimizacion)

95	_EndMethodmain:	39	_EndMethodmain:
96	_EndClassmain:	40	_EndClassmain:
97		41	
98		42	
99	callq ___stack_chk_fail	43	callq ___stack_chk_fail
100	.cfi_endproc	44	.cfi_endproc
101	.subsections_via_symbols	45	.subsections_via_symbols
102		46	

(diferencia de lineas generadas sin optimizador Vs con optimizador)

8	_main:	8	_main:
9	_InitMethodmain:	9	_InitMethodmain:
10	PUSHQ %RBP	10	PUSHQ %RBP
11	MOVQ %RSP, %RBP	11	MOVQ %RSP, %RBP
12	SUBQ \$2048, %RSP	12	SUBQ \$2048, %RSP
13		13	
14	MOVL \$1241, %EAX	14	MOVQ \$2, %RAX
15	MOVL %EAX, -16(%RBP)	15	IMULQ \$3, %RAX
16		16	MOVQ %RAX, -24(%RBP)
17	MOVQ %RDI, -24(%RBP)	17	MOVQ -24(%RBP), %RAX
18	MOVQ %RSI, -32(%RBP)	18	IMULQ \$4, %RAX
19	MOVQ %RDX, -40(%RBP)	19	MOVQ %RAX, -32(%RBP)
20	MOVQ %RCX, -48(%RBP)	20	
21	MOVQ %R8, -56(%RBP)	21	MOVQ -32(%RBP), %RAX
22	MOVQ %R9, -64(%RBP)	22	IMULQ \$3, %RAX
23	MOVQ %R10, -72(%RBP)	23	MOVQ %RAX, -40(%RBP)
24		24	
25	MOVQ -16(%RBP), %RDI	25	MOVQ -40(%RBP), %RAX
26	CALL _printInt	26	ADDQ \$2, %RAX
27	MOVQ -72(%RBP), %R10	27	MOVQ %RAX, -56(%RBP)
28	MOVQ -64(%RBP), %R9	28	
29	MOVQ -56(%RBP), %R8	29	MOVQ -56(%RBP), %RAX
30	MOVQ -48(%RBP), %RCX	30	ADDQ \$45, %RAX
31	MOVQ -40(%RBP), %RDX	31	MOVQ %RAX, -72(%RBP)
32	MOVQ -32(%RBP), %RSI	32	
33	MOVQ -24(%RBP), %RDI	33	MOVQ \$24, %RAX
34		34	IMULQ \$34, %RAX
35		35	MOVQ %RAX, -80(%RBP)
36	LEAVE	36	
37	RET	37	MOVQ -72(%RBP), %RAX
38		38	ADDQ -80(%RBP), %RAX
39	_EndMethodmain:	39	MOVQ %RAX, -96(%RBP)
40	_EndClassmain:	40	
41		41	MOVQ -96(%RBP), %RAX
42		42	ADDQ \$1, %RAX
43	callq ___stack_chk_fail	43	
44		44	MOVQ %RAX, -112(%RBP)

(diferencia del codigo generado con optimizacion Vs sin optimizacion)

La segunda optimización que se le realizó al compilador fue salvar solo los registros necesarios cuando llamamos a una función, es decir, dejar de salvar los 6 registros en todos los casos, incluso cuando la función invocada no toma parámetros. De esta manera, se ahorran 2 líneas de código assembler, junto con un lugar en memoria por cada registro salvado sin necesidad de hacerlo. Es decir, para el ejemplo del método invocado que no toma parámetro, nos ahorramos 12 líneas de código y 6 lugares de memoria, multiplicando esto por todas las llamadas a métodos que puede tener un código hace la diferencia.

```
1  //testing a program with empty class
2  class main{
3      void printInt(integer a) extern;
4      integer set(integer a, integer num){
5          a = num;
6          return a;
7      }
8
9      void print1(){
10         printInt(1);
11         return;
12     }
13
14     void printA(){
15         print1();
16         print1();
17         print1();
18         print1();
19         print1();
20         print1();
21         print1();
22         print1();
23         return;
24     }
25
26     void main(){
27         integer number1;
28         number1 = set(number1, 1);
29
30         printA();
31
32         return;
33     }
34 }
```

*(codigo utilizado para probar la optimizacion)*

22	_EndMethodset:	36	_EndMethodprint1:
23	_InitMethodprint1:	37	_InitMethodprintA:
24	PUSHQ %RBP	38	PUSHQ %RBP
25	MOVQ %RSP, %RBP	39	MOVQ %RSP, %RBP
26	SUBQ \$2048, %RSP	40	SUBQ \$2048, %RSP
27	MOVQ %RDI, -48(%RBP)	41	MOVQ %R10, -48(%RBP)
28	MOVQ %RSI, -48(%RBP)	42	CALL _InitMethodprint1
29	MOVQ %RDX, -56(%RBP)	43	MOVQ -48(%RBP), %R10
30	MOVQ %RCX, -64(%RBP)	44	
31	MOVQ %R8, -72(%RBP)	45	MOVQ %R10, -56(%RBP)
32	MOVQ %R9, -80(%RBP)	46	CALL _InitMethodprint1
33	MOVQ %R10, -88(%RBP)	47	MOVQ -56(%RBP), %R10
34	MOVQ \$1, %RDI	48	
35	CALL _printInt	49	MOVQ %R10, -64(%RBP)
36	MOVQ -88(%RBP), %R10	50	CALL _InitMethodprint1
37	MOVQ -80(%RBP), %R9	51	MOVQ -64(%RBP), %R10
38	MOVQ -72(%RBP), %R8	52	
39	MOVQ -64(%RBP), %RCX	53	MOVQ %R10, -72(%RBP)
40	MOVQ -56(%RBP), %RDX	54	CALL _InitMethodprint1
41	MOVQ -48(%RBP), %RSI	55	MOVQ -72(%RBP), %R10
42	MOVQ -40(%RBP), %RDI	56	
43		57	MOVQ %R10, -80(%RBP)
44		58	CALL _InitMethodprint1
45	LEAVE	59	MOVQ -80(%RBP), %R10
46	RET	60	
47		61	MOVQ %R10, -88(%RBP)
48	_EndMethodprint1:	62	CALL _InitMethodprint1
49	_InitMethodprintA:	63	MOVQ -88(%RBP), %R10
50	PUSHQ %RBP	64	
51	MOVQ %RSP, %RBP	65	MOVQ %R10, -96(%RBP)
52	SUBQ \$2048, %RSP	66	CALL _InitMethodprint1
53	MOVQ %RDI, -96(%RBP)	67	MOVQ -96(%RBP), %R10
54	MOVQ %RSI, -104(%RBP)	68	
55	MOVQ %RDX, -112(%RBP)	69	MOVQ %R10, -104(%RBP)
56	MOVQ %RCX, -120(%RBP)	70	CALL _InitMethodprint1
57	MOVQ %R8, -128(%RBP)	71	MOVQ -104(%RBP), %R10
58	MOVQ %R9, -136(%RBP)	72	
59	MOVQ %R10, -144(%RBP)	73	
60	CALL _InitMethodprint1		

(diferencia del código generado con optimización Vs sin optimización)

```
126 _EndMethodmain:  
127 _EndClassmain:  
128  
129  
130 callq ___stack_chk_fail  
131 .cfi_endproc  
132 .subsections_via_symbols  
133
```

```
171 _EndMethodmain:  
172 _EndClassmain:  
173  
174  
175 callq ___stack_chk_fail  
176 .cfi_endproc  
177 .subsections_via_symbols  
178
```

*(Diferencia de lineas generadas con optimizador Vs sin optimizador)*