Generación de código

Diseño de Compiladores

Dr. Víctor de la Cueva

vcueva@itesm.mx

Generación de código

- Hasta este punto ya estamos completamente seguros que nuestro programa está correctamente escrito.
- Estamos listos entonces para poder generar código.
  - Un compilador debería generar código binario, listo para correr en la máquina (e.g. .exe o .o).
    - Depende completamente de la máquina donde se esté corriendo.
  - También podría generar código en ensamblador listo para poder correrse con un linker para un procesador específico.
    - Cada procesador tiene su propio lenguaje ensamblador.
  - Una alternativa intermedia podría ser la generación de código especial (intermedio) para una máquina virtual (e.g. código de tres direcciones o código P, como Java).

Nuestra alternativa

- En nuestro proyecto vamos a optar por la opción de ensamblador para un procesador específico.
- Lo más común es usar un procesador MIPS, que utiliza una arquitectura RISC y que es muy común en los procesadores actuales.
  - En realidad, vamos a utilizar un simulador de MIPS llamado SPIM (la versión más reciente es QtSpim http://spimsimulator.sourceforge.net/)
  - Sin embargo, si alguien desea generar código para otro procesador, lo puede hacer sin ningún problema (indicarlo en la documentación).

Organización en Runtime

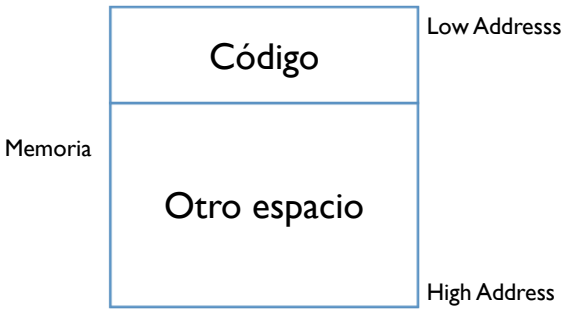
- Antes de ver la generación de código debemos iniciar con la organización de nuestro programa en runtime:
  - Debemos entender qué estamos tratando de generar antes de ver cómo generarlo.
- Hay varias técnicas para generar código ejecutable:
  - La parte fundamental para lograrlo es la administración de los recursos de memoria en runtime.
- Consiste en entender cómo se organiza la memoria en tiempo de ejecución.

En el inicio

- Inicialmente, el sistema operativo (OS) es lo único que está corriendo en la máquina:
  - La ejecución de un programa está inicialmente bajo el control del OS.
- Cuando un programa es invocado:
  - El OS asigna espacio para el programa
  - El código es cargado en parte de ese espacio
  - El OS salta al punto de entrada (i.e. "main") y el programa se deja solo y corriendo

La memoria en el inicio

- Cuando el OS inicia la ejecución de un programa compilado la memoria se puede ver así:



Por tradición, los dibujos de la organización de la memoria en la computadora tienen:
- Low Address arriba
- High Address abajo
- Líneas delimitando áreas para diferentes tipos de datos

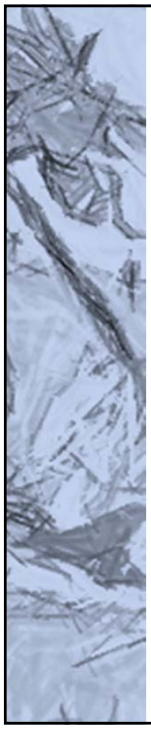
Simplificación

- Los esquemas de la memoria siempre son simplificaciones:
  - No toda la memoria debe estar contigua
  - Por ejemplo, cuando es un sistema de memoria virtual

El compilador en la organización

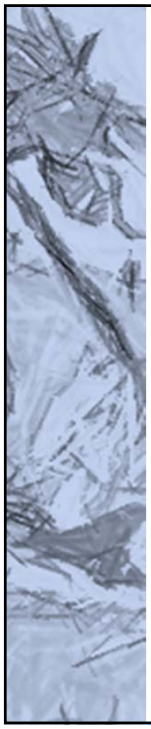
- El compilador, desde el punto de vista de la organización, es el responsable de:
  - Generar código
  - Orquestar el uso del área de datos

Otro espacio = Datos



Layout de los datos

- El compilador es el responsable de decidir cómo será el layout de los datos y entonces generar código que manipule correctamente estos datos.
- Hay diferentes tipos de datos que van en esta área.



Objetivos de la generación de código

- Hay dos objetivos principales en la generación de código:
  - Correctéz: El código debe implementar correctamente el programa del usuario.
  - Velocidad: El código debe ser eficiente y correr rápido.
- Es fácil implementar estas dos cosas por separado.
- La complicación en la generación de código vienen al tratar de ser rápido y correcto al mismo tiempo:
  - La solución ha sido generar frameworks muy elaborados que nos dicen cómo generar código y las correspondientes estructuras de runtime deberían ser hechas para lograr ambas metas

Asunciones

- Hay dos asunciones sobre el tipo de lenguaje de programación para el cual estamos generando código:
  - La ejecución es secuencial: el control se mueve de un punto a otro del programa en un orden bien definido
  - Cuando un procedimiento es llamado el control siempre regresa al punto inmediato siguiente después de la llamada
- Algunas características de algunos lenguajes violan estas asunciones, por ejemplo:
  - Concurrencia
  - Excepciones
- Los procedimientos básicos de generación de código funcionan también para estos lenguajes.

Activaciones

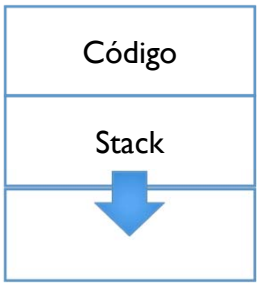
- Una invocación de un procedimiento P es una activación de P.
- El tiempo de vida (*lifetime*) de una activación de P es:
  - Todos los pasos para ejecutar P
  - Incluyendo todos los pasos de las llamadas a procedimientos que haga P
- Es decir, todos los pasos desde que P es llamado hasta que P regresa (*return*).
- El *lifetime* de una variable X es la parte de la ejecución en la cual X está definida.
- Notar que:
  - *Lifetime* es un concepto dinámico (*runtime*)
  - *Scope* es un concepto estático (compilación)

Lifetime de las activaciones

- Cuando un procedimiento P llama a un procedimiento Q, entonces, Q regresa antes de que P regrese.
- Es decir, los *lifetimes* de las activaciones están correctamente anidadas.
- Esto implica que los *lifetimes* de las activaciones pueden ser ilustradas con un árbol. AA 4.3
- Debido a que las activaciones están anidadas, un stack puede mantener los procedimientos activos. AA 4.3.5
  - El stack se puede guardar en el espacio de datos

Stack

- Se puede usar el espacio de datos para mantener el stack.



Activation Records

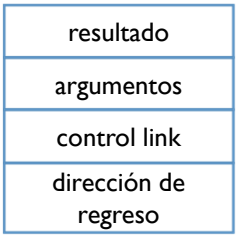
- La información necesaria (que se debe mantener) para manejar una activación de procedimiento es llamada *Activation Record* (AR) o *Frame*.
- Si el procedimiento F llama al procedimiento G entonces el AR de G contiene una mezcla de información de F y G.
  - F es "suspendida" hasta que G se complete, punto en el cual F continua (resumes).
  - El AR de G contiene información necesaria para:
    - Completar la ejecución de G
    - Continuar la ejecución de F

Diseño del AR

- No existe un estándar para diseñar un AR adecuado.
- Lo más importante del diseño de un AR es que logre mantener suficiente información en él para generar el código adecuado para ejecutar correctamente el procedimiento que está siendo llamado (*callee*), y continuar la ejecución del procedimiento llamador (*caller*).

Ejemplo de AR

- Un diseño del AR para una función podría ser:



Resultado: Contienen el valor que regresa la función después de que termina.
Argumentos: Es una porción para mantener los valores de los argumentos con los que se llamó a la función.
Control link: Un apuntador al AR del invocador.
Dirección de regreso: La dirección de memoria a la que tenemos que saltar al terminar la ejecución de la función

AA 4.4 y 4.4.5

Comentarios

- La ventaja de poner el valor de *return* en la primera posición de un *frame* es que el llamador puede encontrarlo con un *offset* fijo a partir de su propio *frame*.
- Sobre esta organización:
  - Puede arreglarse el orden de los elementos del *frame*
  - Puede dividir de forma diferente las responsabilidades entre el *caller* y el *callee*.
  - Una organización es mejor si mejora la velocidad de ejecución o simplifica la generación de código.
- Los compiladores reales mantienen la mayor parte del *frame* (hasta donde sea posible) en los registros
  - Especialmente el resultado y los argumentos

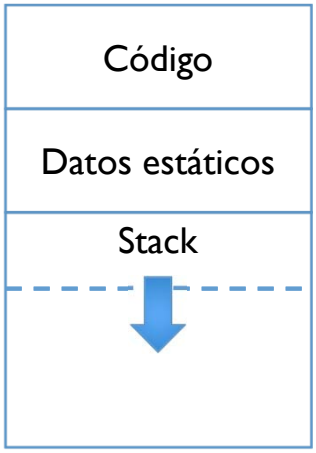
Resumen

- El compilador debe determinar, en tiempo de compilación, la distribución de los ARs y generar el código que accede correctamente las posiciones de los datos en el AR.
- Esto es, la distribución del AR y el generador de código deben diseñarse juntos.

Variables Globales

- Todas las referencias a una variable global apuntan al mismo objeto:
  - Por esta razón no se puede almacenar una variable global en un AR
  - A las globales se les asigna una dirección fija, una sola vez:
    - Son valores estáticamente asignadas
    - El compilador decide dónde van a vivir durante toda la ejecución del programa
- Dependiendo del lenguaje podría haber otros valores asignados estáticamente.

Nuevo Esquema



El Heap

- Un valor que permanece vivo fuera del procedimiento que lo crea no puede mantenerse en el AR:

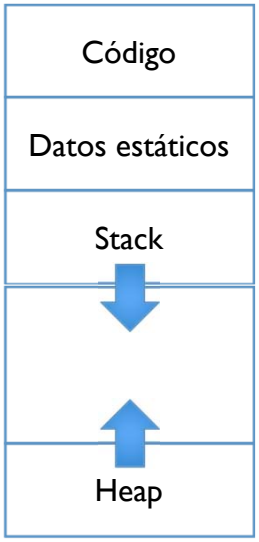
```
method foo() { new Bar }
```

El valor Bar debe sobrevivir la *deallocation* del AR de foo.

- Los lenguajes con datos dinámicamente asignados (*allocated*) usen un *heap* para almacenar los datos dinámicos.
- Una implementación normal permite que el *heap* crezca.

Stack y Heap

- Muchas implementaciones de lenguajes usan tanto un heap como un stack donde ambos pueden crecer.
- Se debe tener mucho cuidado en que ninguno crezca sobre el otro:
  - Solución: Iniciar el *Heap* y el *Stack* en posiciones opuestas de la memoria y dejarlos que crezcan uno hacia el otro.
  - Si sus límite se encuentran el programa se quedó sin memoria y se debe tomar una acción (e.g. abortar, pedir más memoria al OS)



Tipos de datos

- Podemos hacer un resumen de los diferentes tipos de datos con los que las implementaciones de los lenguajes tienen que lidiar:
  - El área de código contiene código objeto
    - Para muchos lenguajes es de tamaño fijo y sólo de lectura
  - El área estática contiene datos con direcciones fijas (e.g. variables globales)
    - De tamaño fijo, puede ser de lectura y escritura
  - El *Stack* contiene AR para procedimiento activo
    - Cada AR es normalmente de tamaño fijo, contiene variables locales
  - El *Heap* contiene todos los otros datos
    - En C, el *heap* es manejado por `malloc` y `free`
    - En Java, por `new` y el *garbage collector*

Alineamiento

- Es un detalle de muy bajo nivel pero muy importante para que los escritores de compiladores lo conozcan.
- La mayoría de las máquinas modernas son de 32 a 64 bits (en una palabra):
  - 8 bits en un byte
  - 4 a 8 byte por palabra
  - Las máquinas son direccionables por byte o por palabra
- Los datos son alineados por palabra si éste inicia en el límite de una palabra.

Stack Machines

- Es el modelo más simple para generar código.
- En estas máquinas, el almacenamiento principal es una clase de *stack*:
  - Sólo se almacena en el *stack*
  - Una instrucción $r = F(a_1, \ldots, a_n)$:
    - Pops n operandos del *stack*
    - Realiza la operación F usando los operandos
    - Pushes el resultado r al *stack*

AA 4.7

- El *stack* tiene una propiedad muy importante, lo que está debajo de la expresión evaluada se mantiene sin cambio.

¿Cómo podemos programar una *stack machine*?

- Considerar dos instrucciones:
  - push i: *push* el entero i en el *stack*
  - add: suma dos enteros que saca del *stack*
- Un programa sería:
  - Push 7
  - Push 5
  - Add
  - Push resultado

Propiedades

- La localización de los operandos/resultado en la memoria no está explícitamente dada en la instrucción
  - Siempre está en el top del *stack*
- En contraste con una máquina de registros:
  - `Add` en lugar de `add r1, r2, r3`
  - Da programas más compactos (es una de las razones de porque Java utiliza un modelo de evaluación de *stack*).
- Sin embargo, las máquinas de registros son más rápidas.

Intermedio

- Hay un intermedio entre una máquina de *stack* y una máquina de registros:
  - Una máquina de *stack* de n-registros
    - Conceptualmente, mantiene n datos del top del *stack* en registros de la máquina y el resto en el *stack*
  - Un caso especial es una máquina de stack de 1-registro
    - El registro es llamado Acumulador

Ventaja de una máquina de 1-registro

- Una máquina de *stack* pura:
  - Una instrucción add requiere 3 operaciones de memoria
  - Dos lecturas y una escritura
- Una máquina de *stack* de 1-registro el add se hace:
  - acc ← acc + top
  - Sólo un acceso a memoria

Estrategia general para evaluar una expresión

- Considere una expresión op(e_1, …, e_n)
  - Note que e_1, …, e_n son subexpresiones
- Para cada e_i (0 < i < n)
  - Calcule e_i → resultado en el acumulador
  - Push el resultado al *stack*
- Pop *n-1* valores del *stack* (e_n continúa en el acc) y calcule op
- Almacene el resultado en el acumulador

AA 4.8.5

GENERACIÓN DE CÓDIGO

Enfoque

- Nos enfocaremos en generar código para una máquina de stack con acumulador
  - No genera un código muy eficiente pero es fácil de implementar para el proyecto
- El resultado lo vamos a correr en una máquina real
  - Procesador MIPS (o simulador SPIM)
  - Simularemos instrucciones para una *stack machine* usando instrucciones y registros MIPS.
- MIPS es una arquitectura de 32 bits (4 bytes por palabra)
  - Es alineado por byte, pero se puede cambiar.

Arquitectura MIPS

- Es una arquitectura un poco vieja.
- Es el prototipo de la maquina *Reduce Instruction Set Computer* (o máquina RISC).
  - La idea detrás de la máquina RISC fue tener un relativamente simple conjunto de instrucciones
- La mayoría de las operaciones usan registros para operandos y resultados.
- Usan instrucciones de *load* y *store* para manejar los datos en memoria.
- Tiene 32 registro de propósito general de 32 bits cada uno
  - Usaremos `$sp`, `$a0`, `$fp` y `$t1` (un registro temporal)
- Leer la documentación de SPIM (simulador de MIPS) para más detalles.

Registros

Register Number	Alternative Name	Description
0	zero	the value 0
1	$at	(assembler temporary) reserved by the assembler
2-3	$v0 - $v1	(values) from expression evaluation and function results
4-7	$a0 - $a3	(arguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	$t0 - $t7	(temporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	$s0 - $s7	(saved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	$t8 - $t9	(temporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.
26-27	$k0 - $k1	reserved for use by the interrupt/trap handler
28	$gp	global pointer. Points to the middle of the 64K block of memory in the static data segment.
29	$sp	stack pointer Points to last location on the stack.
30	$s8/$fp	saved value / frame pointer Preserved across procedure calls
31	$ra	return address

En un buen resumen: http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

Acumulador

- La primera decisión es dónde va a estar el acumulador:
  - Será mantenido en el registro de MIPS llamado `$a0`
  - El *stack* se mantiene en memoria
    - El *stack* crece hacia *lower addesses* (cambio con respecto al dibujo)
    - Es una convención estándar de MIPS
  - La dirección de la siguiente localización vacía en el stack se mantiene en el registro `$sp` de MIPS (estándar para *stack pointer*)
    - El *top* de *stack* está en la dirección `$sp + 4`

NOTA: Algunas implementaciones establecen que el *stack* apunta a la última posición ocupada, por lo que el top del *stack* estaría directamente en `$sp`.

(e.g. en tutorial en http://chortle.ccsu.edu/assemblytutorial/Chapter-25/ass25_3.html)

Generación de código práctica

L 401

- Las técnicas estándar de generación de código involucran modificaciones de los recorridos posorden del AST.
- El algoritmo básico puede ser descrito con el siguiente procedimiento recursivo (para nodos de árbol con máximo dos hijos pero fácilmente extensible a más):

```
procedure genCode(T: treenode);
  if T no es nil then
     genere código para preparar el caso del código del hijo izquierdo de T;
     genCode(hijo izquierdo de T);
     genere código para preparar el caso del código del hijo derecho de T;
     genCode(hijo derecho de T);
     genere código para implementar la acción de T;
end
```

genCode( )

- El procedimiento central para la generación de código es genCode( ), el cual, al implementarlo, se convierte en un gran *SWITCH*, con una opción para cada tipo de nodo.
- Cada tipo de nodo podría requerir un orden diferente de recorrido:
  - Un nodo + (*Plus*) requiere sólo procesamiento postorden
  - Un nodo = (*Assign*) requiere cierto procesamiento tanto preorden (si la variable está en mismo nodo) como posorden
- De este modo, las llamadas recursivas pueden no escribirse del mismo modo para todos los casos.

Un ejemplo simple

- Usemos una gramática para expresiones muy simples:

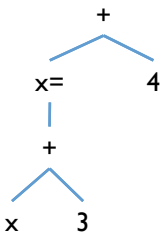
  exp → ***id*** **=** *exp* | *aexp*

  aexp → *aexp* **+** *factor* | *factor*

  factor → **(** *exp* **)** | ***num*** | ***id***

- Veamos el ejemplo de la expresión: (x = x + 3) + 4

Su AST es:



L 411
Código en Word

Código en MIPS

- Para generar código para MIPS vamos a requerir algunas instrucciones MIPS, y vamos a poderlo lograr con un pequeño número de ellas.
- Las 5 básicas son:
  - Load Word: `lw reg1 offset(reg2)`
    - Carga una palabra de 32 bits de la dirección de memoria reg2+offset en el reg1
  - Add: `add reg1 reg2 reg3`
    - reg1 ← reg2 + reg3
  - Store Word: `sw reg1 offset(reg2)`
    - Almacena una palabra de 32 bits que está en reg1, en la dirección reg2+offset
  - Add immediate unsign: `addiu reg1 reg2 imm`
    - reg1 ← reg2 + imm ("u" significa que el *overflow* no se checa)
  - Load immediate: `li reg imm`
    - reg ← imm

Ejemplo: 7 + 5

acc ← 7	`li $a0 7`
push acc	`sw $a0 0($sp)`
	`addiu $sp $sp -4`
acc ← 5	`li $a0 5`
acc ← acc + top	`lw $t1 4($sp)`
	`add $a0 $a0 $t1`
pop	`addiu $sp $sp 4`

Para restaurar el invariante de que el stack pointer apunta a la siguiente posición libre, se tiene que restar un 4 al stack pointer ya que el stack crece hacia las direcciones bajas de memoria.

Gramática de ejemplo para CG

- Veamos la generación de código para un lenguaje de más alto nivel que un simple lenguaje de *stack-machine*.
- Un lenguaje con enteros y operaciones con enteros, su gramática es:

  $P \rightarrow D$ **;** $P \mid D$

  $D \rightarrow$ **def id (** *ARGS* **) =** E **;**

  ARGS $\rightarrow$ **id ,** *ARGS* | **id**

  $E \rightarrow$ **int** | **id** | **if** $E1$ **=** $E2$ **then** $E3$ **else** $E4$

  | $E1$ **+** $E2$ | $E1$ **–** $E2$ | **id (** $E1$**, … ,** En **)**

  - La primera función definida es el punto de inicio (la rutina "main")

Generación de código (CG)

- Para cada expresión **e** vamos a generar código MIPS que:
  - Calcule el valor de **e** y lo deje en `$a0`
  - Preserve el `$sp` y el contenido del *stack* (anterior a la expresión)
- Definimos una función para generación de código **`cgen(e)`**, que recibe una expresión y como resultado obtiene el código generado para evaluar **e**.
- Nuestra función de generación de código va a trabajar por casos:
  - Generar cierto tipo de código para cada tipo de expresión en el lenguaje

Código de color y otras convenciones

- Color:
  - ROJO: en tiempo de compilación
  - VERDE: en tiempo de ejecución (*runtime*)
- `print`:
  - Una instrucción en verde representa código MIPS
  - Este código se tiene que imprimir a un archivo
  - Por lo tanto, cuando se observe algo así:
    - `lw $t1 4($sp)`
  - En realidad debería ser un `print`:
    - `print("lw $t1 4($sp)")`
    - Se omitirá para no hacer muy grande el código presentado.

Casos de código

- Código para evaluar una constante: simplemente se copia al acumulador:
  - `cgen(i) = li $a0 i`
- Código para la suma de dos expresiones:
  - `cgen(e1 + e2) =`

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

Template

- El ejemplo anterior muestra dos características muy importantes del *code-generation*:
  - El código para la suma + es un *template* con "agujeros" para el código para evaluar las expresiones e1 y e2:

```
code(e1)
…
code(e2)
…
```

  - Y es lo mismo para todos los otros tipos de expresiones
  - La generación de código para una *stack-machine* es recursivo:
    - El código para e1+e2 es el código para e1 y para e2 pegado
- El CG puede ser escrito como un *recursive-descent* del AST.
  - Al menos para expresiones

Resta

- Nueva instrucción: sub reg1 reg2 reg3
  - Implementa reg1 ← reg2 – reg3
- El código para una expresión de resta es:
  - cgen(e1 – e2) =

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
sub $a0 $t1 $a0
addiu $sp $sp 4
```

Es el mismo que para la suma de expresiones excepto por la instrucción sub

Ejecicio AA 4.11.5

Control de flujo

- Para realizar el código para algunas expresiones se van a requerir instrucciones de control de flujo:
  - *Branch to label*: `beq reg1 reg3 label`
  - *Unconditional jump to label*: `b label`
- Las cuales hacen que el código "salte" a otro lugar de la memoria para continuar la ejecución.

Código para if-then-else

- cgen(if e1 = e2 then e3 else e4) =

```
        cgen(e1)
        sw $a0 0($sp)
        addiu $sp $sp -4
        cgen(e2)
        lw $t1 4($sp)
        addiu $sp $sp 4
        beq $a0 $t1 true_branch
false_brach:
        cgen(e4)
        b end_if
true_brach:
        cgen(e3)
end_if:
```

Secuencia de activación de una función

- Antes de continuar, recordemos la secuencia de llamada a una función comprende, aproximadamente, los siguientes pasos: L 359
  - Calcula los argumentos y los almacena en sus posiciones correctas en el nuevo AR (esto se logra insertándolos en orden inverso en la pila)
  - Almacena (inserta) el fp como el vínculo de control del nuevo AR
  - Cambia el fp de manera que apunte al inicio del nuevo AR (esto se consigue copiando el sp en el fp en este punto)
  - Almacena la dirección de retorno en el nuevo AR (si es necesario)
  - Realiza un salto hacia el código del procedimiento a ser llamado
- Cuando un procedimiento sale de escena:
  - Copia el fp al sp
  - Carga el vínculo de control en el fp
  - Realiza un salto hacia la dirección de retorno
  - Cambia el sp para extraer los argumentos

Llamada y definición de funciones

- El código para llamada y definición de funciones depende completamente del *layout* que se decidió utilizar para el AR.
- Un muy simple AR, suficiente para este lenguaje:
  - El resultado está siempre en el acumulador
    - No se requiere almacenar el resultado en el AR
  - El AR guarda los parámetros reales con los que se llama la función
    - Para `f(x1, …, xn)` *push* `x1, …, xn` en el *stack*
    - Estas son las únicas variables en este lenguaje

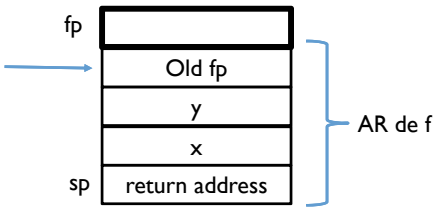
Cont…

- El funcionamiento del *stack* garantiza que cuando una función termina el `$sp` es el mismo que cuando la función inició.
  - En realidad, no se requiere un *control link* en el AR para este lenguaje
  - La idea de un *control link* es que nos ayude a encontrar la activación previa, pero, como el *stack pointer* es preservado (al terminar la llamada) no tendrá ningún problema para encontrar el AR previo cuando regrese de la llamada a la función.
  - Por otro lado, no se requerirá buscar en otra activación durante la llamada a la función porque no existen (para este lenguaje) variables que no sean locales.
- Sí se requiere la dirección de regreso (*return address*).
- También será útil un apuntador a la activación actual.
  - Este *pointer* vivirá en un registro `$fp` (*frame pointer*)

El AR a ser usado

- Para este lenguaje es suficiente un AR con:
  - Un apuntador al *frame* del *caller* (*control link*)
  - Los parámetros actuales
  - Y la dirección de regreso
- Considere una llamada a la función `f(x,y)`, su AR se vería así, exactamente en el momento de la llamada, antes de la ejecución:

Guardamos el fp del AR anterior porque el fp va a tener la dirección del nuevo y lo requerimos para restaurarlos cuando nuestra función termine.



Nueva instrucción

- Nueva instrucción *jump and link*: `jal label`
  - Salta al `label`, guarda la dirección de la siguiente instrucción en el registro `$ra` (*return address*)
  - En otras arquitecturas, la dirección de retorno es almacenada en el *stack* por la instrucción "call".
    - Ejemplo

```
jal L
add …
```

La dirección donde se encuentra el `add` es guardada en el `$ra`.

Llamada a una función

- Estamos listos para generar código para una expresión de llamada a una función:
  - La secuencia de llamada a una función está formada por las instrucciones (de ambos, el *caller* y el *callee*) para preparar una invocación a la función.
  - Será necesario generar un código para la parte del *caller* y otro para la parte del *callee*.

Lado del *caller*

```
cgen(f(e1, …, en)) =
    sw $fp 0($sp)
    addiu $sp $sp -4
    cgen(en)
    sw $a0 0($sp)
    addiu $sp $sp -4
    …
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    jal f_entry
```

- El *caller* guarda su *frame pointer* valor en el *stack* (*old fp*)
- El *caller* guarda los parámetros reales en orden inverso
- Finalmente, el *caller* guarda la *return address* en el registro `$ra`
- El AR, hasta aquí, es de tamaño `4*n + 4` bytes (n es el número de argumentos).

Lado del *callee* en la secuencia de llamado

```
cgen(def f(x1, …, xn)=e) =
    f_entry:
        move $fp $sp
        sw $ra 0($sp)
        addiu $sp $sp -4
        cgen(e)
        lw $ra 4($sp)
        addiu $sp $sp z
        lw $fp 0($sp)
        jr $ra
```

No se conoce la *return address* hasta después de que se ejecuta la llamada, por eso el calle debe guardar este valor.

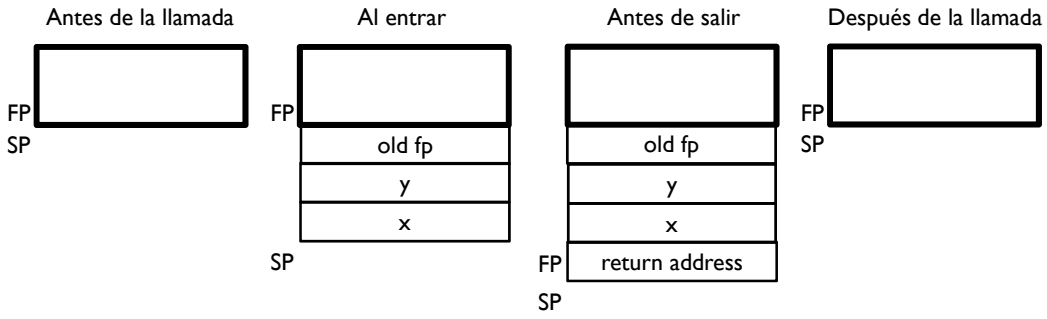
Regresamos la *return address* al registro `$ra`

Retauramos el apuntador al *old frame*

- Nueva instrucción *jump register*:

  `jr reg`
  - Salta a la dirección en el registro `reg`
- Note que el *frame pointer* apunta al *top* del frame (no al *bottom*)
- El *callee* pops la *return address*, los argumentos reales y el valor guardado del *frame pointer*
- `z = 4*n + 8`
  - 2 palabras (8) para el *return address* y el apuntador al *old frame*
  - *n* argumentos, cada uno de 4 bytes

Gráficamente



Referencias a variables

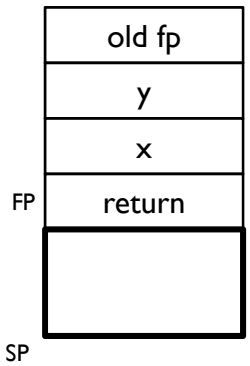
- Las "variables" de una función son solamente sus parámetros.
  - Están en el AR
  - Fueron colocados (*push*) por el *caller*
- Problema: debido a que el *stack* crece cuando se salvan los resultados intermedios, las variables no están a un *offset* fijo del `$sp`.
- Solución: usar un *frame pointer*
  - Siempre apunta a la dirección de regreso en el *stack*
  - Debido a que este no se mueve, puede ser usado para encontrar las variables

Código para referencia a variables

- Sea x_i el i-ésimo (i = 1, …, n) parámetro formal de una función para la cual se está generando código:

  `cgen(`x_i`) = lw $a0 z($fp)`

  Donde `z = 4*i`
- Para una función `def f(x,y)=e` la activación y *frame pointer* son preparados como sigue:
  - `x` es colocado en `fp+4`
  - `y` es colocado en `fp+8`



Variables y la ST

- Es muy importante recordar que la tabla de símbolos tiene información que puede ser útil para generar código… ¡y ya la tenemos!
- Podemos modificarla las veces que haga falta con tal de que guarde información que nos sea útil.
  - Una posibilidad es guardar las direcciones (en realidad los *offsets*) donde se encuentran las variables en la memoria (sobre todo las globales) y los valores que van tomando.
  - Si se usan las ST para esta tarea se debe tener cuidad de cargar el *scope* adecuado a la variable en cuestión.

Print y Write en MIPS

- Las instrucciones para impresión y lectura (y muchas otras, como la solicitud de memoria dinámica al *heap*) se manejan en MIPS como una llamada al sistema por medio de la instrucción `syscall`.
- Para distinguir qué es lo que se quiere hacer, se debe colocar cierta información en ciertos registros antes de hacer la llamada al sistema, de acuerdo con lo que se indica en el manual de MIPS para llamadas al sistema (ver siguiente *slide*)

Llamadas al sistema

Service	Code in $v0	Arguments	Results
print_int	1	$a0 = integer to be printed	
print_float	2	$f12 = float to be printed	
print_double	3	$f12 = double to be printed	
print_string	4	$a0 = address of string in memory	
read_int	5		integer returned in $v0
read_float	6		float returned in $v0
read_double	7		double returned in $v0
read_string	8	$a0 = memory address of string input buffer $a1 = length of string buffer (n)	
sbrk	9	$a0 = amount	address in $v0
exit	10		

Tomado de http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

Al inicio de un programa

- Cuando se corre un programa en ensamblador, el OS decide en qué posición de la memoria se debe cargar el código.
  - Normalmente, tiene un espacio específico para hacerlo pero hay cosas que pueden variar porque no todos los códigos son del mismo tamaño.
- Por esta razón, todas las referencias a memoria se deben hacer con *offsets* a partir de un valor dado.
  - Para facilitar esta forma de crear programas, el OS coloca cierta información a ciertos registros al momento de la carga:
    - En el `$sp` pone la dirección del siguiente byte libre del *stack*
    - En el `$gp` pone la dirección donde guarda las variables globales
    - El `$pc` lo pone en 0
    - Etc.

El Stack y el Heap

- Al iniciar nuestro programa el `$sp` contiene un apuntador a la siguiente posición libre de la región destinada al *stack*.
  - Esto permite que nuestros programa los podamos hacer con referencia al `$sp` y de esta forma no importa dónde lo cargue.
- El *heap* es una región que también crece y decrece pero que se encuentra en el lado opuesto de la memoria de datos del programa (crece hacia el *stack* y ésta hacia el *heap*).
  - Para manejar el *heap* no necesitamos un apuntador inicial ya que la memoria dinámica siempre es manejada por medio del OS
  - Para solicitar espacio dinámico se hace una llamada al sistema (`syscall`)
  - La memoria liberada se traba igual en MIPS pero SPIM no tiene esa función (sólo es un simulador)

Sumario

- El AR debe ser diseñado junto con el generador de código.
- La generación de código puede ser hecha por medio de un recorrido recursivo del AST (como el *type-checking*).
- Se recomienda usar una *stack-machine* para su proyecto ya que es muy simple.

Ejemplo de CG

Generar código para el siguiente programa:

```
def sumto(x) = f x=0 then 0 else x+sumto(x-1)
```

Solución

```
sumto_entry:
      move $fp $sp
      sw $ra 0($sp)
      addiu $sp $sp -4
      lw $a0 4($fp)
      sw $a0 0($sp)
      addiu $sp $sp -4
      li $a0 0
      lw $t1 4($sp)
      addiu $sp $sp 4
      beq $a0 $t1 true1
false1:
      lw $a0 4($fp)
      sw $a0 0($sp)
      addiu $sp $sp -4
      sw $fp 0($sp)
      addiu $sp $sp -4
      lw $a0 4($fp)
      sw $a0 0($sp)
      addiu $sp $sp -4
      li $a0 1
      sub $a0 $t1 $a0
      addiu $sp $sp 4
      sw $a0 0($sp)
      addiu $sp $sp -4
      jal sumto_entry
      lw $t1 4($sp)
      add $a0 $t1 $a0
      addiu $sp $sp 4
      b endif1
true1:
      li $a0 0
endif1:
      lw $ra 4($sp)
      addiu $sp $sp 12
      lw $fp 0($sp)
      jr $ra
```

Dudas

- Con qué valor se inicializa el stack en MIPS?
- Realmente está apuntando a la siguiente posición vacía?
  - Por eso, si se toma la convención de que apunta a la última posición llena sólo se desperdicia un byte.
- El control link es lo mismo que el old fp?
  - ¿Por qué en la lectures AA dice que no se requiere?
- ¿A dónde apunta el fp?
  - Apunta al top de frame, es decir, un byte antes del sp final, con la diferencia de que el fp siempre se mantiene fijo

Referencias

- Alex Aiken. Compilers. Stanford Online (2018).
  - https://lagunita.stanford.edu/courses/Engineering/Compilers/Fall2014/about
- K.C. Louden. *Contrucción de Compiladores: principios y práctica*. Thomson (2004).