

4. Especificación de la máquina de pila simbólica UDMPs99

JosuKa Díaz Labrador, Andoni Eguíluz Morán

(con la colaboración especial en algún momento de su historia de
Jesús Redrado Redrado, Jorge García Iglesias y Ana Cabana Pérez)

Universidad de Deusto, febrero de 1999 (actualización: marzo de 2011)

Índice

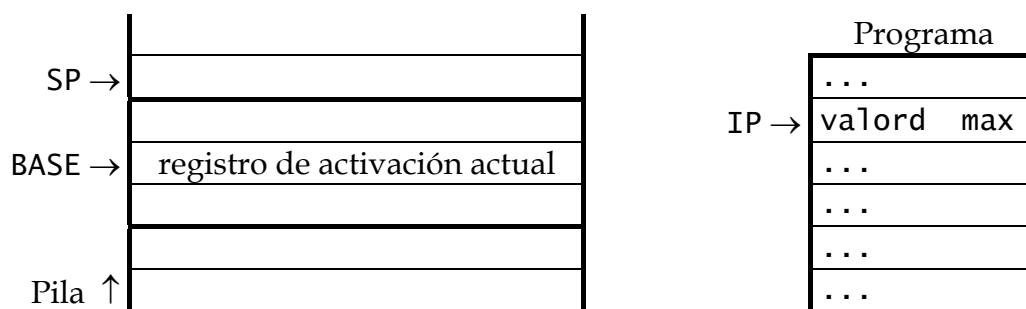
4.1. Introducción.....	1
4.2. Operadores matemáticos	2
4.3. Manejo de la pila	4
4.4. Control de flujo.....	5
4.5. Primitivas de entrada y salida.....	5
4.6. Estructura del programa y subprogramas	6
4.7. Registro de activación	7
4.8. Declaración de variables	8
4.9. Acceso a las variables y asignaciones.....	9
4.10. Subprogramas: llamada y retorno	10
4.11. El intérprete UDMPs99	10
4.12. Historia, agradecimientos y referencias.....	11
4.13. Ejemplo de conversión de AJK a máquina de pila	13

4.1. Introducción

La máquina de pila simbólica UDMPs99 es un código intermedio sencillo (no sirve más que para lenguajes simples, obviamente) que permitirá aproximarse a los conceptos de procesamiento semántico y de generación de código de un compilador.

Se supone que existen tres registros que manejan la pila y el programa:

- SP señala en cada momento la primera posición libre de la pila;
- BASE señala en cada momento el registro de activación actual (ver 4.7);
- IP señala en cada momento la instrucción actual que se ejecuta.



La memoria para los *datos* del programa (variables) se encuentra en la pila, dentro de los diferentes registros de activación.

La especificación es *simbólica* porque se accede a los datos por su nombre (identificadores), por la existencia de etiquetas para los saltos, y porque en ningún momento se hace uso explícito de los registros internos.



Hay tres tipos de datos: *bytes* (u *octetos*), *enteros* y *reales* (el primero sirve principalmente para implementar datos de carácter, lo que habitualmente se conoce como *char*). Muchas instrucciones se diferencian según el dato sea octeto (sufijo *b*), entero (sin sufijo o sufijo *i*) o sea real (sufijo *r*). En la pila, durante la ejecución, pueden existir también *direcciones* (son también los valores que pueden adoptar los registros SP, BASE e IP), que se manejan con instrucciones específicas, o en todo caso, con las que admiten operandos enteros. Los octetos ocupan 1 byte (increíble, pero cierto), los enteros 4 bytes y los reales 8 bytes. Las direcciones ocupan 4 bytes.

La notación que se utilizará en la especificación del repertorio de instrucciones de la máquina de pila es la siguiente:

- *código_instr*
es una instrucción sin argumento;
- *código_instr argumento*
es una instrucción con un argumento;

Las instrucciones de la máquina de pila actúan muchas veces, como se sabe, sobre *datos* que se encuentran en la cumbre de la pila; tales datos no deben confundirse con los *argumentos* de la instrucción, que pueden ser (según los casos):

- *octeto* es una constante literal de tipo octeto; como se usará mayormente para caracteres, se escribirá entre comillas simples un solo carácter; también se admitirá la notación numérica: en decimal (después de \) o en hexadecimal (empezando por \x); en ambos casos, el número deberá ser menor de 256;
- *entero* es una constante literal de tipo entero, que puede venir escrita como en el lenguaje C: en decimal, en octal (si empieza por 0) o en hexadecimal (si empieza por 0x);
- *real* es una constante literal de tipo real, que puede venir escrita como en el lenguaje C: con un punto decimal (puede que no haya dígitos antes o después del punto), y con una parte exponencial opcional;
- *cadena* es una secuencia de caracteres, comenzando y terminando con comillas dobles; los caracteres que se admiten son todos los que están por encima del blanco (incluido), salvo el carácter de las comillas dobles;
- *identificador* es una secuencia que comienza con letra y puede tener letras, números o símbolos del conjunto { #, -, _ };
- *etiqueta* es un *identificador* precedido del signo #.

4.2. Operadores matemáticos

Los operadores matemáticos son instrucciones que no tienen argumentos, y existen normalmente versiones distintas para cada tipo de dato de la máquina de pila (octetos, enteros y reales). Por lo que respecta a su efecto sobre la pila podemos distinguir dos grupos: los *operadores binarios* y los *operadores unarios*. Los binarios requieren la existencia de dos datos en la cumbre de la pila: el primero de ellos (de la cumbre) será el *segundo operando* y el siguiente el *primer operando*, que siempre han de ser del mismo tipo; entonces, dejan en la pila el resultado, que será del mismo tipo que los operandos, salvo cuando se diga lo contrario. Los operadores unarios sólo extraen el último dato de la pila y dejan el resultado en la pila, que será del mismo tipo que el operando, salvo cuando se diga lo contrario.

4.2.1. Aritméticos

Los operadores aritméticos binarios están representados por los códigos habituales (^ es la exponenciación):

- + - * / ^
 para enteros;
- +r -r *r /r ^r
 para reales;
- +b -b
 para octetos (solamente existe suma y resta).

En la exponenciación, el primer operando es la base y el segundo el exponente. Los operadores aritméticos unarios son, respectivamente para enteros, reales y octetos:

- neg
- negr
- negb

y realizan el cambio de signo aritmético. Es importante reseñar que los operadores aritméticos no llevan a cabo conversiones de tipo, esto es, sus operandos deben ser del tipo esperado. La máquina de pila posee instrucciones para conversión de tipos (sección 0).

4.2.2. Relacionales

Los operadores relacionales son, respectivamente para enteros, reales y octetos:

- < > <= >= == !=
- <r >r <=r >=r ==r !=r
- b <=b >=b ==b !=b

En cuanto a su efecto sobre la pila, son binarios, dejando como resultado en la pila *siempre un valor octeto*, 1 para representar *verdadero* y 0 para representar *falso*.

4.2.3. Lógicos

Existen tres operadores lógicos: **and** y **or** (binarios), y **not** (unario); los tres tienen operandos de tipo octeto y producen un octeto como resultado:

- **and**
 toma dos valores de la pila que deben ser 0 o 1, y si ambos son 1 introduce un 1 en la pila como resultado; en caso contrario introduce un 0;
- **or**
 toma dos valores (0 o 1) de la pila y si alguno de ellos es 1 introduce un 1 en la pila; en caso contrario introduce un 0;
- **not**
 toma un valor de la pila, que debe ser 0 o 1, e introduce el valor contrario al que existía.

4.2.4. *Conversión de tipos*

Las instrucciones para convertir unos tipos en otros no tienen argumentos y se comportan como operadores unarios:

- `intareal`
extrae el valor entero de la cima de la pila y lo apila como valor real;
- `intabyte`
convierte el valor entero situado en la cima de la pila a un valor de octeto; la conversión se realiza desechando los tres bytes más significativos (cualquier entero inferior a 128 se convierte en un octeto con el mismo valor, mientras que en otros casos el resultado es indeterminado);
- `byteaint`
convierte el valor octeto situado en la cima de la pila a entero (del mismo valor numérico si el octeto es inferior a 128, e indeterminado en otros casos).

Debe notarse que no existe `realaint` (ver 4.2.5).

4.2.5. *Primitivas de cálculo*

Todas las primitivas de cálculo son operadores unarios sin argumentos y esperan un operando de tipo real. Toman un operando de la pila, realizan el cálculo y dejan en la pila un resultado.

Las primitivas `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp` (*e* elevado al operando), `log` (logaritmo decimal) y `ln` (logaritmo neperiano) producen un valor real. Las primitivas `round` y `trunc` (con el significado habitual) producen como resultado un entero.

4.3. *Manejo de la pila*

Las siguientes instrucciones permiten añadir, quitar, duplicar o intercambiar valores de la pila:

- `insi entero`
introduce el valor *entero* en la pila;
- `insi`
inserta espacio para un entero con un valor indeterminado;
- `desapilar i`
elimina un valor entero de la cima de la pila;
- `insr real`
- `insr`
- `desapilar r`
lo mismo para reales;
- `insb octeto`
- `insb`
- `desapilar b`
lo mismo para octetos;
- `desapilar entero`
elimina *entero* bytes de la cima de la pila;

- **copiari**
duplica el valor entero de la cima de la pila.
- **copiarr**
- **copiarb**
lo mismo para real y octeto, respectivamente;
- **cambiarri**
en la pila hay dos valores enteros; esta instrucción intercambia ambos valores;
- **cambiarr**
- **cambiarbb**
lo mismo para real y octeto, respectivamente;
- **cambiarir**
en la cima de la pila hay un real, y debajo un entero; intercambia ambos valores en la pila;
- **cambiarri**
en la cima de la pila hay un entero, y debajo un real: se intercambian;
- **cambiarbi**
- **cambiarib**
- **cambiarbr**
- **cambiarrb**
lo mismo para otros tipos de datos, según indican los sufijos.

4.4. Control de flujo

Las instrucciones de salto de la máquina de pila tienen un argumento, que siempre es una *etiqueta*:

- **si-cierto-ir-a *etiqueta***
toma un operando octeto de la pila; si es cierto (1) salta a *etiqueta*; si es falso (0) sigue en secuencia;
- **si-falso-ir-a *etiqueta***
toma un operando octeto de la pila; si es cierto (1) salta a *etiqueta*; si es falso (0) sigue en secuencia;
- **ir-a *etiqueta***
salta incondicionalmente a *etiqueta*;
- **eti*q* *etiqueta***
sirve para declarar una etiqueta (no tiene ejecución propiamente dicha).

Obviamente, es un error que una instrucción de salto se refiera a una etiqueta que no existe (que no tiene pseudoinstrucción *eti*q**). En cualquier caso, sobre el ámbito de las etiquetas, ver la sección 4.6.

4.5. Primitivas de entrada y salida

Las siguientes instrucciones permiten realizar mínimas operaciones de entrada y salida:

- **leeri**
toma un valor entero por el teclado (escrito en decimal) y lo deposita en la cima de la pila;

- **leerr**
toma un valor real por el teclado y lo deposita en la cima de la pila;
- **leerb**
toma un valor octeto por el teclado (en forma de carácter) y lo deposita en la cima de la pila;
- **escribiri**
visualiza el valor de la cima de la pila (que debe ser entero) y lo elimina de la misma;
- **escribirr**
visualiza el valor de la cima de la pila (que debe ser real) y lo elimina de la misma;
- **escribirb**
visualiza el valor de la cima de la pila (que debe ser octeto) en forma de carácter y lo elimina de la misma;
- **escribirln**
visualiza una nueva línea; no hace nada con respecto a la pila;
- **escribirs *cadena***
visualiza *cadena*; no hace nada con respecto a la pila.

4.6. Estructura del programa y subprogramas

Un *programa de máquina de pila* contiene declaraciones globales (ver 4.8), cero o más trozos de código de subprogramas y *un programa principal*. Los subprogramas pueden ser *procedimientos* (sin valor de retorno) o *funciones* (con valor de retorno), y pueden tener *parámetros*. Los trozos de código de cada subprograma deberían ser compactos, encerrados entre instrucciones específicas (tal como se explica un poco más adelante). Las declaraciones globales y los trozos de código de los subprogramas pueden mezclarse entre sí (o sea, no es obligatorio que *todas* las declaraciones globales aparezcan lo primero), pero debe tenerse presente que, obviamente y en general, un identificador no puede usarse si no se ha declarado antes.

El programa principal es el conjunto de instrucciones que están entre una instrucción especial llamada *inicio* y el final del programa. No puede incluir declaración de ninguna clase (ni de variables ni de subprogramas, solo pueden ser instrucciones). Para aclararnos definitivamente, un programa de máquina de pila es:

$$\langle ProgMáqPila \rangle \rightarrow (\langle DeclGlobal \rangle \mid \langle DeclSubprog \rangle)^* \\ \text{inicio} \\ \langle InstrEjecutable \rangle^*$$

Hay que tener en cuenta que las instrucciones de lo que se llama programa principal no tienen obligatoriamente que forman un “todo” lógico y coherente, pueden ser trozos inconexos entre sí (pensar a este respecto en lo que podría ser un programa en puro lenguaje máquina) a los que se accede a través de un salto desde algún otro punto del programa (incluso desde un subprograma). Las instrucciones relativas a este asunto son:

- **inicio**
además de lo dicho, marca el punto en que comenzará la ejecución del programa (por eso debe aparecer una sola vez en el programa, y fuera de cualquier subprograma);

- **fin**
termina la ejecución del programa; puede aparecer tantas veces como se precise en el programa (incluso dentro del código de subprogramas).

Las siguientes instrucciones sirven para declarar el comienzo del código de un *subprograma*, y no tienen ejecución propiamente dicha:

- **etiqr** *identificador*
declara el comienzo de un procedimiento;
- **etiqr** *identificador*
- **etiqr** *identificador*
- **etiqr** *identificador*
declaran el comienzo de una función que devuelve un entero, un real o un octeto, respectivamente;

La siguiente instrucción sirve para declarar el final del código de un subprograma, y no tiene ejecución propiamente dicha:

- **fin** *identificador*
declara el final de un subprograma; esta instrucción no solo no es ejecutable, sino que en realidad la ejecución no debería llegar nunca a ella; en caso contrario, el intérprete de máquina de pila señalará un error y terminará la ejecución.

Como se ha dicho, el código correspondiente a un subprograma debería estar completamente escrito entre las pseudoinstrucciones de comienzo y de final (aunque esto podría no ser así, y aún funcionar el programa, si se dan ciertas condiciones):

$$\langle DeclSubprog \rangle \rightarrow \text{etiqr? } \textit{identificador} \\ \langle DeclParam \rangle^* \\ (\langle DeclLocal \rangle \mid \langle InstrEjecutable \rangle)^* \\ \text{fin } \textit{identificador}$$

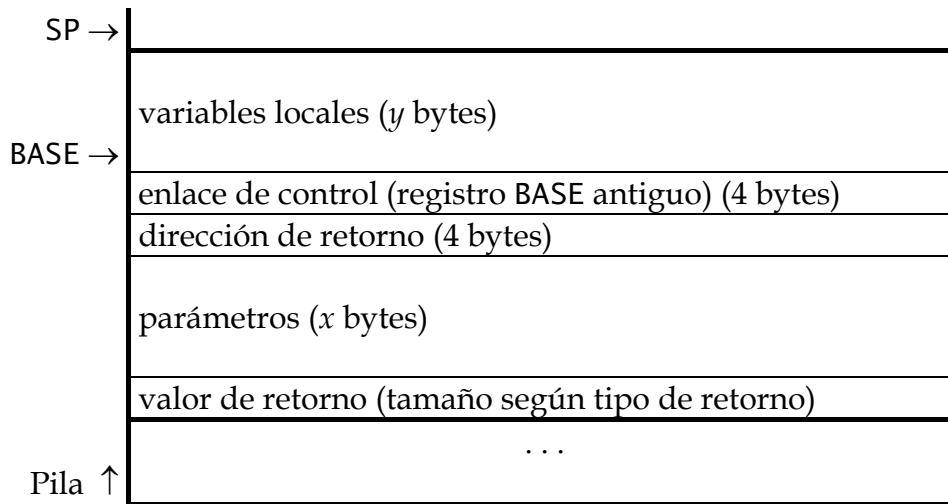
Las declaraciones de parámetros y de locales se explican en la sección 4.8.

En lo concerniente a las etiquetas de las instrucciones de salto (sección 4.4), sus nombres han de ser únicos a lo largo de todo el programa (incluidos los subprogramas). Una instrucción de salto y la etiqueta (instrucción **etiqr**) a la que se refiere pueden estar una dentro de un subprograma y la otra fuera, aunque esto no es lo normal, pudiendo producirse errores fatales en la ejecución, si las instrucciones que se ejecuten con posterioridad al salto no tienen sentido dado el contexto de la máquina.

4.7. Registro de activación

La pila de control es la misma pila de datos. El formato del registro de activación es el recogido en la figura que aparece más adelante.

Debe tenerse presente que cuando se dice que SP “es la cumbre de la pila”, lo que ocurre es que señala a la primera dirección libre por encima; es decir, si la cumbre de la pila tiene un entero (o una dirección), este ocupa los bytes entre SP (excluido) y SP-4 (incluido), hasta SP-1 (incluido) si tiene un octeto, y hasta SP-8 (incluido) si tiene un real. También ocurre lo mismo con el registro BASE.



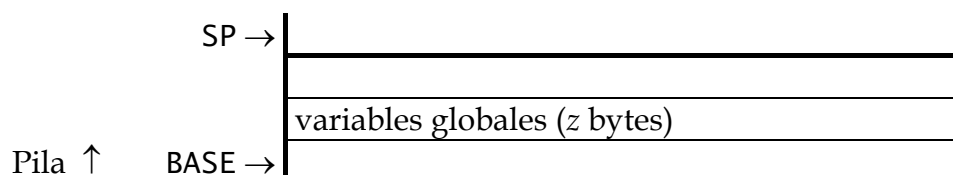
4.8. Declaración de variables

Hay tres clases de instrucciones, que sirven básicamente para declarar un nombre de variable y su tipo, declaraciones que son utilizadas por el intérprete de máquina de pila para acceder luego a los datos mediante los nombres declarados. Debe tenerse presente que la memoria para los datos se encuentra en algún lugar de la pila durante la ejecución, y que esta clase de instrucciones sería la que primero debería desaparecer en una “máquina de pila no simbólica”. Por eso, alguna de las instrucciones siguientes tiene cierto efecto sobre la pila, además de la propia declaración.

Las variables *globales* son visibles durante todo el programa (salvo si los mismos nombres se declaran como locales o parámetros), desde el momento en que se declaran con las siguientes instrucciones (categoría *DeclGlobal*):

- `globalb` *identificador*
- `globali` *identificador*
- `globalr` *identificador*

Es evidentemente un error si se dan dos declaraciones de esta clase para el mismo nombre. Ahora bien, los identificadores de subprogramas (instrucciones `eti??`, sección 4.6) también son *globales*, por lo que no puede haber una variable global con el mismo nombre que un subprograma ya declarado o viceversa. Por otro lado, el único efecto sobre la pila o la ejecución de las instrucciones `global?` es que el intérprete coloca espacio para estas variables en la base de la pila justo antes de comenzar la ejecución del programa, según indica la siguiente figura:



Los nombres de *parámetros* de subprogramas son visibles solamente durante el cuerpo del subprograma en que se declaran (categoría *DeclParam*) con:

- `paramb` *identificador*
- `parami` *identificador*
- `paramr` *identificador*

Estas instrucciones no tienen ejecución, pero tienen relación con el espacio asignado a los parámetros de los subprogramas (sección 4.7); por ello, han de colocarse exactamente después de la etiqueta de comienzo del procedimiento o la función, y habrá de conocerse la manera de eliminar este espacio cuando el subprograma termina. Es un error obvio que haya dos nombres de parámetros iguales en un mismo subprograma, pero un parámetro y un nombre global pueden ser iguales.

Las variables *locales* son visibles solamente durante el cuerpo del subprograma en que se declaran (categoría $\langle DeclLocal \rangle$) con:

- `localb` *identificador*
- `locali` *identificador*
- `localr` *identificador*

Estas instrucciones sí tienen ejecución: crean (apilan) espacio, de tamaño correspondiente al tipo y con valor indeterminado, en la cumbre de la pila para localizar la variable. Por ello, las instrucciones deben colocarse exactamente dentro del cuerpo de un subprograma para que el espacio se cree en el lugar adecuado del registro de activación (sección 4.7). También habrá que estar atentos, por tanto, a la eliminación de dicho espacio cuando termine un subprograma. Está claro que un nombre de variable local no puede ser igual al de otra variable local o parámetro del mismo subprograma, pero puede ser un nombre global.

En definitiva, existen dos *ámbitos* para la resolución de nombres en la máquina de pila: el *ámbito global* (para nombres de subprogramas y de variables globales), y el *ámbito local* (para nombres de parámetros y de variables locales). En las instrucciones que aparecen en las siguientes secciones que tengan un argumento *identificador*, se supone que la resolución de tal nombre se lleva a cabo del siguiente modo:

- si el nombre está dentro del trozo de código de un subprograma, se examina en primer lugar el ámbito local, y solo si no está ahí, se pasa al global;
- si el nombre está en una instrucción que no pertenece al código de un subprograma, se examina solo el ámbito global.

Es obviamente un error el que no se encuentre el *identificador*, o que no posea las características esperadas según el uso que se está haciendo del mismo.

4.9. Acceso a las variables y asignaciones

Las siguientes instrucciones permiten acceder a las variables del programa de máquina de pila (ya sean globales, parámetros o locales). Se supone que las siguientes instrucciones pueden usarse solamente en los puntos donde está activa una declaración de la variable *identificador* (tal como se explica en la sección 4.8, conociéndose entonces el tipo de dicha variable):

- `valord` *identificador*
inserta el contenido (*r-valor*) de la variable *identificador* en la pila;
- `valori` *identificador*
inserta la dirección (*l-valor*) de la variable *identificador* en la pila;

Esta última instrucción puede usarse también, estrictamente dentro del trozo de código de una función, con el *identificador* de la misma, teniendo el propósito de asignar su valor de retorno:

- *valori identificador*
si *identificador* es el nombre de la función actual, introduce la dirección de su valor de retorno en la pila;

Finalmente, las instrucciones para realizar modificaciones de las variables o datos en general son las siguientes:

- **:=**
es la asignación para enteros; extrae dos datos de la pila: el primero (en el orden de extracción) es un valor entero (r-valor), y el segundo es una dirección (l-valor); coloca el r-valor en la dirección l-valor, y no añade nada en la pila;
- **:=r**
- **:=b**
es igual que lo anterior, pero para reales y octetos, respectivamente.

4.10. Subprogramas: llamada y retorno

Las siguientes instrucciones se utilizarán para los procesos de llamada y retorno, respectivamente:

- **llamar identificador**
guarda la dirección de la siguiente instrucción (dirección de retorno) en la pila y salta a la etiqueta del subprograma *identificador*;
- **ret entero**
toma el valor de la cima de la pila como dirección de retorno, eliminando además de la pila tantos *bytes* como indique *entero*, y cede el control a dicha dirección de retorno; si el argumento no se indica, se supone 0; dado el registro de activación de la sección 4.7, queda claro que el espacio que se desapila es el correspondiente a los parámetros del subprograma.

Las siguientes instrucciones se utilizarán durante tales procesos (recordar nuevamente la estructura del registro de activación):

- **ponerbase**
introduce en la pila el valor (que es una dirección) del registro BASE, y asigna a este el valor de SP; por lo tanto, BASE queda señalando la posición de la pila que es cumbre en este momento;
- **cogerbase**
saca el valor (que es una dirección) de la cima de la pila y lo carga en BASE; son exactamente las operaciones contrarias a las que realiza **ponerbase**.

4.11. El intérprete UDMPS99

El intérprete del lenguaje de máquina de pila ejecuta los programas de máquina de pila, que se supone residen en archivos con la extensión *mpv*. Así:

```
udmps99 miprog
```

ejecutará el programa del archivo *miprog.mpv*. El intérprete procesa el programa (si encuentra errores, informa de ello, y además genera el archivo *miprog.err*), y genera un archivo llamado *miprog.mpn*, en lenguaje de máquina de pila *no simbólica*, que es lo que realmente se ejecuta.

En este lenguaje no simbólico, se han eliminado todas las declaraciones de variables (pseudoinstrucciones de la sección 4.8), y se han sustituido los identificadores de las instrucciones `valori` y `valord` por referencias a memoria.

Existen mínimas ayudas de depuración. Para empezar, las operaciones de entrada y salida se duplican en el archivo `miprog.out`. Por otro lado, si se ejecuta:

```
udmps99 miprog -d
```

se genera un archivo `miprog.log`, donde se escribe cada instrucción según se va ejecutando. Si el programa de máquina de pila no se ejecuta como se espera, puede verse en tal archivo lo que está pasando. Sin embargo, esperamos que no sea una gran noticia que advirtamos de lo *peligrosa* que puede llegar a ser esta última opción.

4.12. Historia, agradecimientos y referencias

El lenguaje de código intermedio UDMPs99 empezó a gestarse por Andoni Eguíluz y JosuKa Díaz, incluso ya desde el año 1993, siendo al principio una versión aumentada, si cabe el atrevimiento, del código usado por Aho, Sethi y Ullman [1990] en el mítico *libro del dragón* con el que todos aprendimos compiladores. Una versión inicial completamente funcional fue terminada por JosuKa para su tesis en 1994-1996, para un compilador de PASCAL que optimizaba la recursividad final (en ciertas condiciones [Díaz-Labrador 1996]). La máquina era código intermedio válido para un subconjunto de PASCAL suficientemente significativo, y el intérprete estaba escrito en SCHEME (como el resto del compilador), lo que aportaba un grado adicional de abstracción y ausencia de errores que, vistas desde la distancia, resultan de todavía más valor.

El impulso dado a la misma idea por esa misma época por el lenguaje JAVA [Gosling, Joy y Steele 1996] y la JAVA VIRTUAL MACHINE [Lindholm y Yellin 1996] (a la postre y salvando las distancias, un lenguaje de máquina de pila conceptualmente idéntico, pero completamente desarrollado y no simbólico) vino a ser el revulsivo que faltaba para que Andoni Eguíluz y JosuKa Díaz decidieran desarrollar unas buenas prácticas de compiladores, y la necesidad de contar con un código intermedio real y funcional.

El primer desarrollo técnico de la máquina virtual fue llevado a cabo por Jesús Redrado durante el curso 1996-1997. En el siguiente curso, Jorge García (ayudante de las prácticas), realizó una de las labores más importantes, al construir el intérprete de la primera versión operativa, entonces llamada UDMPs98. Durante el curso 1998-1999, Ana Cabana primero, y Sergio Argüello y José David Vázquez después, realizaron diversas correcciones de errores, y añadieron diversos cambios de la especificación, para llegar a lo que fue la versión definitiva, llamada UDMPs99. El intérprete se escribió en C/C++, con un analizador léxico, obviamente, en FLEX, pero generando el *scanner* en C++. Lamentablemente, solo se compiló para DOS/WINDOWS, y la versión final tiene fecha de 19 de abril de 1999.

Una versión mejorada de esta máquina de pila se intentó desarrollar para servir a la implementación del lenguaje de programación en euskera llamado EPI [Guenaga *et al.* 2002], pero ahora mismo no recordamos que llegara a buen puerto...

Por otro lado, diversas circunstancias llevaron a la imposibilidad de continuar con el enorme esfuerzo que requerían aquellas prácticas de compiladores, por lo que el intérprete quedó “dormido” durante años... hasta que en 2006 volvieron a

retomarse, en forma mucho más sencilla. Enseguida se planteó el doble objetivo de eliminar errores del código y, sobre todo, hacerlo disponible para GNU/LINUX y otras máquinas.

Varias personas han hecho muy valiosos esfuerzos en este sentido, pero la poca disponibilidad de tiempo de JosuKa y Andoni para culminar el proceso hizo que no llegara a su final: Igor Ruiz en 2007 y David Ausín en 2008 aportaron mejoras considerables, recopiladas por JosuKa en una versión intermedia de octubre de 2009, pero que producía ciertos errores de ejecución que no pudieron resolverse (y que el tiempo ha demostrado que provenían de la primera versión).

Finalmente, en febrero de 2011, la gran ayuda de Endika Gutiérrez y Luis Rodríguez conduce a la solución de los errores conocidos (lo que no quiere decir que no haya otros desconocidos) y la posibilidad de “liberar” finalmente una versión multiplataforma (1.1β), probada en diversas versiones de WINDOWS y de GNU/LINUX, tanto de 32 bits como de 64 bits. Dicho lo de liberar en el sentido estricto del término, ya que esta versión se publica bajo la licencia GPL.

Nuestro agradecimiento por tanto a todas las personas que aquí se han mencionado, sin cuya ayuda este proyecto no habría sido posible (literalmente).

4.12.1. Referencias

- Alfred V. Aho, Ravi Sethi, Jeffery D. Ullman [1990] *Compiladores: principios, técnicas y herramientas*, Addison-Wesley Iberoamericana: México; traducción al castellano de Alfred V. Aho, Ravi Sethi, Jeffery D. Ullman [1986] *Compilers: Principles, Techniques, and Tools*, Addison-Wesley: Reading, Massachusetts.
- JosuKa Díaz Labrador [1996] *Implementación eficiente de la recursividad final en lenguajes imperativos basada en nuevas técnicas de compilación*, tesis doctoral, Universidad de Deusto.
- James Gosling, Bill Joy, Guy Steele [1996] *The Java Language Specification*, Addison-Wesley: Reading, Massachusetts.
- M^a Luz Guenaga, Andoni Eguíluz, JosuKa Díaz, David López, José Ignacio Herrero, Ricardo Casas, Iñaki Páramo, Daniel Yohn [2002] *EPI konpiladorea: gida didaktikoa*, EDEX: Bilbao.
- Tim Lindholm, Frank Yellin [1996] *The Java Virtual Machine Specification*, Addison-Wesley: Reading, Massachusetts.

4.13. Ejemplo de conversión de AJK a máquina de pila

4.13.1. Código fuente

```
int    e1, e2;
real  numr1;

function factorial( int n ) return (int)
{
    if (n == 1) return 1;
    else return n*factorial(n-1);
}

function potencia( real base; int exponente ) return (real)
int    i;
real  pot;
{
    i := 1;
    pot := 1;
    while (i <= exponente)
        pot := base * pot;
    return( pot );
}

function stirling (real n) return (real)
real pi, e;
{
    pi := 3.14159;
    e := 2.72;
    return( sqrt( 2*n*pi ) * (n/e)^n * (1+1/(12*n)) );
}

procedure main ()
int    numero, i;
int    fact;
real  stirn;
{
    numero := readint();
    i := 0;
    while (i < numero ) {
        fact := factorial( numero );
        stirn := stirling( numero );
        writeln( fact );
        writeln( stirn );
        writeln( stirn - fact );
        i := i + 1;
    }
    return;
}
```

4.13.2. Traducción a máquina de pila simbólica

```
;int  e1, e2;

      globali    e1          ; datos globales
      globali    e2

;real numr1;

      globalr    numr1

;function factorial( int n ) return (int)

      etiqi      factorial   ; función factorial
      parami     n           ; parámetro

      ponerbase          ; actualiza BASE
```

```

;   if (n == 1)
    valord      n
    insi        1
    ==
    si-falso-ir-a #elseif1
;
    return 1;
    valori      factorial ; empieza asignación valor retorno
    insi        1
    :=          ; termina asignación valor retorno
    ir-a        #factorial ; va a secuencia de retorno
;
    else
    ir-a        #finif1
    etiq        #elseif1
;
    return n*factorial(n-1);
    valori      factorial ; empieza asignación valor retorno
    valord      n
                                ; empieza llamada
    insi        ; espacio para valor de retorno
    valord      n
    insi        1
    -           ; parámetro (n-1) de llamada a factorial
    llamar      factorial ; termina llamada
    *
    :=          ; termina asignación valor retorno
    ir-a        #factorial ; va a secuencia de retorno
    etiq        #finif1
; }
    ir-a        #EXE001
    etiq        #factorial ; empieza secuencia de retorno
    cogerbases ; recuperar BASE
    ret         4           ; retorno desapilando parámetros
    fin         factorial ; aquí no debería llegar

;function potencia( real base; int exponente ) return (real)
    etiqr       potencia
    paramr      base
    parami      exponente
    ponerbase
;int i;
;real pot;
    locali      i           ; datos locales aquí (después de ponerbase)
    localr     pot
;{
;   i := 1;
    valori     i
    insi       1
    :=
;   pot := 1;
    valori     pot
    insi       1

```

```

    intareal
    :=r
;   while (i <= exponente)
    etiq      #iniWhile1
    valord    i
    valord    exponente
    <=
    si-falso-ir-a #finWhile1
;
    pot := base * pot;
    valori    pot
    valord    base
    valord    pot
    *r
    :=r
    ir-a     #iniWhile1
    etiq     #finWhile1
;   return( pot );
    valori    potencia
    valord    pot
    :=r
    ir-a     #potencia
; }
    ir-a     #EXE001
    etiq     #potencia
    desapilar 12      ; saca locales de la pila; también podría ser
                    ; desapilarr
                    ; desapilari
    cogerbaseret 12
    fin potencia
;function stirling (real n) return (real)
    etiqr     stirling
    paramr   n
    ponerbase
;real pi, e;
    localr    pi
    localr    e
; {
;   pi := 3.14159;
    valori    pi
    insr      3.14159
    :=r
;   e := 2.72;
    valori    e
    insr      2.72
    :=r
;   return( sqrt( 2*n*pi ) * (n/e)^n * (1+1/(12*n)) );
    valori    stirling      ; empieza asignación valor retorno
    insi      2
    valord    n

```

```

    cambiarir
    intareal
;   cambiarri          ; no es necesario por la conmutatividad
    *r
    valord      pi
    *r
    sqrt
    valord      n
    valord      e
    /r
    valord      n
    ^r
    *r
    insi        1
    insi        1
    insi        12
    valord      n
    cambiarir
    intareal
;   cambiarri          ; idem
    *r
    cambiarir
    intareal
    cambiarr
    /r
    cambiarir
    intareal
;   cambiarri          ; idem
    +r
    *r
    :=r          ; termina asignación valor de retorno
    ir-a        #stirling

;}

    ir-a        #EXE001
    etiq        #stirling      ; empieza secuencia de retorno
    desapilar  16             ; o también:      desapilarr
                                ;                       desapilarr

    cogerbases
    ret         8
    fin         stirling

;procedure main ()

    etiqv       main          ; es otro procedimiento más
    ponerbase

;int  numero, i;
;int  fact;
;real stirn;

    locali     numero        ; locales del programa principal
    locali     i
    locali     fact
    localr     stirn

;{
;   numero := readint();

    valori     numero
; puede ponerse lo siguiente:
;   escribirs "Escribe un número entero "
    leeri
    :=

;   i := 0;

```

```

    valori      i
    insi        0
    :=

;   while (i < numero ) {

    etiq        #iniWhile2
    valord      i
    valord      numero
    <
    si-falso-ir-a #finWhile2

;       fact := factorial( numero );

    valori      fact
    insi
    valord      numero
    llamar      factorial
    :=

;       stirn := stirling( numero );

    valori      stirn
    insr
    valord      numero
    intareal
    llamar      stirling
    :=r

;       writeln( fact );

    valord      fact
    escribiri
    escribirln

;       writeln( stirn );

    valord      stirn
    escribirr
    escribirln

;       writeln( stirn - fact );

    valord      stirn
    valord      fact
    intareal
    -r
    escribirr
    escribirln

;       i := i + 1;

    valori      i
    valord      i
    insi        1
    +
    :=

;   }

    ir-a        #iniWhile2
    etiq        #finWhile2

;   return;

    ir-a        #main

;}

```

```

    ir-a      #EXE001
    etiq      #main
    desapilar 36
    cogerbases
    ret
    fin      main

; empieza programa principal de máquina de pila

    inicio           ; aquí empezará la ejecución
    llamar    main   ; se llama al programa principal AJK
    fin              ; aquí se detiene la ejecución

; esta parte también se considera programa principal
; aunque parezca desconectada del resto

    etiq      #EXE001
    escribirln
    escribirs "Error EXE001"
    escribirln
    escribirs "Procedimiento o función no retorna"
    escribirln
    escribirs "Programa terminado"
    fin

```