Departamento de Cs. e Ingeniería de la Computación
Universidad Nacional del Sur

Compiladores e Intérpretes

Máquina Virtual de Compiladores e Intérpretes (CeIVM) [1]

Segundo Cuatrimestre de 2014

1. Generalidades

Nuestro objetivo final es construir un compilador para MiniJava, esto es, construir un programa que traduzca un programa fuente en MiniJava a un programa objeto en un lenguaje de máquina determinado. Antes de emprender la tarea de escribir un compilador, debemos establecer precisamente una traducción para cada construcción de MiniJava. Traducir a un lenguaje máquina de una computadora real es, en general, una tarea muy trabajosa. Los lenguajes máquina tienen muchas características a tener en cuenta, las cuales dificultan establecer la correspondencia entre las construcciones del programa fuente y sus traducciones. Entonces, en lugar de traducir los programas de MiniJava a un lenguaje máquina real, se definirá una máquina hipotética más conveniente para nuestra finalidad. Esta forma de tratar los problemas es muy común en la implementación de un lenguaje de programación. La máquina hipotética así definida se llama un sistema de ejecución para el lenguaje. Supondremos, por lo tanto, que este sistema de ejecución será luego programado en un lenguaje de máquina real, facilitando la traducción.

Las próximas secciones estarán dedicadas a la especificación de una máquina hipotética que denominaremos CeIVM (Máquina Virtual de Compiladores e Intérpretes). Dicha máquina consiste en un intérprete capaz de cargar, ensamblar y ejecutar programas especificados en un lenguaje de ensamblado que denominaremos CeIASM. Este lenguaje cuenta con un set de instrucciones (denominado CeIISA) y un conjunto de directivas que se describirán a lo largo de esta especificación. La presentación de la máquina será gradual, y se complementará con la discusión de varias de las construcciones de MiniJava.

2. Características Generales de la CeIVM

Describiremos en esta sección la estructura básica de la CeIVM. La máquina tiene una arquitectura de pila. Esto facilita la generación de código del compilador ya que permite procesar las expresiones mediante traducciones a notación posfija evitando, de esta manera, tener que gestionar un banco de registros generales. Además, esta arquitectura resulta muy natural para implementar la dinámica de los métodos de MiniJava, cuyos registros de activación presentan un comportamiento de Pila.

La CeIVM consta de un espacio lineal de direcciones M (que va de 0 a un máximo preestablecido) de locaciones de memoria enteras de 32 bits signados. Toda referencia a una dirección

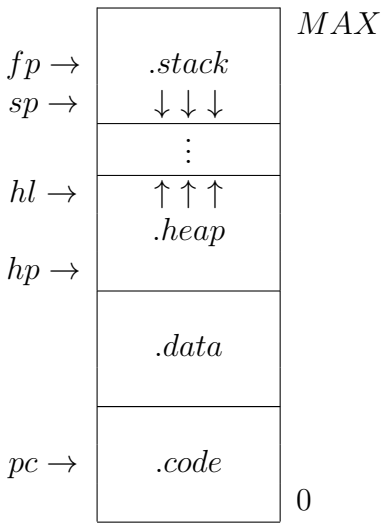
[1] Esta especificación ha sido diseñada en base a la documentación de MEPa utilizada en Compiladores e Intérpretes hasta el año 2010 inclusive. Entre otros numerosos cambios se le ha añadido soporte para la implementación de lenguajes orientados a objetos. Realizado por Sebastián Escarza, modificado por Sebastian Gottifredi - CeI - DCIC - UNS - 2011-2014.

i dentro de este espacio de memoria se notará mediante $M[i]$ en la especificación.

En el espacio de memoria M se definen las cuatro secciones de memoria en las que se organizará el ambiente de tiempo de ejecución:

- La sección de programa *.code* que contendrá las instrucciones del CeIISA que se ejecutarán. Esta región toma las locaciones más bajas y progresa en sentido creciente. Las instrucciones en memoria, luego de ensambladas, se representan por su Opcode seguido por sus argumentos (si existen), ocupando una celda cada uno.
- La sección de datos *.data* que contendrá los datos estáticos del programa. Esta región se define a continuación de *.code* y contigua a esta.
- La sección *.heap* destinada a mantener los datos y las estructuras necesarias para alojarlos y liberarlos dinámicamente.
- La sección de pila de datos *.stack* que contendrá los valores que serán manipulados por las instrucciones del CeIISA.

Las secciones *.heap* y *.stack* coexisten en un área de memoria compartida en la cual el *.heap* utiliza las direcciones de memoria más bajas y crece hacia las altas, y el *.stack* parte de las direcciones más altas y progresa de manera decreciente. La organización de las secciones de memoria se puede observar en la siguiente figura:



La CeIVM cuenta con los siguientes registros especiales algunos de los cuales serán usados para describir el efecto de las instrucciones:

- El registro contador de programa *pc* (program counter) que contendrá la dirección de la instrucción que está siendo ejecutada, cuyo opcode estará en $M[pc]$.
- El registro tope de pila *sp* (stack pointer) que indicará el elemento en el tope de la pila, cuyo valor está por lo tanto en $M[sp]$.
- El registro base de pila fp (frame pointer) que apunta al registro de activación de la unidad actual.

- El registro base del heap *hp* (heap pointer) que apunta a la dirección de comienzo del *.heap*.
- El registro límite del heap *hl* (heap limit) que indica la dirección más alta del *.heap*. Dicho registro se utiliza en el control de colisión entre las secciones *.heap* y *.stack*.

Una vez que el programa de la CeIVM ha sido analizado, ensamblado y cargado en la sección *.code* y que los datos estáticos han sido cargados en la región *.data*, comienza la ejecución del programa. El funcionamiento de la máquina es muy sencillo y consiste en ejecutar una a una las instrucciones indicadas por el registro *pc* hasta encontrar la instrucción de detención o hasta la ocurrencia de algún error. Como consecuencia de la ejecución de cada instrucción (excepto las de transferencia de control) el valor del *pc* se incrementa en función del tamaño de la instrucción.

El formato general de las instrucciones del CeIISA impuesto por el lenguaje CeIASM consiste de una etiqueta opcional (que es mapeada a la dirección de la instrucción durante el proceso de ensamblado), seguido del mnemónico que hace referencia a la instrucción, y una lista potencialmente vacía de argumentos separados por comas (que dependen de la instrucción especificada):

$$label: \qquad \text{MNEMÓNICO} \qquad Arg_1, Arg_2, \ldots, Arg_n$$

En cada línea del programa CeIASM a lo sumo podrá haber una instrucción. Las etiquetas son identificadores conformados por secuencias de letras, dígitos y los símbolos _ (underscore), $ o @. Dichas secuencias no podrán comenzar con dígitos. Otro aspecto a tener en cuenta viene dado por el hecho de que el lenguaje CeIASM no es sensible a mayúsculas.

A continuación describiremos el repertorio de instrucciones CeIISA a partir de analizar los principales aspectos de MiniJava. Cabe destacar que el CeIISA es lo suficientemente general como para permitir una implementación sencilla de una variedad más amplia de construcciones que las provistas por MiniJava, por lo que no todas las instrucciones del repertorio necesariamente se utilizarán en esta implementación particular. Además el CeIISA presenta cierta redundancia para facilitar la implementación por lo que existe más de una implementación posible para ciertos tipos de construcciones.

3. Evaluación de Expresiones

Supongamos que E es una expresión de MiniJava de la forma $E = E_1 \square E_2$, donde E_1 y E_2 son dos expresiones más simples y $\square$ es un operador binario como $+, -, *, /, \wedge, \vee$, etc. La expresión E debe ser evaluada calculando primero los valores de E_1 y E_2, y aplicando a continuación la operación correspondiente a $\square$. Este cálculo puede ser implementado de varias maneras, guardándose los valores intermedios de E_1 y E_2 en lugares de memoria especiales. En una máquina con arquitectura de pila como la CeIVM, la manera más conveniente de guardar estos valores intermedios es justamente en la pila.

Observemos que estamos resolviendo el problema de evaluar las expresiones de manera inductiva, suponiendo primero que sabemos cómo evaluar las expresiones más simples, en este caso E_1 y E_2. Asimismo, suponemos que los valores de E_1 y E_2 son calculados posiblemente en varios pasos, quedando en el final el resultado correspondiente en el tope de la pila, una posición debajo (la pila en CeIVM decrece) de la posición en la que se inició la evaluación. La base de esta inducción corresponde a las expresiones más simples, es decir, a expresiones sin subexpresiones

tales como constantes literales, y accesos a variables y atributos. Para estas, basta con colocar sus valores respectivos en el tope de la pila. Los operadores unarios serán tratados de manera análoga. El caso de llamadas a unidades que devuelven resultados será tratado más adelante, pero es importante destacar que para cualquiera de estas invocaciones, el efecto será siempre dejar su resultado en el tope de la pila, una posición por debajo del tope en el que se inició la evaluación.

En consecuencia, la CeIVM posee instrucciones que almacenan en la pila valores de constantes, variables y atributos, y otras que ejecutan las operaciones correspondientes a los operadores de MiniJava. El efecto de cada instrucción está descripto indicando las modificaciones en el estado de los registros especiales y de la memoria de la CeIVM. Para la representación de valores booleanos adoptaremos la convención de representarlos a través de valores enteros: *true* es 1 y $false$ es 0.

PUSH	k	Apila una constante k: $sp := sp - 1;\ M[sp] := k;\ pc := pc + 2$
ADD		Adición: $M[sp+1] := M[sp+1] + M[sp];$ $sp := sp + 1;\ pc := pc + 1$
SUB		Sustracción: $M[sp+1] := M[sp+1] - M[sp];$ $sp := sp + 1;\ pc := pc + 1$
MUL		Multiplicación: $M[sp+1] := M[sp+1] * M[sp];$ $sp := sp + 1;\ pc := pc + 1$
DIV		División entera: $M[sp+1] := M[sp+1]/M[sp];$ $sp := sp + 1;\ pc := pc + 1$
MOD		Módulo: $M[sp+1] := M[sp+1]\ mod\ M[sp];$ $sp := sp + 1;\ pc := pc + 1$
NEG		Menos unario: $M[sp] := -M[sp];\ pc := pc + 1$
AND		Conjunción Lógica: si $M[sp+1] \neq 0$ y $M[sp] \neq 0$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0;$ $sp := sp + 1;\ pc := pc + 1$
OR		Conjunción Lógica: si $M[sp+1] \neq 0$ o $M[sp] \neq 0$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0;$ $sp := sp + 1;\ pc := pc + 1$
NOT		Negación Lógica: $M[sp] := 1 - M[sp];\ pc := pc + 1$
EQ		Comparación por igual: si $M[sp+1] = M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0;$ $sp := sp + 1;\ pc := pc + 1$

NE	Comparación por desigual: si $M[sp+1] \neq M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0$; $sp := sp+1;\ pc := pc+1$
LT	Comparación por menor: si $M[sp+1] < M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0$; $sp := sp+1;\ pc := pc+1$
GT	Comparación por mayor: si $M[sp+1] > M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0$; $sp := sp+1;\ pc := pc+1$
LE	Comparación por menor o igual: si $M[sp+1] \leq M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0$; $sp := sp+1;\ pc := pc+1$
GE	Comparación por mayor o igual: si $M[sp+1] \geq M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0$; $sp := sp+1;\ pc := pc+1$

Nótese la disminución del valor del registro *sp* cada vez que se apila un elemento, y el incremento de *sp* cada vez que se desapila. Esto se debe a que la sección *.stack* progresa decrecientemente en la memoria de la CeIVM.

Ejemplo

Consideremos la expresión $a + (b/9 - 3) * c$, supongamos que existe una instrucción llamada "LOAD" que apila el valor de una variable de acuerdo a su dirección de memoria y que los desplazamientos atribuidos por el compilador a las variables a, b, c dentro del registro de activación de la unidad que las contiene son, respectivamente $-3, -5, -2$. Entonces el segmento de programa objeto correspondiente a la traducción de esta expresión sería:

```
LOAD    -3
LOAD    -5
PUSH     9
DIV
PUSH     3
SUB
LOAD    -2
MUL
ADD
```

Una observación interesante es que el código CeIASM generado para las expresiones está directamente relacionado con la notación polaca posfija, que en el ejemplo anterior es $ab9/3-c*+$. Esta secuencia de símbolos puede ser comparada con las instrucciones de CeIASM del ejemplo.

4. Instrucciones de Gestión de Pila

Todas las instrucciones vistas hasta el momento tienen la particularidad de consumir los operandos de la pila al momento de evaluar cada operador. Sin embargo, existen circunstancias en las cuales es deseable realizar tareas tales como conservar resultados parciales para realizar múltiples operaciones sobre los mismos, o eliminar resultados no deseados. Para ello el lenguaje CeIASM cuenta con las siguientes instrucciones:

DUP	Duplica el tope de la pila: $sp := sp - 1;\ M[sp] := M[sp + 1];\ pc := pc + 1$
POP	Descarta el tope de la pila: $sp := sp + 1;\ pc := pc + 1$
SWAP	Intercambia las dos celdas del tope de la pila: $M[sp - 1] := M[sp + 1];\ M[sp + 1] := M[sp];$ $M[sp] := M[sp - 1];\ pc := pc + 1$

5. Instrucciones de Control de Flujo

Para implementar las estructuras de control el CeIASM cuenta con las siguientes instrucciones de desvío:

JUMP	l	Salto incondicional: $pc := addr(l)$
BF	l	Salta si el tope es falso: si $M[sp] = 0$ entonces $pc := addr(l)$ sino $pc := pc + 2$; $sp := sp + 1$
BT	l	Salta si el tope es verdadero: si $M[sp] \neq 0$ entonces $pc := addr(l)$ sino $pc := pc + 2$; $sp := sp + 1$

En estas instrucciones, $addr(l)$ representa un número entero que indica la posición de la etiqueta l en el program CeIASM. Como se indicó precedentemente, un programa CeIASM permite el uso de etiquetas para hacer referencia a locaciones de memoria las cuales son resueltas durante la fase de ensamblado y carga de la CeIVM. Observar que, haya o no desvío, las instrucciones de salto condicional siempre eliminan el valor testeado del tope de la pila.

Además, introduciremos por conveniencia, una instrucción más que no tiene ningún efecto sobre la ejecución, pero que simplifica el proceso de traducción:

NOP	No realiza ninguna operación. $pc := pc + 1$

Luego, una sentencia condicional de la forma: "if (E) S_1 else S_2" donde E es una expresión y S_1 y S_2 son sentencias, podrá ser traducida a:

```
            ...             Traducción de E
            BF      l1
            ...             Traducción de S1
            JUMP    l2
l1 :        NOP
            ...             Traducción de S2
l2 :        NOP
```

En el caso de una sentencia que tenga la forma: "if (E) S", podemos traducirlo a:

```
            ...             Traducción de E
            BF      l
            ...             Traducción de S
l :         NOP
```

Las mismas instrucciones de desvío pueden ser usadas para implementar sentencias repetitivas. En el caso de la sentencia "while (E) S" tenemos la siguiente traducción:

```
l1 :        NOP
            ...             Traducción de E
            BF      l2 :
            ...             Traducción de S
            JUMP    l1 :
l2 :        NOP
```

Es fácil mostrar que estas traducciones de sentencias condicionales y repetitivas son tales que, cuando finalizan su ejecución, dejan el tope de la pila en el mismo estado que al inicio.

6. Entrada/Salida

El lenguaje CeIASM define las siguientes instrucciones para implementar la lectura y escritura de los streams de entrada y salida de un lenguaje:

READ	Lectura de un valor entero: $sp := sp - 1;\ M[sp] :=$"próximo valor entero en el stream de entrada"; $pc := pc + 1$
BPRINT	Impresión de un valor booleano: si $M[sp] = 0$ entonces imprimir $false$ sino imprimir $true$; $sp := sp + 1;\ pc := pc + 1$
CPRINT	Impresión de un valor char: imprimir $chr(M[sp]);\ sp := sp + 1;\ pc := pc + 1$

IPRINT	Impresión de un valor entero: imprimir $M[sp]$; $sp := sp + 1$; $pc := pc + 1$
SPRINT	Impresión de un string: mientras $M[M[sp]] \neq 0$ hacer imprimir $chr(M[M[sp]])$; $M[sp] := M[sp] + 1$; $sp := sp + 1$; $pc := pc + 1$
PRNLN	Impresión del caracter de nueva línea: imprimir "caracter de nueva línea"; $pc := pc + 1$

En las instrucciones anteriores $chr(x)$ denota el caracter asociado al valor entero x.

La instrucción de lectura "READ" deja el valor leído en el tope de la pila al igual que una expresión o función, por lo que dicha instrucción puede utilizarse para implementar funciones o métodos de lectura de enteros, tanto de manera "inlined" como mediante una llamada convencional.

Hay que tener en cuenta que la representación de todos los datos simples en la CeIVM se realiza mediante valores enteros. Por ello, las instrucciones de impresión de datos formatean el dato (si es necesario) como paso previo a escribirlo en el stream de salida.

La instrucción "SPRINT" permite imprimir una cadena de caracteres a partir de contar con un puntero a la misma en el tope de la pila. Es importante destacar que las cadenas de caracteres en CeIVM se consideran terminados con el caracter nulo (valor 0 en dicha locación de memoria) y deben almacenarse progresando hacia locaciones crecientes. Si el puntero a la cadena es nulo (igual a 0), la máquina virtual reportará el error.

La instrucción "PRNLN" permite insertar un caracter de nueva línea en el stream de salida y resulta útil para implementar aquellas primitivas que tienen esta característica.

7. Instrucciones de Asignación y Acceso a Valores

Consideremos ahora las instrucciones necesarias para realizar una operación de asignación a una variable de la forma $V = E$. Si el valor de la expresión E ya se calculó y está en el tope de la pila, se necesita una instrucción capaz de desapilar ese valor y guardarlo en la ubicación de memoria correspondiente a la variable V. En este caso se presenta el problema del direccionamiento de las variables.

El compilador asociará a cada unidad una región en la sección *.stack* en la cual alojará espacio para sus variables (entre otros elementos). Dicha región, conocida como Registro de Activación de la unidad, será referenciada en ejecución por el registro especial fp, quien establecerá su dirección de base.

Por convención, esta dirección de base apunta a la primera variable local del registro de activación correspondiente. Además, el compilador determinará de manera estática un desplazamiento fijo dentro del registro de activación para cada variable de la unidad. Consecuentemente, el desplazamiento asignado por el compilador a la primera variable local será 0.

Las instrucciones que se refieren al acceso y modificación de variables son las siguientes:

LOAD	n	Apila el valor de una variable: $sp := sp - 1;\ M[sp] := M[fp + n];\ pc := pc + 2$
STORE	n	Almacena el tope de la pila en una variable: $M[fp + n] := M[sp];\ sp := sp + 1;\ pc := pc + 2$

En estas instrucciones el valor n constituye el desplazamiento de la variable en el Registro de Activación. Nótese que a raíz de que el *.stack* crece desde las direcciones altas hacia las bajas, las variables locales a una unidad poseerán desplazamientos no positivos a partir de 0.

Adicionalmente al acceso y almacenamiento en locaciones de memoria en el *.stack*, un compilador orientado a objetos (también se aplica para compiladores que implementen primitivas de manejo dinámico de memoria) requiere instrucciones que le permitan acceder a variables o atributos de objetos almacenados en el *.heap*. A diferencia de lo que ocurre con el *.stack*, el cual presenta un comportamiento LIFO, donde existe un único registro de activación activo referenciado por fp, en el *.heap* es posible disponer de múltiples referencias a distintas entidades alojadas dinámicamente y tener un comportamiento en los accesos totalmente arbitrario. Por ello, CeIASM cuenta con las siguientes instrucciones que permiten utilizar como base para el acceso a memoria referencias arbitrarias:

LOADREF	n	Apila el valor de un atributo o variable dinámica: $M[sp] := M[M[sp] + n];\ pc := pc + 2$
STOREREF	n	Almacena el tope de la pila en un atributo o variable dinámica: $M[M[sp + 1] + n] := M[sp];\ sp := sp + 2$ $pc := pc + 2$
DEREF		Desreferencia el puntero del tope de la pila: $M[sp] := M[M[sp]];\ pc := pc + 1$

Nuevamente, n constituye el desplazamiento para la variable, pero dentro de la estructura dinámica o del Registro de Instancia de Clase (CIR). A diferencia del caso anterior, estas instrucciones toman la dirección base de la pila y, como el *.heap* crece conforme aumentan las direcciones de memoria, los desplazamientos en el mismo siempre presentan offsets no negativos. Si la referencia hallada en la pila es nula (igual a 0), la máquina virtual informará del error.

El formato del CIR y las demás estructuras de tiempo de ejecución para implementar la orientación a objetos serán presentadas posteriormente en esta especificación.

Ejemplo

Veremos a continuación el código producto de la traducción de las siguientes sentencias de asignación de MiniJava "$v = 11 * x + v + a$" y "$a = 11 * x + v + a$", donde:

- v es la tercer variable local del método (offset -2 en el registro de activación actual),
- x es el cuarto atributo del objeto actual (offset 4 en el CIR),
- a es el quinto atributo del objeto actual (offset 5 en el CIR).

(trad: $v = 11 * x + v + a$)

```
PUSH      11
LOAD       3
LOADREF    4
MUL
LOAD      -2
ADD
LOAD       3
LOADREF    5
ADD
STORE     -2
```

(trad: $a = 11 * x + v + a$)

```
PUSH      11
LOAD       3
LOADREF    4
MUL
LOAD      -2
ADD
LOAD       3
LOADREF    5
ADD
LOAD       3
SWAP
STOREREF   5
```

Como se verá más adelante, la referencia al objeto actual *this* (necesaria para acceder a x) se encuentra almacenada en el registro de activación del método con un offset igual a 3.

En el caso de asignar a una variable local, almacenar el resultado de evaluar la expresión se realiza sin inconvenientes. Sin embargo, para el caso de actualizar el valor de una variable de instancia, es necesario incluir instrucciones adicionales. Primero se evalúa la expresión. Luego, para almacenar el valor en el atributo, se necesita tener en la pila la referencia al objeto que lo contiene (para usar dicha referencia como base en el CIR). Por ello mediante la instrucción "LOAD 3" se apila la referencia a al objeto actual (*this*). El problema surge a raíz de que "STOREREF" espera los argumentos exactamente en el orden contrario al que han sido apilados, lo que obliga a invertir dichos argumentos previamente mediante la instrucción "SWAP".

Por otro lado, es importante notar que cuando se utilizan expresiones en el contexto de sentencias simples el valor final apilado de la expresión no será requerido. Por lo tanto, para mantener la consistencia del *.stack* deberá descartarse dicho valor mediante el uso de la instrucción "POP". Claramente, si la expresion en el contexto de una sentencia simple resulta en una llamada a un metodo "*void*" entonces no será necesario realizar el POP, ya que no habrá ningun valor apilado.

8. Unidades

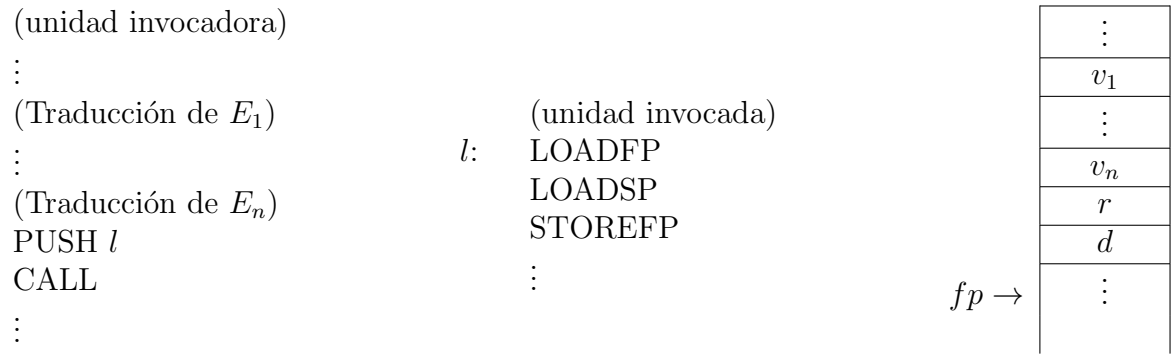
Al momento de definir cómo invocar una unidad, resulta necesario establecer convenciones para determinar las responsabilidades de la unidad invocadora y de la unidad invocada (i.e. calling conventions) y establecer la manera en la que se mantendrá el contexto de la ejecución (i.e. el formato del registro de activación). Resulta necesario almacenar el puntero de retorno y el puntero al registro de activación de la unidad invocadora (i.e enlace dinámico), para poder retomar la ejecución de dicha unidad y reconstruir su contexto de ejecución una vez que la unidad invocada finalice. Estas convenciones, si bien son fijadas por el compilador, están fuertemente condicionadas por el set de instrucciones de la máquina subyacente. El CeIISA define las siguientes instrucciones para este propósito:

CALL	Invoca la unidad cuya dirección se encuentra en el tope de la pila: $M[sp-1] := pc;\ pc := M[sp];\ M[sp] := M[sp-1] + 1;$
LOADFP	Apila el valor del registro fp: $sp := sp - 1;\ M[sp] := fp;\ pc := pc + 1$
LOADSP	Apila el valor del registro sp (al finalizar la instrucción): $sp := sp - 1;\ M[sp] := sp;\ pc := pc + 1$
STOREFP	Almacena el tope de la pila en el registro fp: $fp := M[sp];\ sp := sp + 1;\ pc := pc + 1$

La instrucción "CALL" produce un salto al punto de entrada de la unidad invocada, apilando el puntero de retorno en el *.stack*. Las instrucciones "LOADFP", "LOADSP" y "STOREFP" permiten guardar el enlace dinámico (valor del registro fp para la unidad llamadora) y permiten inicializar el registro de activación de la unidad llamada dando un valor adecuado al registro fp.

Consideremos una llamada a un método estático "*void*" de la forma "$m(E_1, \ldots, E_n)$" y sean $v_1, \ldots, v_n$ los valores efectivos de los parámetros $E_1, \ldots, E_n$ calculados en el tiempo de la llamada. Los valores $v_1, \ldots, v_n$ son apilados en orden antes de la llamada al método por la unidad invocadora. Luego, la unidad invocadora apila la dirección del método m y ejecuta el "CALL", el cual apila la dirección de retorno. Como primera tarea, la unidad invocada tiene que preparar su registro de activación para poder comenzar a ejecutar. Para ello guarda el enlace dinámico ejecutando "LOADFP" y luego inicializa el fp mediante las instrucciones "LOADSP" y "STOREFP".

El esquema de código generado y el estado de la pila luego de la ejecución de estas instrucciones será el siguiente, suponiendo que l es la etiqueta del método a invocar, r es la dirección de retorno almacenada por "CALL" y d es el valor previo del registro fp (enlace dinámico).



Observando la figura, queda claro que dentro del cuerpo del método m se puede tener acceso a los parámetros formales usando la dirección de base fp y desplazamientos positivos que pueden ser calculados por el compilador. Si el método tiene n parámetros, entonces el i-ésimo parámetro tendrá un desplazamiento $(n + 3 - i)$. Si tenemos en cuenta parámetros estructurados (cosa que en MiniJava no se da, ya que cada parámetro ocupa una única locación de memoria), entonces el desplazamiento del i-ésimo parámetro responderá a la siguiente generalización: $\sum_{j=1}^{n} long(v_j) + 3 - \sum_{j=1}^{i-1} long(v_j) - 1$, donde $long(v_j)$ hace referencia a la cantidad de celdas de memoria que componen el parámetro j.

Para completar la construcción del registro de activación, la unidad invocada deberá reservar espacio para las variables locales. Este espacio puede reservarse al comienzo de la unidad (luego de guardar el enlace dinámico e inicializar el fp) o a medida que ocurre cada declaración local de variables. Para la gestión de memoria para variables locales el CeIISA dispone, de las siguientes instrucciones:

RMEM	m	Reserva memoria para variables locales: $sp := sp - m;\ pc := pc + 2$
FMEM	m	Libera memoria de variables locales: $sp := sp + m;\ pc := pc + 2$

Por cada declaración de variables de la forma "*tipo* $V_1, V_2, \ldots, V_n$" deberán reservarse m locaciones de memoria, donde $m = n * k$, siendo k la cantidad de palabras de memoria que ocupa una variable de tipo "*tipo*". En el caso de MiniJava, k siempre es 1 ya que no se manejan tipos estructurados que no sean referencias. Observemos entonces que las variables pueden ser direccionadas fácilmente con desplazamientos $0, -1, -2, \ldots, -(n-1)$ en el registro de activación.

Como consecuencia de la ejecución de las instrucciones anteriores, la unidad invocada quedará correctamente inicializada y la CeIVM podrá proceder a ejecutar las instrucciones resultantes de la traducción del cuerpo de dicha unidad.

La finalización de una unidad debe realizar las tareas correspondientes para dejar la pila en el mismo estado en que estaba antes de su ejecución. Esto implica desalojar la memoria de las variables locales (mediante la instrucción "FMEM"), restablecer el contexto (mediante la instrucción "STOREFP") y transferir el control para retomar la unidad invocadora. El retorno de una unidad se implementa mediante la siguiente instrucción, la cual además de realizar la transferencia de control siguiendo el puntero de retorno, libera el espacio utilizado por los parámetros de la unidad invocada:

RET	m	Retorna de una unidad liberando $m = \sum_{i=1}^{n} long(v_i)$ celdas de memoria correspondientes a los n parámetros: $pc := M[sp];\ sp := sp + m + 1$

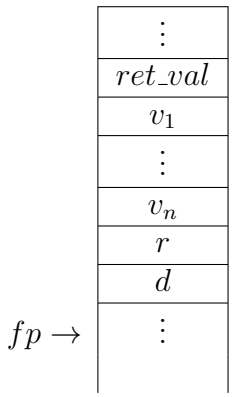
Hasta ahora nos hemos centrado en la implementación de unidades estáticas que no devuelven resultado alguno (i.e. cuyo tipo resultado es "*void*"). La implementación de métodos estáticos que retornan valores es muy semejante a la de los métodos "*void*". La única diferencia es que se asigna una locación de memoria adicional para el valor de retorno, y que dicho valor deberá estar en el tope de la pila después del retorno del método para que se mantenga la coherencia con la implementación de las expresiones.

Una manera conveniente de conseguir el efecto descripto, es hacer que la unidad invocadora reserve una posición en el tope de la pila (indicada con ret_val) antes de evaluar los parámetros actuales de una llamada (usando, por ejemplo, la instrucción "RMEM 1").

Dentro del cuerpo del método, una posición así reservada será usada como una variable local más, asignada durante la sentencia de retorno. Dado que esta variable precede a los parámetros, su desplazamiento será también positivo, con valor $m+3$, donde $m = \sum_{i=1}^{n} long(v_i)$, siendo los v_i los n parámetros. Es decir, se puede pensar esta locación como si fuese el parámetro 0-ésimo.

```
(unidad invocadora)
⋮
RMEM 1
(Traducción de E_1)
⋮
(Traducción de E_n)
PUSH l
CALL
⋮

l:  (unidad invocada)
    LOADFP
    LOADSP
    STOREFP
    ⋮
```



9. Orientación a Objetos

En la sección precedente se analizó la implmementación de unidades vinculadas de manera estática (i.e. su punto de entrada es conocido al momento de compilar y generar el código para su invocación). En un lenguaje orientado a objetos es requisito contar con unidades de instancia que usualmente se vinculan de manera dinámica en función del objeto al que hace referencia una variable en ejecución. En esta sección se presentará el esquema adoptado para representar las estructuras en ejecución que dan soporte a objetos, clases y sus métodos en la CeIVM. En la sección siguiente, se avanzará sobre la implementación de unidades vinculadas de manera dinámica.

Para la implementación de la orientación a objetos en la CeIVM se utilizará una aproximación lo más simplificada posible. En ella no se mantendrá información simbólica (i.e. información de tipos y/o identificadores de métodos, atributos o clases) en tiempo de ejecución. Si bien esta decisión limita la implementación de ciertos aspectos dinámicos presentes en lenguajes orientados a objetos tales como consultas de tipos en tiempo de ejecución (instanceof/getClass), o facilidades para hacer instrospección y/o reflexión; estos aspectos están ausentes en MiniJava, por lo que el esquema adoptado resulta muy adecuado.

Consideremos la siguiente jerarquía de clases para introducir los conceptos de Tabla Virtual (VT) y Registro de Instancia de Clase (CIR):

- Clase A: hereda implícitamente de Object y define:
  - los atributos de instancia $aai_1, \ldots, aai_n$,
  - los métodos de clase $mac_1(\ldots), \ldots, mac_r(\ldots)$, y
  - los métodos de instancia $mai_1(\ldots), \ldots, mai_s(\ldots)$.
- Clase B: hereda de Clase A y define:
  - los atributos de instancia $abi_1, \ldots, abi_n$,
  - los métodos de clase $mbc_1(\ldots), \ldots, mbc_r(\ldots)$, y
  - los métodos de instancia $mbi_1(\ldots), \ldots, mbi_s(\ldots)$.
- Clase C: hereda de Clase A y define:
  - los atributos de instancia $aci_1, \ldots, aci_n$,

- los métodos de clase $mcc_1(\ldots), \ldots, mcc_r(\ldots)$, y
- los métodos de instancia $mci_1(\ldots), \ldots, mci_s(\ldots)$.

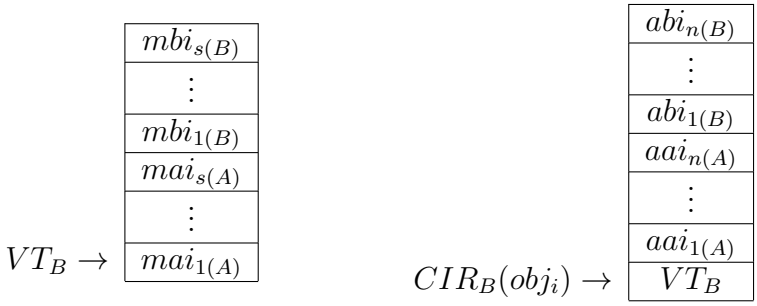
y "redefine" aac_1, aai_1, $mac_1(\ldots)$ y $mai_1(\ldots)$.

A continuación se muestran las estructuras (CR, VT y CIR) para la clase A y sus instancias:

VT_A y $CIR_A(obj_i)$ denotan respectivamente las direcciones de la Tabla Virtual de A, y del Registro de Instancia para la instancia obj_i de A. La VT se aloja en la sección *.data* (también podrían alojarse en el *.heap* pero no es algo estrictamente necesario) ya que su contenido es conocido al momento de iniciar la ejecución. Los CIRs se inicializan en el *.heap* cada vez que se produce la creación de un nuevo objeto. Es importante destacar que dado que tanto la sección *.data* como la sección *.heap* progresan en direcciones de sentido creciente, y que las bases de estas estructuras se establecen en la dirección más baja de las mismas, los desplazamientos en estas estructuras siempre serán no negativos.

La VT, contiene las direcciones de entrada de los métodos vinculados dinámicamente (en MiniJava todos los métodos de instancia siguen este mecanismo de vinculación), y el CIR contiene una referencia a la VT de la clase a la que pertenece la instancia y espacio reservado para almacenar cada uno de los atributos de instancia. A partir de encontrar el CIR de un objeto, siguiendo la referencia a la VT (desplazamiento 0 en el CIR), se podrán encontrar los métodos de instancia aplicables a dicho objeto. Nótese que en ningún momento se mantiene una tabla en ejecución para los métodos estáticos. Si bien para ellos podría utilizarse una tabla de direcciones de salto similar a la VT, justamente, por tratarse de métodos estáticos esto no es necesario. El compilador conoce al momento de implementar una llamada estática a qué código tiene que saltar (tal como se ilustró en la sección anterior), por lo que no resulta necesario destinar recursos en ejecución para resolver la llamada.

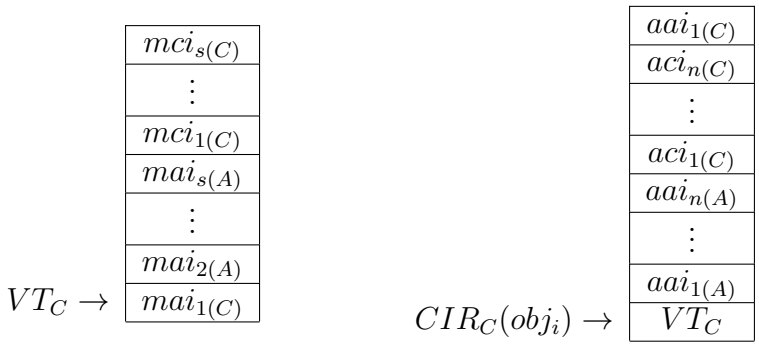
Las estructuras para la clase B se ilustran a continuación:



Como consecuencia de la ausencia de información simbólica en tiempo de ejecución, se utiliza una implementación basada en copia para la herencia. Por lo tanto, todos los miembros de una

clase pueden ser hallados en ejecución mediante un desplazamiento en el registro correspondiente calculado por el compilador. Los desplazamientos de los miembros heredados, se preservan en las estructuras de instancia (CIR y VT) de las clases herederas. En consecuencia, si una variable de tipo referencia a A, hace referencia a una instancia de B (una referencia a un CIR de clase B para dicho objeto), el compilador permitirá y resolverá sólo los accesos a atributos y métodos de la interface provista por A (como si hubiese una referencia a un CIR de A) pero sobre un objeto de clase B (en realidad hay una referencia a un CIR de clase B que no ocasiona problemas ya que tiene desplazamientos consistentes con un CIR de clase A para las entidades heredadas de A). Por ejemplo, un acceso al atributo aai_1 tendrá un desplazamiento de 1 tanto en el CIR para la clase A (si se realiza desde una variable de clase A que referencia a un objeto de clase A) como para la B (si se realiza desde una variable de clase A o B que referencia a un objeto de clase B). En consecuencia, la generación de código se realiza sobre la información de la variable, independientemente del objeto que será referenciado en ejecución.

La VT y la estructura de los CIRs para la clase C y sus instancias son los siguientes:



La clase C a diferencia de B, además de añadir miembros a A, redefine algunos de ellos. De la observación de sus estructuras debe notarse que la única redefinición que se produce es sobre el método mai_1. Los métodos estáticos son accedidos por el compilador a partir de conocer la clase a la que pertenecen. Por ello, mac_1 en A y C son, en cada caso, entidades completamente disociadas que sólo comparten el mismo identificador. Algo similar ocurre con aai_1 en sus versiones de clase A y C. Dado que MiniJava vincula estáticamente los atributos de instancia, lo que ocurre es que el atributo aai_1 de C "oculta" al de A, pero ambos siguen coexistiendo.

Lo interesante ocurre con los métodos como mai_1 que son vinculados dinámicamente (también denominados virtuales). Observando la VT_C debe notarse que la redefinición se implementa sobreescribiendo la dirección del método para que apunte a la versión redefinida. Consideremos como ejemplo en este contexto una variable v de clase A a partir de la cual se quiere invocar al método mai_1. El compilador conoce que el desplazamiento para mai_1 es 0 en la clase A y también es 0 en la clase C (porque ha sido redefinido). En consecuencia, si v referencia a un objeto de la clase A, a partir de un CIR_A se accederá a la VT_A y en ella se encontrará (con desplazamiento 0) la dirección de entrada a la versión de mai_1 de A. Si por el contrario, v hace referencia a un objeto de clase C, entonces a partir de un CIR_C, se accederá a la VT_C y en ella se encontrará (con desplazamiento 0) la dirección de entrada a la versión mai_1 de C. La generación de código para resolver la llamada en ambos casos no difiere (ambas llamadas se realizan a partir de una entidad de clase A). Lo que condiciona la versión del método a ejecutar es la referencia al método hallada en la VT asociada al objeto. Estáticamente el procedimiento de invocación es el mismo, pero dinámicamente (en función del contenido de las estructuras en ejecución y del objeto referenciado) se determina qué versión del método ejecutar.

10. Unidades de Instancia

Habiendo definido las estructuras en ejecución que dan soporte a la orientación a objetos, estamos en condiciones de analizar cómo implementar invocaciones a unidades de instancia. Como se verá, el procedimiento no difiere sustancialmente del utilizado con anterioridad para unidades de clase. La principal diferencia viene dada por el hecho de que dentro de un método de instancia se tiene acceso a los miembros de la clase que lo define. Esto conlleva una modificación en el formato del registro de activación para este tipo de unidades, el cual contendrá una referencia al CIR del objeto asociado.

Consideremos una llamada a un método realizado a partir de una variable v de clase A, de la forma "$v.mai_1(E_1, \ldots, E_n)$", donde $v_1, \ldots, v_n$ son los valores efectivos de los parámetros $E_1, \ldots, E_n$ calculados en el tiempo de la llamada. En este caso se ejemplificará con un método que devuelve un resultado, aunque la implementación para métodos "*void*" es análoga a lo visto anteriormente.

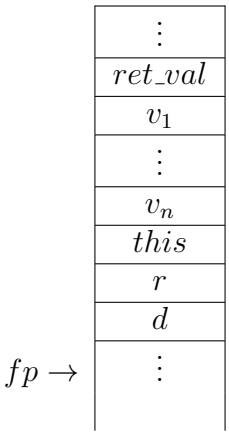
Al igual que para métodos de clase, se genera código para que la unidad llamadora reserve espacio para el resultado de la unidad invocada (a menos que su tipo resultado fuese "*void*"), y para que a continuación se apilen los valores $v_1, \ldots, v_n$ resultantes de la evaluación de los parámetros. Luego, se debe resolver la llamada.

A diferencia de la invocación a unidades estáticas, en el caso de un método virtual el compilador no conoce la dirección del método mai_1 y debe generar código para obtenerla de la VT asociada al objeto referenciado por v. Para ello, se apila la referencia al objeto (como para el ejemplo se trata de una variable local se puede utilizar la instrucción "LOAD", pero en otras circunstacias podría ser algo mas complejo como el resultado de otra llamada). Dado que dicha referencia al objeto será la referencia a *this* una vez que el método mai_1 haya sido invocado, esta referencia tiene que estar al lugar correspondiente para *this* en el registro de activación de mai_1. Sin embargo, tendremos un problema: la referencia al futuro *this* al comenzar la llamada no se encuentra en el lugar adecuado, sino está antes del lugar de retorno y las locaciones utilizadas para los parámetros. Por lo tanto, es necesario utilizar la instrucciones "SWAP" para llevar la referencia al futuro *this* lugar indicado. Luego, se duplica dicha referencia (para dejar una copia en el registro de activación). Con el duplicado de dicha referencia se accede al CIR y se apila la referencia a la VT (desplazamiento 0 en el CIR) mediante la instrucción "LOADREF", y luego se apila la dirección del método mai_1 buscándola en la VT (mediante otro "LOADREF" a partir de la referencia anterior y el desplazamiento de mai_1). Como consecuencia de estas acciones, termina quedando la dirección de mai_1 en el tope de la pila para realizar el "CALL". El protocolo de inicio de la unidad invocada es idéntico al de las unidades de clase.

El esquema de código generado y el estado de la pila luego de la ejecución de estas instrucciones será el siguiente, suponiendo que ret_val es la celda reservada para el valor de retorno, *this* es la referencia al CIR del objeto de la unidad invocada, r es la dirección de retorno almacenada por "CALL", d es el valor previo del registro fp (enlace dinámico), d_v es el desplazamiento de v en el Registro de Activación de la unidad invocadora, y d_{mai_1} es el desplazamiento de mai_1 en la Tabla Virtual:

(unidad invocadora)
⋮
LOAD d_v
RMEM 1
SWAP
(Traducción de E_1)
SWAP
⋮
(Traducción de E_n)
SWAP
DUP
LOADREF 0
LOADREF d_{mai_1}
CALL
⋮

$mai_{1(X)}$: (unidad invocada)
LOADFP
LOADSP
STOREFP
⋮



Nótese que, a diferencia del caso anterior, en el código generado para una llamada virtual no se hace referencia explícita al método invocado. De hecho, se ha notado la etiqueta del método invocado con $mai_{1(X)}$, ya que no puede saberse qué clase representa X (si A o C) hasta conocer en ejecución el contenido de la VT asociada al CIR del objeto a partir del cuál se realiza la llamada. El código generado por el compilador simplemente se limita a realizar la búsqueda necesaria para resolver el vínculo en ejecución (con los costos adicionales que ello conlleva).

La reserva de espacio para variables locales (y el posterior desalojo de las mismas), la generación de código para el cuerpo del método y la restauración del contexto de la unidad invocadora (fp) no presentan variaciones respecto de lo realizado para métodos de clase. Un detalle que hay que considerar es que al momento de retornar, la instrucción de retorno para una unidad de instancia deberá generarse de manera de eliminar de la pila la celda *this* además de los parámetros.

11. Creación de Objetos

Un aspecto que aún no se ha tratado es la creción de instancias de clase. Como se ilustró anteriormente, la representación en memoria de un objeto viene dada por su Registro de Instancia de Clase (CIR). Al momento de construir un objeto, el compilador tiene que generar código para alojar espacio en el *.heap* para el CIR, inicializar la referencia a la VT de la clase, y luego invocar al constructor que correspondiente.

Consideremos una sentencia/expresión para crear un objeto en MiniJava de la forma "*new* $C(E_1, \ldots, E_n)$", donde $v_1, \ldots, v_n$ son los valores efectivos de los parámetros $E_1, \ldots, E_n$ calculados en el tiempo de la llamada, y C es la clase de ejemplo vista en secciones anteriores.

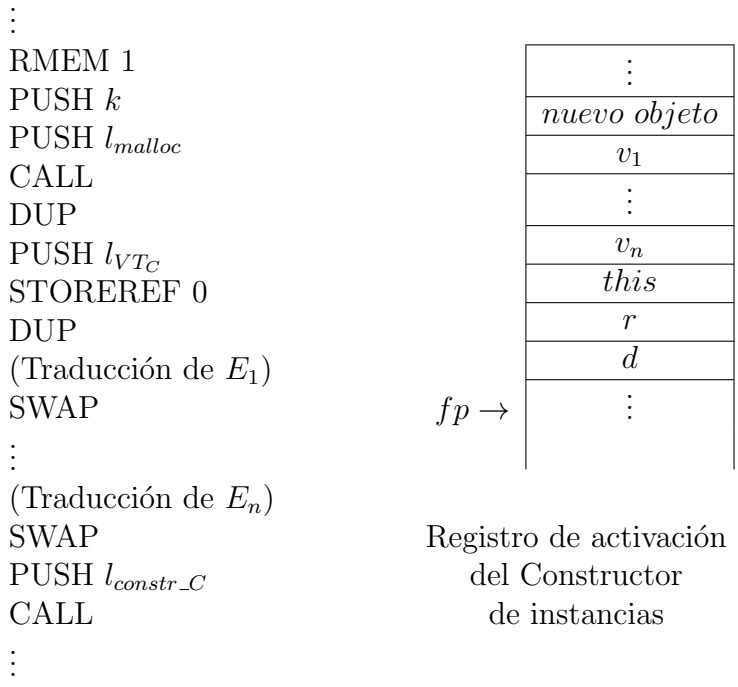
Para resolver esta construcción consideremos a un constructor como una unidad de instancia que en vez de devolver un resultado deja en el tope de la pila una referencia al CIR del objeto creado. Por lo tanto, en primer instacia, el compilador debe armar el CIR. Para esto, genera una llamada a una rutina auxiliar de reserva de memoria en el *.heap*, la cual toma como parámetro el espacio mínimo a reservar y deja en el tope de la pila una referencia a la dirección más baja del bloque alojado (base del CIR). Esta rutina forma parte de un conjunto de rutinas auxiliares

que será descripto posteriormente. Luego, con la referencia al bloque del *.heap* obtenida, se inicializa la referencia del CIR a la VT de la clase.

En este momento tenemos en la estructura del CIR del objeto a crear en el tope de la pila. Dado que dicha referencia al objeto será tanto el resultado del constructor como la referencia a *this* una vez que el constructor haya sido invocado, hay que duplicarla y luego llevarla al lugar correspondiente para *this* en el registro de activación del constructor. Enotonces, al igual que para las unidades de instancia, luego de realizar la evaluacion de cada parametro del constructor será necesario utilizar la instruccion "'SWAP" para ir llevando la referencia al lugar adecuado.

Una vez realizada la inicialización básica del CIR hay que invocar al constructor. A diferencias de los metodos de instancia y los estaticos, el constructor es un método de instancia vinculado estáticamente. Por un lado, su dirección se determina estaticamente (como con los métodos estáticos), mientras que por otro lado requiere la inicialización de la referencia a *this* en su registro de activación (como con los métodos dinámicos). Dado que, para este momento de la ejecución, en la pila ya se tendrán la celda resultado con la referencia al CIR creado, los parámetros apilados, y *this* tambien con una referencia a ese CIR, sólo resta realizar el salto al código del constructor.

La estructura del código de creación de objetos, conjuntamente con el formatos para los registros de activación del constructor, quedarán como sigue, donde k denota el tamaño del CIR para la clase C (el espacio que será alojado en el *.heap*, l_{malloc} es la etiqueta asociada a la dirección de la rutina de reserva de memoria, l_{VT_C} es la etiqueta asociada a la dirección de la Tabla Virtual para la clase C, y l_{constr_C} es la etiqueta del constructor que el compilador halló en C y cuyos parámetros son compatibles con los utilizados en la sentencia *new*:



Observe que al volver de la llamada a un constructor quedara en el tope de la pila una referencia al objeto recientemente creado, lo cual es el resultado esperado de un constructor. Adicionalmente, note que, hubiese sido posible realizar creación de la estructura del CIR (llamada a *malloc* y asociación de la VT) luego de procesar los paramentos. Esto evitaría el uso de los SWAPs. La desventaja de esa alternativa es que el pasaje de parámetros para un método de instancia difiere del de un constructor.

12. Inicialización y Programas

La CeIVM toma un programa CeIASM, lo analiza, lo ensambla y lo carga en memoria. Una vez cumplidas estas etapas, inicializa los registros especiales y procede a dar comienzo a la ejecución. Si bien en este punto, la máquina de ejecución se encuentra inicializada, todo programa debe realizar, como parte de sus tareas más temparanas, la inicialización del ambiente de tiempo de ejecución. La CeIVM inicializa sus registros especiales como sigue:

- $pc := 0$: la máquina comienza su ejecución por la instrucción ensamblada en la dirección más baja de la sección *.code*.
- $sp := EOSTACK$: se inicializa el tope de pila de manera que al apilar el próximo valor, este quede alojado en la dirección anterior (la pila decrece) a la dirección más baja ensamblada en la sección *.stack* ($EOSTACK$ representa esta última dirección y equivale a $MAX + 1$ si no se ha ensamblado ninguna locación en el *.stack*, donde MAX es la dirección más alta implementada).
- $fp := EOSTACK - 1$: el frame pointer se inicializa apuntando a la dirección anterior a la dirección más baja alojada en el *.stack*.
- $hp := EODATA + 1$: el heap pointer se inicializa apuntando a la dirección posterior al final de la sección *.data* ($EODATA$ representa la dirección más alta alojada para la sección *.data*).
- $hl := EOHEAP$: el heap limit se inicializa apuntando a la dirección más alta alojada para la sección *.heap* ($EOHEAP$ representa dicha dirección y equivale a $EODATA$ si no se ha ensamblado ninguna locación en el *.heap*).

En estas condiciones, la máquina se encuentra lista para iniciar la ejecución de un programa con manejo de memoria stack-dynamic. El *pc* se encuentra apuntando a la primera instrucción del programa, el *sp* está listo para comenzar a apilar valores en el *.stack* y el fp tiene un valor tal que permite comenzar a hacer referencias a potenciales variables locales de una unidad global almacenados en las locaciones iniciales del *.stack*. De hecho, las secciones *.data* y *.code* se encuentran protegidas ante overflows en el *.stack* por el valor dado al registro *hl*.

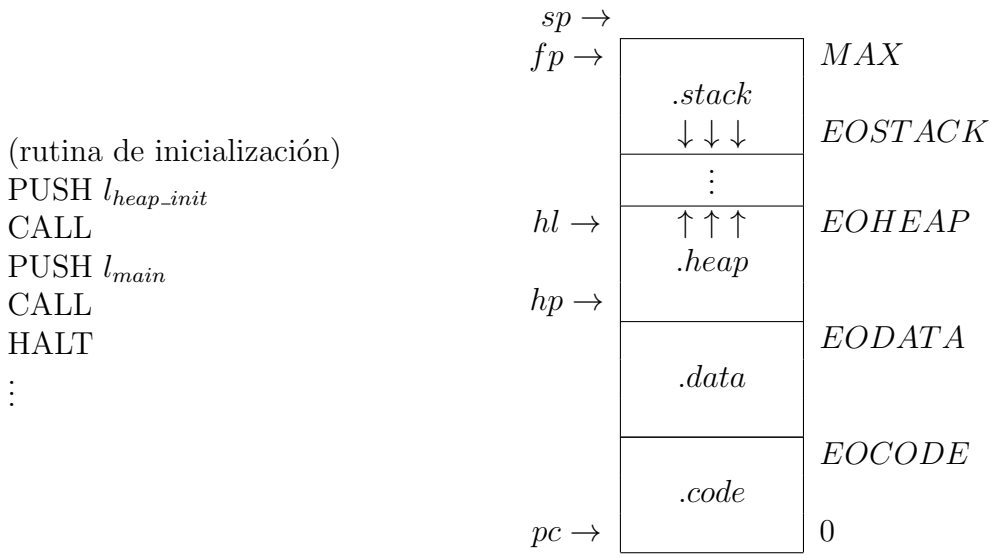
Sin embargo, si el programa a ejecutar requiere del uso del *.heap* (tal y como ocurre en MiniJava y en todo programa orientado a objetos o con manejo dinámico de memoria), hay que inicializar dicha estructura. Si bien la CeIVM en su inicialización delimita correctamente el *.heap*, dicha inicialización no construye la lista de bloques en los que se dividirá al *.heap* para su gestión. La inicialización del *.heap* será implementada mediante una rutina perteneciente a un conjunto de rutinas auxiliares que serán descriptas posteriormente. El compilador entonces, se limitará a incluir dichas rutinas en el código intermedio y a generar la llamada a la rutina de inicialización del *.heap*.

Luego de lo anterior, se está en condiciones de saltar al programa principal (el método *main*). Nótese que *main* en MiniJava es un método *void* estático sin parámetros, por lo que el código de inicialización simplemente apilará la dirección del método y producirá el "CALL". El método *main* como cualquier otro método, completará la inicialización de su registro de activación y continuará con la ejecución del resto del programa. Al finalizar, retornará y el código de inicialización deberá finalizar la ejecución.

La finalización de la ejecución de la máquina se logra mediante la siguiente instrucción:

HALT	Detiene la máquina CeIVM

Por todo lo anterior, la estructura del código de inicialización y el mapa de memoria inicializado quedará como sigue, donde l_{heap_init} es la etiqueta asociada a la dirección de la rutina de inicialización del *.heap*, y l_{main} es la etiqueta asociada a la dirección del método *main*:



13. Directivas al Ensamblador

En lo precedente se introdujeron las instrucciones del CeIISA. Además del set de instrucciones propiamente dicho, el lenguaje CeIASM provee una serie de directivas al ensamblador que facilitan la especificación del programa de código intermedio y condicionan el modo en que la imagen objeto de dicho programa es construida.

1. **Secciones**

   El lenguaje CeIASM no exige realizar un ensamblado continuo de cada una de las cuatro secciones del mapa de memoria de la CeIVM. Para ello provee las siguientes directivas para alternar entre secciones:

.CODE	Prosigue el ensamblado en la sección de *.code*
.DATA	Prosigue el ensamblado en la sección de *.data*
.HEAP	Prosigue el ensamblado en la sección de *.heap*
.STACK	Prosigue el ensamblado en la sección de *.stack*

   Estas directivas le indican al ensamblador que a partir de dicho punto en el programa intermedio, continúe el ensamblado en la sección indicada. Al retomar el ensamblado de una sección ensamblada con anterioridad, este continúa a partir de la última dirección que se había ensamblado (concatenando ambos segmentos). De esta manera, el generador de código intermedio puede, por ejemplo, especificar una secuencia de código, pasar a definir algunos datos, continuar generando código, etc. El ensamblado de las secciones *.code*,

.data y *.heap* procede de manera ascendente en el espacio de direcciones de la máquina, y el ensamblado de la sección *.stack* lo hace en sentido decreciente.

2. **Definición de datos**

   Otra facilidad que provee el lenguaje CeIASM permite definir regiones de memoria preinicializadas mediante la directiva "DW" (define word) en alguno de los siguientes formatos:

$l_i:$	DW	$n_1, \ldots, n_m$	Inicializa m celdas de memoria con los valores $n_1, \ldots, n_m$
$l_i:$	DW	$c_1, \ldots, c_m$	Inicializa m celdas de memoria con los valores numéricos asociados a los caracteres $c_1, \ldots, c_m$
$l_i:$	DW	k DUP(n)	Inicializa k celdas de memoria, todas con el valor n
$l_i:$	DW	$s, 0$	Inicializa $long(s)$ celdas con los caracteres del string s y su terminador nulo correspondiente

A diferencia de lo que ocurre con la JVM para el lenguaje Java, en la CeIVM para MiniJava no se proveen mecanismos de linkeo y carga dinámica de clases (class loaders), las estructuras correspondientes a las clases del programa (CR y VT) se pueden inicializar al comenzar la ejecución de manera estática mediante las directivas anteriores.

A continuación se muestran las inicializaciones de las VT para las clases A, B y C ejemplificadas en secciones precedentes:

	$\vdots$	
	(definición de A)	
	.DATA	
$VT\_ClaseA:$	DW	$mai_{1(A)}, \ldots, mai_{s(A)}$
	.CODE	
	(código de A)	
	$\vdots$	
	(definición de B)	
	.DATA	
$VT\_ClaseB:$	DW	$mai_{1(A)}, \ldots, mai_{s(A)}$
	DW	$mbi_{1(B)}, \ldots, mbi_{s(B)}$
	.CODE	
	(código de B)	

$$
\begin{array}{lll}
 & \vdots & \\
 & \text{(definición de C)} & \\
 & \text{.DATA} & \\
VT\_ClaseC: & \text{DW} & mai_{1(C)}, mai_{2(A)}, \ldots, mai_{s(A)} \\
 & \text{DW} & mci_{1(C)}, \ldots, mci_{s(C)} \\
 & \text{.CODE} & \\
 & \text{(código de C)} & \\
 & \vdots &
\end{array}
$$

La reserva de espacio para los VT puede realizarse con una única directiva "DW" que reserve todo el espacio. En el ejemplo anterior se utilizan, para mayor claridad, una directiva "DW" separada para las declaraciones de cada clase y otra DW para los metodos heredados y redefinidos.

3. **Cadenas de texto literales**

   La directiva "DW" también permite reservar e inicializar memoria para strings literales de manera muy sencilla:

$$
\begin{array}{lll}
l\_str1: & \text{DW} & \text{“}hola\ mundo\text{”}, 0 \\
l\_str1: & \text{DW} & \text{‘}h\text{’,‘}o\text{’,‘}l\text{’,‘}a\text{’,‘ ’,‘}m\text{’,‘}u\text{’,‘}n\text{’,‘}d\text{’,‘}o\text{’}, 0
\end{array}
$$

   Ambas alternativas son equivalentes. Es esencial no omitir el 0 final en la inicialización a los efectos de representar correctamente las cadenas de caracteres en memoria. Esta directiva resulta muy adecuada para alojar strings estáticos en la sección *.data* (por ej. texto estático para mensajes). Sin embargo, a los Strings de MiniJava no es recomendable alojarlos en *.data* ya que los mismos pueden manejarse de manera mucho más uniforme creando objetos en el *.heap* y alojando la cadena con su terminador, conjuntamente con la referencia a la VT, en el CIR del objeto.

4. **Comentarios**

   El CeIASM además admite comentarios de línea comenzados en ; o #, que permiten realizar anotaciones en las instrucciones al momento de generar el código.

$$
\begin{array}{llll}
label: & \text{MNEMÓNICO} & Arg_1, Arg_2, \ldots, Arg_n & ; \textit{Esto es un comentario...} \\
label: & \text{MNEMÓNICO} & Arg_1, Arg_2, \ldots, Arg_n & \#\textit{Esto es un comentario...}
\end{array}
$$

   Si bien los comentarios no son directivas, se incluyen en este apartado para completar la descripción de los elementos del CeIASM.

14. Errores en Tiempo de Ejecución

Un programa fuente puede contener errores léxicos, sintácticos, semánticos y/o lógicos. Además, debido a las limitaciones de recursos de cualquier máquina real, al momento de ejecutar el programa es posible que ocurran errores que van más allá de su correctitud.

El compilador deberá ser capaz de señalar aquellos errores detectables estáticamente, y deberá generar un código CeIASM confiable, con instrucciones destinadas a detectar errores potenciales en tiempo de ejecución.

Podemos agrupar los errores de ejecución en cuatro categorías:

1. **Errores en el código** CeIASM

   Estos errores ocurren durante el análisis y ensamblado, cuando el código CeIASM contiene errores en sus instrucciones y/o directivas. Ejemplos de estos errores son instrucciones mal formadas, etiquetas repetidas o indefinidas, etc.

2. **Errores de insuficiencia de recursos de la máquina virtual** (CeIVM)

   Estos errores se deben a la insuficiencia de espacio en las estructuras definidas para el proceso de análisis, ensamblado y carga del programa. Ejemplos de errores de este tipo son los desbordes en las secciones del programa producto de tener un código intermedio demasiado largo, o con demasiadas etiquetas, etc.

   También a esta categoría pertenencen los errores producto de problemas para iniciar el sistema de E/S de la máquina virtual.

3. **Errores de insuficiencia de recursos reales**

   Estos errores pueden ocurrir cuando la capacidad del sistema subyacente (el sistema que la máquina virtual emula) es insuficiente para atender los requerimientos del intérprete al ejecutar un programa. Por ejemplo, errores como el desbordamiento de pila (`stack overflow`) o la insuficiencia de memoria al momento de alojar espacio en el *.heap*, pertenecen a esta categoría.

   Hay que tener en cuenta que algunos de estos errores podrían ocurrir debido a que el programa es lógicamente incorrecto. Por ejemplo, un programa con un método infinitamente recursivo ocasionará indefectiblemente el desbordamiento de la pila.

4. **Errores de programación controlados por instrucciones de la** CeIVM

   Las instrucciones de la CeIVM realizan ciertos controles como parte de su ejecución. Como consecuencia de estos controles algunas instrucciones pueden detectar errores. Algunos ejemplos de este tipo de errores son:

   - División por cero (instrucción "DIV").
   - Desreferenciamiento de un puntero nulo (instrucciones "LOADREF", "STOREREF", "DEREF" y "SPRINT").
   - Acceso a una locación de memoria no implementada.

Apéndice A: Instrucciones de Acceso a Registros

El CeIISA cuenta con las siguientes instrucciones que permiten modificar y salvaguardar todos los registros especiales:

LOADPC	Apila el valor del registro pc: $sp := sp - 1;\ M[sp] := pc;\ pc := pc + 1$
STOREPC	Almacena el tope de la pila en el registro pc: $pc := M[sp];\ sp := sp + 1;$
LOADSP	Apila el valor del registro sp (al finalizar la instrucción): $sp := sp - 1;\ M[sp] := sp;\ pc := pc + 1$
STORESP	Almacena el tope de la pila en el registro sp: $sp := M[sp];\ pc := pc + 1$
LOADFP	Apila el valor del registro fp: $sp := sp - 1;\ M[sp] := fp;\ pc := pc + 1$
STOREFP	Almacena el tope de la pila en el registro fp: $fp := M[sp];\ sp := sp + 1;\ pc := pc + 1$
LOADHP	Apila el valor del registro hp: $sp := sp - 1;\ M[sp] := hp;\ pc := pc + 1$
STOREHP	Almacena el tope de la pila en el registro hp: $hp := M[sp];\ sp := sp + 1;\ pc := pc + 1$
LOADHL	Apila el valor del registro hl: $sp := sp - 1;\ M[sp] := hl;\ pc := pc + 1$
STOREHL	Almacena el tope de la pila en el registro hl: $hl := M[sp];\ sp := sp + 1;\ pc := pc + 1$

Por uniformidad, el CeIISA brinda instrucciones de acceso y actualización para todos los registros especiales de la CeIVM, pero no todas son estrictamnte necesarias. Algunas de estas instrucciones fueron presentadas para guardar el contexto e inicializar el registro de activación al invocar una unidad. Otras serán utilizadas para la implementación de las rutinas de manejo del *.heap* y no serán manejadas directamente por el código generado por el compilador. Una tercer aplicación para estas instrucciones consiste en modificar el mapa de memoria utilizado por el ambiente de tiempo de ejecución, una posibilidad que no utilizaremos en la implementación de MiniJava.

El set completo de instrucciones de acceso a registros especiales se incluye por una cuestión de completitud en la especificación. Sin embargo, se las detalla en este apartado ya que no todas son utilizadas directamente en la generación de código del compilador. También es muy importante destacar que un uso inadecuado de algunas de estas instrucciones podría dejar el sistema de ejecución en un estado inconsistente.

Apéndice B: Gestión del Heap

En este apartado de presentan diversas alternativas para la implementación de las rutinas de librería que se encargan del manejo del *.heap*. La principal responsabilidad de dichas rutinas consiste en inicializar la estructura de gestión de bloques en el *.heap*, brindar un mecanismo para alojar espacio en dicha estructura, y brindar un mecanismo para liberar y reutilizar el espacio disponible.

Estas tareas son normalmente realizadas por rutinas del sistema operativo (mediante system calls provistas para tal fin), por la Máquina Virtual subyacente que implementa el ambiente de tiempo de ejecución (la cuál dispone de instrucciones para gestionar la memoria dinámica), o, como en el caso de la CeIVM, de rutinas utilitarias que se linkean con el programa del usuario

y lo complementan llevando a cabo estas tareas. Esta última alternativa presenta la ventaja de simplificar el diseño de la máquina de ejecución, al tiempo que permite al implementador variar la estrategia de gestión de memoria que desee emplear. En este contexto, cada compilador que genere código CeIASM podría utilizar su propio esquema de gestión de memoria para el *.heap*.

Gestión simplificada del Heap

En esta sección presentamos un esquema ultra-simplificado para realizar la gestión del *.heap*. En este esquema sólo se permite alojar memoria pero no liberarla. Dado que los bloques sólo pueden ser alojados, la estrategia simplemente consiste en ir incrementando el tamaño de la sección de *.heap*, aumentando el valor del registro *hl* a medida que se van alojando los bloques.

Este esquema simplifica enormemente las tareas a realizar ya que se eliminan las complejidades propias de buscar, en un espacio no compacto, la región de locaciones libres contiguas más adecuada al momento de alojar un bloque. También se evitan las complicaciones propias de mantener la estructura enlazada de bloques libres y los mecanismos de fusión de bloques libres contiguos.

Cabe destacar que con este esquema es posible implementar un ambiente de tiempo de ejecución plenamente funcional pero tremendamente ineficiente en cuanto a ocupación de memoria para aplicaciones prácticas. Sin embargo, esto no supone inconveniente para programas de la escala de los utilizados para comprobar el correcto funcionamiento del compilador. Adicionalmente, la presentación de este esquema simplificado permite establecer una base de comparación con el esquema completo, y además, ilustrar la posibilidad que tiene el implementador de variar su esquema de asignación dinámica.

El esquema simplificado se compone, entonces, de dos rutinas. A continuación se introduce la rutina de inicialización del *.heap*:

```
simple_heap_init:   RET     0           ; Inicialización simplificada del .heap
```

Dado que el crecimiento del *.heap* en este esquema es lineal y no se produce liberación de bloques, no resulta necesario inicializar una estructura de soporte para gestionarlo. Por ello, la rutina *simple_heap_init* sólo se limita a retornar sin realizar actividad alguna. Un compilador podría omitir directamente la llamada a esta rutina. Sin embargo, la misma se incluye por razones de uniformidad. Si un compilador genera las llamadas a las rutinas de gestión de *.heap*, cambiar el esquema de gestión es tan simple como cambiar el cuerpo de las rutinas invocadas.

La rutina *simple_malloc* permite alojar bloques en el *.heap* de manera creciente y contigua. Tiene un argumento (el cual tiene que ser positivo) que indica cuántas locaciones de memoria se alojarán para el bloque. La rutina deja en el tope de la pila una referencia al bloque recientemente alojado (por lo que debe realizarse un "RMEM 1" previamente a apilar el parámetro para su invocación).

```
simple_malloc:  LOADFP          ; Inicialización unidad
                LOADSP
                STOREFP         ; Finaliza inicialización del RA
                LOADHL          ; hl
                DUP             ; hl
                PUSH    1       ; 1
                ADD             ; hl + 1
                STORE   4       ; Guarda resultado (puntero a base del bloque)
                LOAD    3       ; Carga cantidad de celdas a alojar (parámetro)
                ADD
                STOREHL         ; Mueve el heap limit (hl)
                STOREFP
                RET     1       ; Retorna eliminando el parámetro
```

Con estas dos rutinas se puede implementar un sistema minimal que permita alojar objetos en el *.heap*. El esquema completo, que incluye la posibilidad de liberar memoria y una estructura para darle soporte a esta tarea, será abordado en la sección siguiente.