# Lenguajes de Programación I

## Subrutinas - Pasaje de Parámetros

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"







## Subrutinas

- Mecanismo principal para abstracción de control.
  - Asocian un nombre a una secuencia de instrucciones.
  - Comportamiento varía según parámetros.
  - **Funciones** – cuando retornan valores.
    - **Puras** – No causan efectos de borde.
    - **Impuras** – Causan efectos de borde.
  - **Procedimientos** – cuando no retornan valores.
- Usualmente deben ser declaradas antes de ser utilizadas.
  - Declaraciones previas (*forward declarations*)
  - Múltiples pasadas del compilador para verificar tipos.





## Registro de Activación

### . . . ahora con más detalles

- **Registro de Activación** en la Pila de Ejecución por cada invocación.
  - Se arma en parte por el llamador y en parte por el llamado.
  - Se desarma cuando la rutina retorna.
- Construcción aprovecha características del hardware subyacente.
  - *Stack Pointer* – registro para apuntar al tope de la Pila de Ejecución.
  - *Frame Pointer* – registro para apuntar a una posición *fija* dentro del Registro de Activación, y usarla como referencia.
  - Instrucciones de máquina especiales simplifican el código generado – frecuentes en CISC, muy raro en RISC.
- Lenguajes dinámicos simulan parte o todo en el *heap*.





## Registro de Activación

### La importancia de ser *Frame Pointer*

- El *frame pointer* es la referencia central del registro de activación – todos los objetos se alcanzan con desplazamientos relativos.
- Objetos con *tamaño fijo*
  - Desplazamientos son conocidos a tiempo de compilación.
  - Ubicados cerca del *frame pointer* – ambos "lados".
  - Acceso directo a tiempo de ejecución.
- Objetos con tamaño variables – típicamente arreglos o cadenas.
  - Desplazamientos desconocido a tiempo de compilación.
  - Ubicados lejos del *frame pointer* – ambos "lados".
  - Acceso indirecto a tiempo de ejecución (*dope vector*).
  - Aplica para argumentos y variables – orientación diferente
    - Locales – *después* de la parte fija en el *llamado*.
    - Argumentos – *antes* de la parte fija en el *llamador*.

# Cadenas Dinámica y Estática

## Criterios de inclusión y dependencia de ejecución

- Cadena Dinámica – manifiesta el orden de llamadas
  - Conservando en cada Registro de Activación, el apuntador al anterior.
  - Responsabilidad del **llamado** a tiempo de **ejecución**.
  - Indispensable para alcance dinámico y manejo de excepciones.
- Cadena Estática – manifiesta el anidamiento de alcances.
  - Sólo necesaria en lenguajes que proveen anidamiento de rutinas.
  - Responsabilidad del **llamador** a tiempo de **compilación**.
  - Indispensable para alcance estático a objetos no locales.
  - Aprovecha acceso indirecto del procesador de ser posible.





# Mantenimiento del Registro de Activación

- Responsabilidad compartida entre llamador y llamado.
  - Llamador controla su espacio y manifiesta alcance.
  - Llamado controla su espacio y manifiesta el orden de llamada.
  - Algunas cosas pueden ser hechas por cualquiera de los dos – decisión del implantador según la plataforma.
- Tres fragmentos de código generados a tiempo de compilación:
  - **Secuencia de Llamada** – ejecutado por el llamador justo antes de transferir el control al llamado y justo después de recibirlo de vuelta.
  - **Prólogo** – ejecutado por el llamado justo al recibir el control.
  - **Epílogo** – ejecutado por el llamado justo antes de retornar el control.





# ¿Qué se hace en la Secuencia de Llamada?

## Justo antes de llamar

**(S.1)** Salvar en pila valores de registros que sean necesarios al retornar.
- No se puede suponer nada sobre lo que hará el llamado.
- Particularmente necesario en plataformas con pocos registros.

**(S.2)** Evaluar y empilar los parámetros actuales (argumentos) – los de tamaño variable van *antes* que los de tamaño fijo.

**(S.3)** Reservar espacio en la pila para valores de retorno.
- En algunas plataformas se usa un registro para esto – debe haber sido salvado en el paso (S.1).
- No hace falta si el llamado no retornará nada.

**(S.4)** Calcular y empilar el apuntador para mantener la Cadena Estática – sólo cuando tiene sentido en el lenguaje.

**(S.5)** Transferir el control a la Subrutina.
- Empilar la dirección de retorno antes saltar a la subrutina.
- En lo posible se aprovechan instrucciones especializadas.





# ¿Qué se hace en el Prólogo?

## Se acaba de recibir el control del llamador

**(P.1)** Empilar el valor actual del *frame pointer* – esto mantiene la cadena dinámica.

**(P.2)** Apuntar el *frame pointer* al "centro" del registro de activación.
- En general el tope actual de la pila – económico y simple.
- Si hay cadena estática, apuntar a esa posición – ahorra indirecciones.

**(P.3)** Reservar espacio en la pila para las variables locales
- Basta una modificación directa del *stack pointer* – sumar o restar.
- Aprovechar instrucciones especializadas – comunes en CISC.

**(P.4)** Empilar registros que sean utilizados en la subrutina – pareciera redundante después de (S.1), pero no lo es.

# ¿Qué se hace en el Epílogo?

**Justo antes de regresar el control al llamador**

**(E.1)** Colocar el valor de retorno en la posición adecuada.
- Si debe estar en la pila, quedará del lado del llamador.
- Si debe estar en un registro, simplemente se copia allí.
- No hace falta si el llamado no retornará nada.

**(E.2)** Desempilar los registros que fueron utilizados en la subrutina – aquellos que fueron salvados en (P.4).

**(E.3)** Liberar el espacio reservado para las variables locales
- Si se reservó modificando el *stack pointer*, efectuar la inversa.
- Si el *frame pointer* quedó apuntando al tope de la pila en (P.2), basta copiar su valor al *stack pointer*.

**(E.4)** Desempilar el *frame pointer* al registro de activación del llamador.

**(E.5)** Retornar el control al llamador.
- Desempilar la dirección de retorno y saltar para regresar.
- En lo posible se aprovechan instrucciones especializadas.






# ¿Qué resta en la Secuencia de Llamada?

**Justo al recibir el control de vuelta**

**(S.6)** Recuperar los valores de retorno.
- Tomarlos de la pila o del registro específico – liberar el espacio en el caso de la pila.
- No hace falta si el llamado no retorna resultados.

**(S.7)** Liberar el espacio donde se mantiene la cadena estática – sólo cuando tiene sentido en el lenguaje.

**(S.8)** Restaurar los valores de los registros salvados antes de la llamada.






# ¿Cómo se mantiene la Cadena Estática?

**Deducción simple a tiempo de compilación**

- A tiempo de compilación la Tabla de Símbolos permite saber:
  - El anidamiento de rutinas – "profundidad" de cada rutina.
  - Cuál rutina está llamando a cuál otra – propia (anidada) o ajena.
- Cuando el compilador genera código para una llamada, observar:
  - Cuando el *llamado* está anidado en el *llamador*:
    - Cadena Estática del *llamado* debe apuntar al *llamador*.
    - El compilador genera código para copiar el *frame pointer* del **llamador** como cadena estática del **llamado**.
  - Cuando el *llamado* está $k \geq 0$ alcances fuera del *llamador*.
    - $k$ alcances envuelven al *llamador* – y seguro están en la pila.
    - El compilador genera código para recorrer $k$ nodos de la cadena estática del **llamador** y copiar el valor final como cadena estática del **llamado**.






# Pasaje de Parámetros

- Permiten influir sobre el comportamiento de la subrutina.
- **Formales** – indicados simbólicamente en la definición

  ```
  proc foo(int bar, bool baz, float qux)
  ```

- **Reales** – pasados efectivamente a tiempo de ejecución

  ```
  foo(42,true,3.1415)
  ```

  También llamados *argumentos* o *actuales*.
- Invocación
  - Notación prefija en la mayoría de los lenguajes.
  - LISP asume que primer símbolo de expresión es la función.
  - Notación infija opcional (ML, Haskell).
  - Notación mezclada como en Smalltalk.

# Pasaje por Valor (*Call by Value*)

**La rutina recibe copias privadas**

- Implantación directa y simple.
  - Calcular el *valor* del parámetro actual.
  - Se copia el valor al espacio para el formal en el registro de activación.
  - En algunas circunstancias y plataformas benévolas, puede mejorarse usando un registro en lugar de espacio en la pila.
  - Se usa en la subrutina como si fuese una variable local.
- Ventaja: imposible que la rutina modifique los actuales.
- Desventaja: costoso en espacio y tiempo pasar objetos grandes.





# Pasaje por Referencia (*Call by Reference*)

**La rutina recibe apuntadores implícitos**

- Implantación directa y simple.
  - Calcular la *dirección* del parámetro actual.
  - Se copia la dirección al espacio para el formal en el registro de activación – es una *referencia implícita*.
  - Se usa en la subrutina como un *l-value*.
- Ventaja: no hay copia de valores ni requiere spacio adicional.
- Desventajas
  - Acceso más lento – compilador genera una indirección implícita.
  - Los actuales **deben** ser *l-values*.
  - Crean aliases.





# La diferencia...

```
x : integer
procedure foo( y : integer )
  y := 3;
  print x;
end
x := 2
foo(x)
print x
```

- Si `y` es pasada por **valor**, imprime `2 2`.
- Si `y` es pasada por **referencia**, imprime `3 3`.





# Variaciones en las Modalidades

**¿Por valor o por referencia?**

- Lenguajes que sólo pasan por valor – la mayoría.
  - Simular pasaje por referencia pasando un apuntador – particularmente útil para pasar estructuras y arreglos.
  - Casos especiales (arreglos en C).
- Lenguajes que sólo pasan por referencia – Fortran.
- Lenguajes que ofrecen ambos mecanismos – calificador `var` para parámetros en Pascal.

# Variaciones en las Modalidades

- **Parámetros Sólo-Lectura** (Modula-3, C)
  - Eficiencia de pasaje por referencia.
  - Seguridad de pasaje por valor.
  - Declaración del formal indica que no puede ser modificado.
  - Compilador verifica estáticamente la inmutabilidad, o genera código para verificarlo dinámicamente.
- **Dirección del Pasaje** (Ada, PL/SQL)
  - Entrada (`in`), salida (`out`) y ambos (`inout`).
  - La subrutina trabaja con copias locales.
  - `in` es equivalente a pasaje por valor.
  - `out` es denominado *llamada por resultado*.
  - `inout` es denominado *llamada por valor/resultado*.
  - El efecto de los cambios para un `inout` puede no ser inmediato.





# Clausuras

- En lenguajes que permiten pasar subrutinas como parámetros.
  - Debe pasarse la referencia al cuerpo de la subrutina – usualmente compartida entre varias invocaciones.
  - Debe pasarse el ambiente de referencia solamente si el lenguaje soporta anidamiento de subrutinas.
- Rutinario y simple en lenguajes funcionales
  - Los ambientes de referencia sólo interesan por sus *valores*.
  - . . . basta sacarles una copia al momento de pasar la subrutina.
- En lenguajes imperativos el ambiente de referencia incluye *referencias* a los objetos alcanzables al momento de construir la clausura.





# Parámetros de Propósito Especial

- **Arreglos con Forma Dinámica** (*Conformant Arrays*).
  - Cuando la forma del arreglo sólo se conoce a tiempo de ejecución.
  - Pasar una referencia a la base y las dimensiones.
- Valores por defecto para parámetros omitidos.
- Parámetros nombrados (*keyword parameters*) vs. posicionales.
- Número variable de argumentos.
  - Recibir todo como una lista y manipularla (LISP, Perl).
  - Utilizar macros/librerías para acceder a ellos en la pila de ejecución (C).
  - Obligar a que **todos** sean del mismo tipo (C#, Java).





# Valores de Retorno

- La mayoría de los lenguajes son restrictivos al respecto:
  - Un valor escalar.
  - ...que podría ser un apuntador.
- Lenguajes más flexibles permiten retornar elementos de tipos compuestos (registros y listas) o funciones.
- ¿Cómo retornarlo?
  - Instrucción explícita de retorno (`return`).
  - Valor de la última expresión evaluada.

# Bibliografía


- [*Scott*]
  - Secciones 8.1 a 8.3
  - Ejercicios y Exploraciones
- Página en Wikipedia sobre la Pila de Ejecución
- Página en Wikipedia sobre la Secuencia de Llamada
- Página en Wikipedia sobre Pasaje de Parámetros