Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 – Lenguajes de Programación I
Abril-Julio 2006

Carnet: _____

Nombre: _____

Examen I

(40 puntos)

Antes de empezar, revise bien el examen, el cual consta de 4 (CUATRO) preguntas.

Pregunta 0	Pregunta 1	Pregunta 2	Pregunta 3	Total
10 puntos	10 puntos	10 puntos	10 puntos	40 puntos

Pregunta 0 — 10 puntos

Considere la estructura de selección condicional múltiple ofrecida bajo el nombre `case` en la familia de "los Algol", incluyendo Pascal, Modula y Ada, y bajo el nombre `switch` en C, C++ y Java.

En relación con esta estructura de control, responda las siguientes preguntas:

(a) La posibilidad de que el valor de la expresión discriminante del `case`/`switch` no esté contemplada en ninguna de las ramas es manejada de manera diferente en varios lenguajes.

Veamos varios ejemplos: En C y en Fortran 90, la instrucción no tiene ningún efecto (esto es, se comporta como el `skip` de GCL). En Pascal y Modula, debe darse un error de ejecución, aunque entre ellos se tiene la diferencia de que la definición de Modula exige ofrecer al programador la posibilidad de utilizar una cláusula por defecto, mientras que en Pascal Estándar no existe tal posibilidad (todos los valores considerados deben ser listados explícitamente). En Ada, se prohíbe este hecho exigiendo que las ramas cubran todos los posibles valores del tipo de la expresión discriminante (y, como Modula, la definición del lenguaje exige ofrecer la posibilidad de utilizar una cláusula por defecto).

Comente estas cuatro alternativas en relación con nuestros acostumbrados criterios de diseño/análisis de lenguajes de programación.

(b) Michael Scott señala, en el texto *"Programming Language Pragmatics"*, que una de las ventajas de que un lenguaje de programación provea el `case`/`switch` como alternativa a una secuencia de `if`/`else` anidados es el permitir a los compiladores de tal lenguaje la posibilidad de generar código más eficiente mediante "tablas de salto".

Igualmente, Scott señala que, según los valores considerados en un `case`/`switch`, resulta más conveniente manejar la tabla de salto mediante búsqueda secuencial, búsqueda binaria o como tabla de *hash*.

¿En qué casos, de acuerdo con Scott, conviene utilizar cada una de estas alternativas de implementación? ¿Puede un buen compilador alternar la implementación en cada `case`/`switch` que procese, o está obligado a fijar de antemano una estrategia a utilizar en todos los casos?

Pregunta 1 — 10 puntos

Considere el siguiente encabezado/especificación para un procedimiento de cálculo de números factoriales, en el que n es un parámetro de entrada que debe mantenerse constante y f es un parámetro de salida:

procedure *factorial* (**in** n : *int* ; **out** f : *int*)
{ **precondition** $n \geqslant 0$ }
{ **postcondition** $f = n!$ }

Responda ahora las siguientes preguntas:

(a) Dé dos soluciones recursivas al problema planteado, una que no sea recursiva de cola (*tail recursive*) y otra que sí lo sea.

En ambos casos, sólo debe considerarse como caso base a $n = 0$. En todas las instancias en las que ocurra $n > 0$ debe ocurrir al menos una llamada recursiva.

Nota: En una solución recursiva, la recursión puede ocurrir en un procedimiento auxiliar en lugar de ocurrir directamente en el procedimiento principal (*factorial* en nuestro caso).

Otra nota: No puede utilizar iteración.

(b) Considere una llamada *factorial* $(2, a)$ realizada desde un contexto en el que a es una variable entera con tiempo de vida global, y suponga que estamos trabajando con un compilador/interpretador ingenuo en el que toda activación recursiva de un procedimiento es manejada con un nuevo marco de pila (esto es, con nuevo espacio en la pila para la información local del procedimiento). Muestre, para cada una de sus dos soluciones de (a), todos los estados de la pila a medida que se ingresa a todas las activaciones recursivas y se regresa de ellas.

Sea clara/o en su diagramación de la evolución de la pila. Puede mostrar un único primer diagrama con el estado de la pila que se obtiene al terminar de ingresar a todas las activaciones recursivas, pero luego debe mostrar por cada retorno un nuevo diagrama con el estado de la pila resultante.

Nota: La semántica operacional con la que debe manejar a los parámetros es la siguiente: Cada parámetro formal (n y f en el caso de *factorial*) es como una variable local del procedimiento; los parámetros de entrada (**in**) son inicializados con el valor de su parámetro real asociado; los parámetros de salida (**out**), de manera dual a los de entrada, no son inicializados pero son copiados hacia el parámetro real asociado al finalizar la activación del procedimiento.

(c) Suponga ahora que disponemos de un nuevo compilador/interpretador, no ingenuo, que sabe sacar provecho de la recursión de cola. Muestre cómo evolucionaría la pila ahora para su solución recursiva de cola con la misma llamada anterior *factorial* $(2, a)$.

De nuevo, sea clara/o en su diagramación de la evolución de la pila.

Pregunta 2 — 10 puntos

Los lenguajes Pascal y Modula ofrecen un constructor de tipos `set` para manejo de conjuntos con tipos bases discretos (preferiblemente pequeños, usualmente subrangos), como por ejemplo: `set of 50..100` para manejar conjuntos de enteros cuyos elementos deben estar entre 50 y 100, ambos extremos incluidos; `set of 'a'..'z'` para manejar conjuntos de caracteres que sólo puedan contener letras minúsculas.

Considere ahora el siguiente fragmento de un programa en Pascal:

```
var
    A: set of 1..10;
    B: set of 10..20;
    C: set of ?..?;
    i: 1..30;

...

C := A + B * [1..5, i]
```

En este fragmento se utilizan las operaciones sobre conjuntos de unión y de intersección, denotadas respectivamente con los símbolos `+` y `*`, y la construcción por enumeración `[1..5, i]` que denota un conjunto compuesto por los elementos 1, 2, 3, 4, 5 e `i`.

Para la realización de las operaciones, todos los operandos son considerados del tipo `set of integer`, pero debe realizarse un proceso de inferencia de tipos más detallado para analizar la compatibilidad del resultado de la expresión del lado derecho de la asignación con la variable del lado izquierdo de la misma.

En relación con este ejemplo, responda entonces las siguientes preguntas:

(a) Dé el subtipo mínimo de `set of integer` que puede ser inferido estáticamente para cada una de las subexpresiones del lado derecho de la asignación, incluyendo por supuesto la expresión total.

(b) Determine para cuáles subtipos de `set of integer` de la variable `C`, según los posibles valores enteros de las incógnitas `?`, debería un compilador generar un error estáticamente, generar código que verifique la posibilidad de error dinámicamente, o generar código que incondicionalmente realice la asignación.

Pregunta 3 — 10 puntos

Considere el siguiente fragmento de código en lenguaje C, correspondiente al cuerpo de algún procedimiento, en el que el alcance de cada una de las variables cubre únicamente el bloque en el que está declarada:

```
{   int a, b, c;
    ...
    {   int d, e;
        ...
        {   int f, g;
            ...
        }
        ...
    }
    ...
    {   int h, i, j;
        ...
    }
    ...
}
```

Asuma además que cada variable entera ocupa N *bytes* (considerando irrelevante que N sea 2, 4, 8 o cualquier otra cantidad).

Responda ahora las siguientes preguntas:

(a) ¿Cuál es la mínima cantidad de espacio requerida por las variables de este fragmento de código?

(b) Se desea construir una clase, a ser utilizada por un compilador, que permita calcular la mínima cantidad de espacio requerida por las variables de un fragmento de código cualquiera similar al presentado en el ejemplo (esto es: el cuerpo de un procedimiento, con bloques anidados, sólo variables enteras).

Su clase deberá contener métodos para:

- Inicialización de los atributos de la clase, el cual es llamado por el compilador al iniciar la lectura del fragmento de código.
- Entrada a bloque, que será llamado por el compilador cada vez que detecte el inicio de un nuevo bloque (incluyendo el bloque principal del cuerpo del procedimiento). Este método recibirá un parámetro de entrada que indique cuántas variables locales tiene el nuevo bloque.
- Salida de bloque, a ser llamado cada vez que se detecte un fin de bloque.
- Finalización de cuerpo, el cual será llamado por el compilador cuando el cuerpo del procedimiento termine de ser leído. Este método devolverá la cantidad mínima total requerida por las variables del fragmento procesado.

Puede construir la clase requerida usando el lenguaje Java, o construir un TAD (tipo abstracto de datos) en cualquier otro lenguaje de su preferencia.

Los métodos deben manejar condiciones de error (en el caso de Java, mediante excepciones; en cualquier otro lenguaje, como Ud. quiera). Por ejemplo: salir de una cantidad de bloques mayor que la cantidad de bloques a los que se ha entrado, que la finalización sea invocada aún con bloques abiertos, que la cantidad de variables a la entrada de un bloque sea un número negativo.