Proyecto de Autómatas y Lenguajes
Curso 18/19
Grupo Miércoles 16:00-18:00

Objetivo de la sesión
Esta sesión tiene los siguientes objetivos

- **Avisos**
    - Fechas de entrega
    - Material adicional
        - Programa entrada / salida de ejemplo y .ol
        - Casos de prueba
    - Correcciones de algunas erratas durante los próximos días
- **Proc y OO (ver el guión)**

TAREA OO1:
Comprobación del progreso, incorporación de instancias a la escritura de la TS y generación de código de instanciación, descarte y asignación instancias.

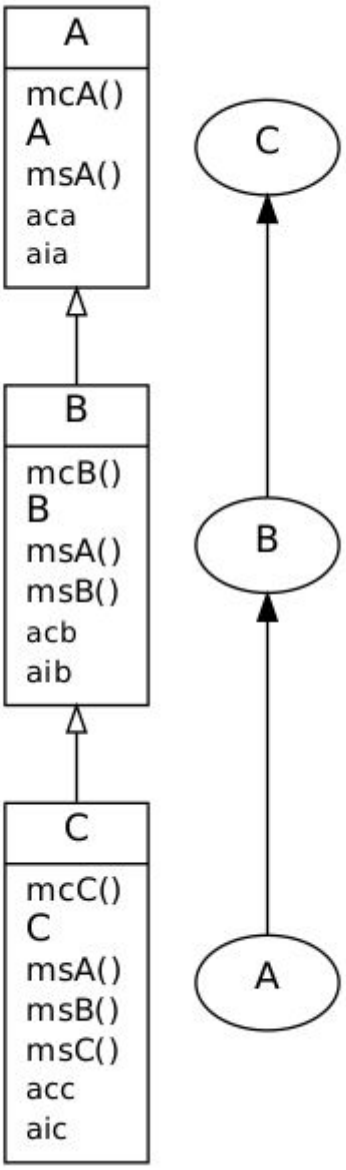
1. Analiza el siguiente código .ol

```
main
{

        class A {
                unique boolean aca;
                int aia;
                  function unique   none  mcA () { printf 1; return none; }
                  function int msA() {printf 2;  return 2;}
                 };
        class B inherits A {
                unique int acb;
                int aib;
                  function unique   none  mcB () { printf 3; return none; }
                  function int msB() {printf 4;  return 4;}
                  function int msA() {printf 5;  return 5;}
                };
        class C inherits B {
                unique int acc;
                int aic;
                  function unique   none  mcC () { printf 6; return none; }
                  function int msC() {printf 7;  return 7;}
                  function int msB() {printf 8;  return 8;}
                  function int msA() {printf 9;  return 9;}
                };

        {A} a;
        {B} b;
        {C} c;


        c = instance_of C ();

        c.aia = 10;
        c.aib = 100;
        c.aic = 1000;

        b = c;
        a = b;

        printf a.aia;
        printf b.aib;
        printf c.aic;



        a =instance_of A();
        b =instance_of B();

        a.msA();
        b.msA();
        c.msA();

        discard a;
        descard b;
```

```
        discard c;


        printf 0;
}
```

2. Utiliza la tabla de símbolos del día pasado

3. Comprueba que tu función de escritura de .dot genera algo similar a esto



4. Invoca a tu función de escritura para que escriba la información de métodos sobreescribibles a un fichero ts_s1_goo.asm

5. Consulta la documentación en la parte de acceso a atributos de instancia y métodos sobreescribibles

9. Considera implementar las siguientes funciones

```
void llamarMetodoSobreescribibleCualificadoInstanciaPila
      (FILE * fd_asm, char * nombre_metodo)
```

- Función que genera código para la llamada a un método sobreescribible cualificado por lo que ocupa cima de la pila.
- El método es nombre_metodo

```
void asignarDestinoEnPila(FILE* fpasm, int es_variable)
```

- Función que genera el código para realizar una asignación que toma de la pila
    - Primero el valor
        - Si es el valor que hay que asignar es_variable es 0
        - En otro caso (es variable o equivalente) es 1
    - Encima la dirección donde hay que dejarlo
- Sobre las asignaciones es importante mencionar
    - Las asignaciones sólo usan la pila.
    - Eso quiere decir que todo está en la pila.
    - El hecho de poder indexar y asignar por ejemplo algo de un objeto hace que la secuencia de acciones pueda parecer un poco enrevesada pero basta con introducir en la pila las cosas en orden inverso.
    - Por ejemplo, para v = a[3];
        - Primero 3
        - Luego se escribe el elemento 3 del vector a
        - Luego se escribe v
        - Luego se llama a asignar con destino en pila
    - Por ejemplo a.b = b[3].c;
        - Primero 3
        - Luego se escribe el elemento 3 del vector b
        - Luego se accede al atributo c de la instancia en la pila
        - Luego se escribe a
        - Luego se accede al atributo b de la instancia en la pila
        - Luego se llama a asignar con destino en pila.
- Respecto a qué es variable y que no en las asignaciones
    - Los valores concretos como c = 9; a.c = 9; o a[3] = 8; o a[4].c = 8; no lo son y es_variable == 0
    - Las instancias como a = instance_of C no lo es y es_variable == 0
    - Cuando la parte derecha de una asignación es variable (x = y;) o equivalente (x = c.a; x = a[4];) sí lo es y es_variable == 1

```
void accederAtributoInstanciaDePila
       (FILE * fd_asm, char * nombre_atributo)
```

- Función que genera el código para acceder al atributo nombre_atributo de la instancia cuya dirección está en la cima de la pila

9. Utilizalas en un programa principal como el siguiente

```
#include <stdio.h>
#include "generacion_omicron.h"



int main(int argc, char ** argv)
{
       FILE * fd_asm;


       fd_asm = fopen(argv[1],"w");
       escribir_subseccion_data(fd_asm);


       escribir_cabecera_bss(fd_asm);

       // {A} a;
       declarar_variable(fd_asm, "a", 0, 1);
       // {B} b;
       declarar_variable(fd_asm, "b",  1,  1);
       // {C} c;
       declarar_variable(fd_asm, "c",  2,  1);

       escribir_segmento_codigo(fd_asm);
       fprintf(fd_asm,"\textern _msA,  _msB, _BmsA, _msC, _CmsA, _CmsB,
_no_defined_method, _mcA, _mcB, _mcC,  _set_offsets, _create_ms_table,
_offset_msA, _offset_msB,  _offset_msC,  _offset_aia, _offset_aib, _offset_aic,
_msA, _msB, _msC, _aca, _acb, _acc\n");



// PROGRAMA PRINCIPAL

       escribir_inicio_main(fd_asm);
       fprintf(fd_asm,"\tcall _create_ms_table\n");


       // c = instanceof C;
       instance_of(fd_asm,"C",3);
       escribir_operando(fd_asm,"c",1);
       asignarDestinoEnPila(fd_asm,0,"eax","ebx");

       // b = c;
       escribir_operando(fd_asm,"c",1);
       escribir_operando(fd_asm,"b",1);
       asignarDestinoEnPila(fd_asm,1,"eax","ebx");
```

```
	// a = b;
	escribir_operando(fd_asm,"b",1);
	escribir_operando(fd_asm,"a",1);
	asignarDestinoEnPila(fd_asm,1,"eax","ebx");


	// discard c;
	escribir_operando(fd_asm,"c",1);
	discardPila(fd_asm,"eax");



	// b = instanceof B;
	instance_of(fd_asm,"B",2);
	escribir_operando(fd_asm,"b",1);
	asignarDestinoEnPila(fd_asm,0,"eax","ebx");

	// a = instanceof A;
	instance_of(fd_asm,"A",1);
	escribir_operando(fd_asm,"a",1);
	asignarDestinoEnPila(fd_asm,0,"eax","ebx");

	// discard a;
	escribir_operando(fd_asm,"a",1);
	discardPila(fd_asm,"eax");

	// discard b;
	escribir_operando(fd_asm,"b",1);
	discardPila(fd_asm,"eax");

	escribir_fin(fd_asm);
	fclose(fd_asm);

	return 0;
}
```

10. Comprueba que tras estos comandos, tu programa no deja fugas de memoria

```
gcc -Wall -g -o  main_s2_goo *.c

valgrind --leak-check=yes ./main_s2_goo  s2goo.asm

nasm -felf32 s2goo.asm

nasm -felf32 ts_s2_goo.asm

gcc -o s2goo -g -m32 s2goo.o ts_s2_goo.o

valgrind --leak-check=yes ./s1goo
```

Actividad PROC1: Gestión de funciones

Material de consulta

- La presentación de generación de código
- La presentación de análisis semántico
- Este guión.

La gestión de funciones tiene relación con las siguientes funciones de la librería de generación de código de ómicron.

OBS. Muy importante.
A lo largo de este guión se muestra la generación de código tal y como está en la documentación referida.
Sin embargo, será muy interesante que cuando generes código utilices las funciones de tu librería de generación
Busca cuando tengas que generar código entre las siguientes funciones las más adecuadas.
Verás que en algunos puntos usar estas funciones cambia un poco la filosofía (tal vez debas utilizar la pila para almacenar temporalmente algo más que las subexpresiones aritméticas)
Estas funciones son básicamente el contenido de la práctica de generación.

```
void declararFuncion(FILE * fd_asm, char * nombre_funcion, int num_var_loc)
```

- Generación de código para iniciar la declaración de una función.
- Es necesario proporcionar
    - Su nombre
    - Su número de variables locales

```
void retornarFuncion(FILE * fd_asm, int es_variable)
```

- Generación de código para el retorno de una función.
    - La expresión que se retorna está en la cima de la pila.
    - Puede ser una variable (o algo equivalente) en cuyo caso exp_es_direccion vale 1
    - Puede ser un valor concreto (en ese caso exp_es_direccion vale 0)

```
void escribirParametro(FILE* fpasm, int pos_parametro, int num_total_parametros)
```

- Función para dejar en la cima de la pila la dirección efectiva del parámetro que ocupa la posición pos_parametro (recuerda que los parámetros se ordenan con origen 0) de un total de num_total_parametros

```
void escribirVariableLocal(FILE* fpasm, int posicion_variable_local)
```

- Función para dejar en la cima de la pila la dirección efectiva de la variable local que ocupa la posición posicion_variable_local (recuerda que ordenadas con origen 1)

```
void operandoEnPilaAArgumento(FILE * fd_asm, int es_variable)
```

- Como habrás visto en el material, nuestro convenio de llamadas a las funciones asume que los argumentos se pasan por valor, esto significa que siempre se dejan en la pila "valores" y no "variables"
- Esta función realiza la tarea de dado un operando escrito en la pila y sabiendo si es variable o no (es_variable) se deja en la pila el valor correspondiente

```
void llamarFuncion(FILE * fd_asm, char * nombre_funcion, int num_argumentos)
```

- Esta función genera código para llamar a la función nombre_funcion asumiendo que los argumentos están en la pila en el orden fijado en el material de la asignatura.
- Debe dejar en la cima de la pila el retorno de la función tras haberla limpiado de sus argumentos
- Para limpiar la pila puede utilizar la función de nombre limpiarPila

```
void limpiarPila(FILE * fd_asm, int num_argumentos)
```

- Genera código para limpiar la pila tras invocar una función
- Esta función es necesaria para completar la llamada a métodos, su gestión dificulta el conocimiento por parte de la función de llamada del número de argumentos que hay en la pila

La gestión de las funciones conlleva los siguientes conceptos y elementos

1. Elemento	T ASem	T GenCód
1. Declaración de parámetros ● Cómo se gestiona su declaración	t34	
1. Gestión del nombre de la función en sí ● Cuándo se crea la nueva tabla de símbolos? ● Cuándo se sabe cuántos parámetros tiene? ● Cuándo se actualiza la información del ámbito global?	t35-39 **t35-37**	
1. Convenio de llamadas C ● Dónde y quién deja los argumentos de la función. ● Cómo se dejan los argumentos de la función ● Dónde se deja el retorno de la función ● Quién limpia los argumentos de la llamada		t63
1. Gestión de la llamada a función ● Cómo compruebo que la función existe ● Cómo compruebo que la llamada no es un argumento de una función (prohibido por la semántica de ALFA) ● Cómo sé que el número de parámetros de la llamada coincide con el de la declaración ● Se comprueba algo relacionado con el tipo de los argumentos ? ● Qué tipo propaga como expresión la llamada a una función? ● Es una dirección una llamada a una función ? ● ¿Cómo calculo el número de parámetros de la llamada? ● Ten en cuenta comprobaciones semánticas adicionales para retorno none de funciones aunque **NO SE USARÁN FUERA DE LOS ELEMENTOS ORIENTADOS A OBJETO**	t60-62 **56-60**	
1. Generación de código de la llamada a una función ● Qué código ensamblador se genera? ● Uso de función void llamarFuncion(FILE * fd_asm, char * nombre_funcion, int num_argumentos);		t 88-91

● Ten en cuenta que hay que hacer lo mismo en la regla de llamada a función como sentencia simple, no ligada a una expresión		
1. ¿Cómo evoluciona la pila cuando se gestiona una función ? ● Para qué se usa ebp? ● Cómo se preserva el valor de ebp? ● Dónde (y cómo se accede) están los argumentos en función de ebp? ● Dónde (y cómo se accede) están las variables locales en función de ebp? ● Cómo se restaura ebp?		t64-87
1. "Plantillas" de inicio y final del cuerpo de las funciones ● Uso de funciones void declararFuncion(FILE * fd_s, char * nombre_funcion, int num_var_loc); void retornarFuncion(FILE * fd_s, int es_variable);		t93-95
1. Gestión de los parámetros de una función ● Son valores? ● Son direcciones? ● Qué se hace cuando el argumento es un identificador? (x) ● Qué se hace cuando el argumento es un vector indexado (v[3]) ● Qué se hace cuando el argumento es una expresión distinta de estas (x - 3 * v[3]) ● Uso de función void operandoEnPilaAArgumento(FILE * fd_asm, int es_variable);		t88-90
1. Cómo referirse a los diferentes tipos de variables nuevos (argumentos, variables locales) junto con los anteriores (variables globales) desde una función ● Uso de funciones void escribirParametro(FILE* fpasm, int pos_parametro, int num_total_parametros); void escribirVariableLocal(FILE* fpasm, int posicion_variable_local);		t96-103