Universidad Nacional Autónoma de México

Facultad de Ciencias

Práctica 8

Organización y Arquitectura de Computadoras

Johann Ramón Gordillo Guzmán - 418046090

José Jhovan Gallardo Valdéz - 310192815

Diana Laura Nicolás Pavia - 314183093

Práctica presentada como parte del curso de **Organización y Arquitectura de Computadoras** impartido por el profesor **José de Jesús Galaviz Casas**.

06 de Noviembre del 2019

Link al código fuente: https://github.com/JohannGordillo

1. Preguntas

1. ¿Por qué es necesaria una convención para llamar a rutinas?

   Como hemos visto durante el curso, una computadora es útil porque tiene software. Para hacer que una computadora pueda hacer todas las cosas que puede hacer hoy en día se utilizan diversos lenguajes de programación, cada uno con su paradigma, sintaxis y convenciones distintas y como vemos, la diversidad de lenguajes no es obstáculo para que nuestra computadora pueda ejecutar nuestro código. Lo anterior se debe a que aunque que cada lenguaje de programación cuente con una sintaxis distinta, ésta está bien definida y los programas válidos sólo pueden estar construidas a partir de ésta. Esto se debe a que cada código es procesado y validado, para esto se construye su árbol de análisis sintáctico el cual verifica que en efecto, se tenga una sintaxis válida.

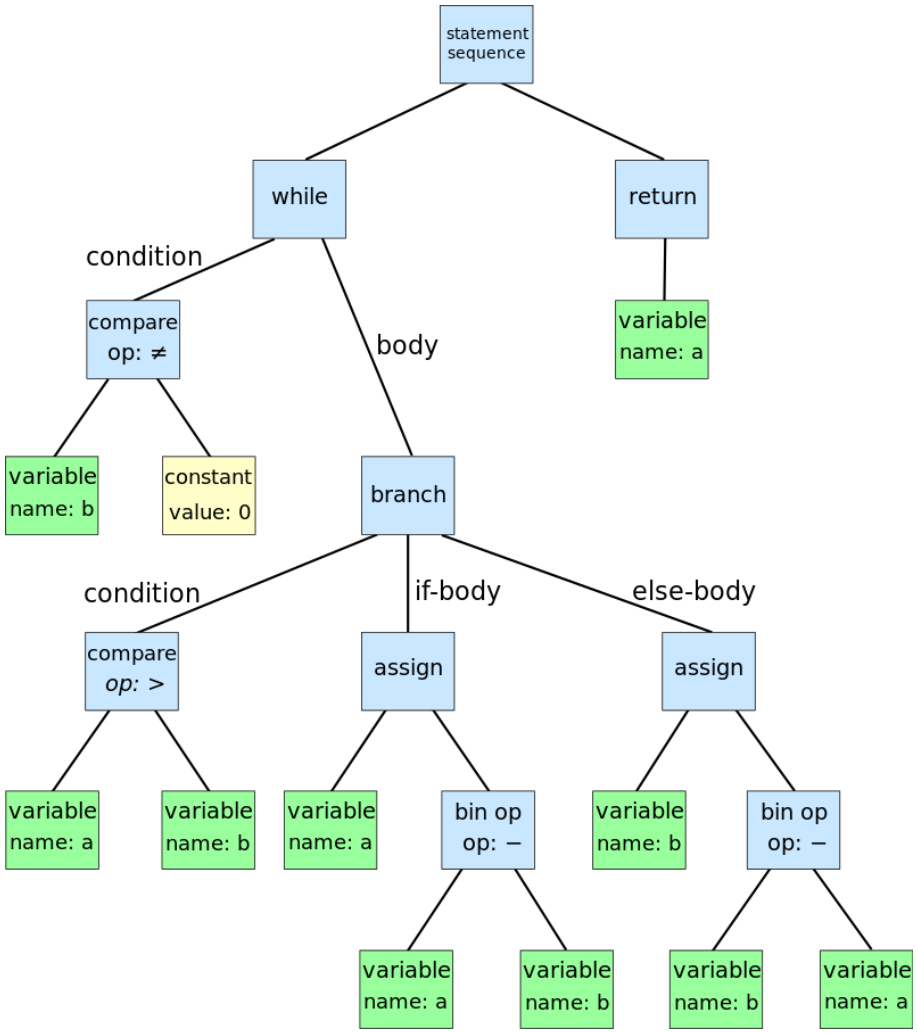

Figura 1: Ejemplo de análisis sintáctico

Luego, una vez que nuestro programa cuenta con una sintaxis válida éste es traducido a lenguaje ensamblador (usando otro programa) las llamadas a subrutinas tienen una forma genérica de ser traducidas (pues ya se sabe que cuentan con una sintaxis adecuada para la traducción a ensamblador), esto genera una convención (forma genérica) de traducción de llamadas a subrutinas por el ensamblador, que sirven para que la ejecución del programa pueda seguir de forma consistente una vez que se ha completado

la subrutina.

Por otra parte, las conveciones surgen de la necesidad de hacer uso libre de los registros del procesador mientras se está ejecutando una tarea. Imaginemos que una rutina está ejecutándose y manda a llamar a una subrutina; si no utilizarmos las convención del *Stack* para llamadas a subrutinas correríamos el riesgo de modificar los valores almacenados en registros que estaban siendo usados por la tarea invocadora, para evitar estos errores se utiliza la convención donde se acuerda respaldar los valores que sean necesarios para poder hacer uso de los registros.
Es esta práctica, basta con declarar el tamaño del *Stack* para la subrutina una vez debido a que cada llamada a subrutina utilizará el mismo tamaño ya que estamos usando recursión.

2. ¿Qué calcula la subrutina del ejercicio 2?

   Dado un número entero n mayor que cero, pasado como argumento en el registro $a0, la subrutina 'foo' regresa el n-ésimo término de la sucesión de Fibonacci en el registro $v0.

   Sabemos que la sucesión de Fibonacci es 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... y así sucesivamente, y que podemos obtener el enésimo término de la misma de manera recursiva tomando dos casos base:

   Primer caso base: n = 1. Regresamos el primer término de la sucesión, que es 1.

   Segundo caso base: n = 2. Regresamos el segundo término de la sucesión, que de nuevo es 1.

   Y calculando los demás casos usando el hecho de que el n-ésimo término está dado por la suma del (n-1)-ésimo término de la sucesión sumando con el (n-2)-ésimo término de la sucesión, que es justamente lo que la subrutina hace.

   En resumidas cuentas:

   Si n = 0, regresa 1.

   Si n = 1, regresa 1.

   De otra manera, regresa fibonacci(n-1) + fibonacci(n-2).

3. ¿Qué utilidad tiene el registro **$fp**? ¿se puede prescindir de él?

   Basándonos en el libro de Organización y Diseño de Computadoras de Patterson, citado en la bibliografía, el frame pointer es un valor que denota la localización de los registros guardados y de las variables locales para un procedimiento dado en el programa.

   El frame pointer no es indispensable, y se puede prescindir de él; además es raramente usado, excepto cuando la alocación en pila de una subrutina es determinada en tiempo de ejecución. Necesitamos usar el frame pointer cuado estamos guardando cantidades variables de espacio en la pila, por ejemplo, en procedimientos que usen arreglos de longitud variable.

   El registro $fp apunta a la primera palabra del frame de un procedimiento, también llamado marco, que es un segmento de la pila que contiene las variables locales y los registros guardados del procedimiento.

   Cabe destacar que en los ejemplos presentados en el libro anteriormente mencionado, el autor no hace uso del registro $fp, ya que al evitar hacer cambios al registro $sp dentro de un procedimiento, deja de ser necesario, pues la pila es ajustada únicamente durante el inicio y el final del procedimiento.

4. Considera el siguiente pseudocódigo de la figura 2. En donde **a[5]** es un arreglo de tamano 5 y "..." son otras acciones que realiza la rutina, además, supón que en la función B se realizan cambios en los registros **$s0**, **$s1** y **$s2**. Bosqueja la pila de marcos después del preámbulo de la **función B**.

Algorithm 1 Algoritmo del ejercicio 4

```
1: procedure FUNCION_A(a, b)
       a[5]
       ...
2:     procedure FUNCION_B(a, b, arreglo[0], arreglo[1], arreglo[2])
       ...
```

La pregunta es algo ambigua, la forma en que debería verse la pila cuando todavía no inicia la llamada a subrutina pero se está ejecutando A es la siguiente:
Suponiendo que el código de dicho algoritmo se tradujo a ensamblador con la convención descrita en [**Patterson**] los registros deberían lucir de la siguiente forma:

% ra	Lugar al que debe volver si A es subrutina
% fp	Apuntador al marco
% sp	Apuntador a la pila de A
% k0 - k1	Reservados para uso del OS
% t8 - t9	Valores temporales
%s0 - s7	Valores que deben permanecer en el mismo estado al concluir A
%t0 - t9	Valores almacenados por funcion_A
% a0, a1, a2, a3	a, b respectivamente
% v0 - v3	No ha concluido la subrutina, aún no hay valores de retorno
% at	Lugar mágico para el ensamblador
% zero	Constante cero
Marco de A	

2. Bibliografía

- Hennessy, J. Patterson, D. (2014). *Computer Organization and Design: The Hardware/Software Interface*. Quinta edición. Editorial Morgan Kaufmann. Estados Unidos de América.

- Universidad Tecnica Federico Santamaria (4 de noviembre 2019). MIPS Funciones. Recuperado de: http://profesores.elo.utfsm.cl/∼tarredondo/info/comp-architecture/paralelo2/C06_MIPS.pdf