Lenguaje MiniJavaC

I. Especificación del lenguaje

1. Objetivo del lenguaje

El objetivo es simplificar los lenguajes Java y C++, generando así un pequeño lenguaje que mezcle características de ambos y que resulte más intuitivo que estos. Su principal característica será la gran distinción de los bloques que irán introducidos entre corchetes **"[]"**.

El programa principal estará siempre formado por uno o varios de estos bloques, y además pueden ser anidados conteniendo un bloque otro bloque. El programa principal empezará con la palabra reservada **"begin"** y terminará cuando aparezca la palabra **"end"**, todo aquello que esté por fuera serán declaración de constantes, funciones o procedimientos.

2. Definición de bloques

El uso de las variables dentro de los bloques es parecido a C++ pero no igual, puesto que la declaración de las variables de un bloque se realizará al principio de éste. Si no se declaran al comenzar, el bloque en cuestión no tendrá variables locales, solamente podrá utilizar las que herede de los bloques que lo contengan.

Por lo tanto, los bloques que se encuentren al mismo nivel no podrán utilizar las variables de los demás bloques. Sin embargo, como ya hemos mencionado, un bloque puede utilizar las variables de los bloques en los que esté contenido, pero no al revés.

La declaración de variables irá entre llaves **"{}"** y empezará con la palabra reservada "declare" seguida de dos puntos **"declare:"**. Como hemos dicho antes, no será obligatorio poner una declaración de variables si ese bloque no va a usar variables locales. Solo se declararán variables de esta forma, no habrá otras instrucciones que sirvan para ello.

Después de la declaración de las variables a utilizar, lo siguiente que aparecerá en nuestro bloque serán instrucciones para realizar con las variables declaradas o nuevos bloques hijos contenidos en el bloque padre.

3. Tipos de instrucciones

Vamos a definir qué tipos de instrucciones están permitidas y cómo las vamos a implementar. La primera es la **asignación** que tratará de darle un valor a una variable, previamente declarada. En el lado de la izquierda pondremos el identificador de la variable seguido del símbolo **"->"**, y al lado derecho de la flecha el valor a asignar en forma de expresión aritmética o booleana según el tipo de variable. En el caso de un array, tendremos que poner en la parte del identificador el índice al que queremos acceder. Si queremos asignarle un valor a la posición i-ésima del array, lo escribiremos de la forma **"id[i]"** con i un entero.

La siguiente instrucción que vamos a comentar es el **if-else**, que implementaremos como en C++ o Java. No obstante, debemos precisar que en cada condición se ejecutará un bloque hijo de los mencionados anteriormente. Es decir, la instrucción tendrá la siguiente forma:

if (expresión booleana)
[Bloque Hijo 1]
else [Bloque Hijo 2]

En esta instrucción, el Bloque Hijo 1 se ejecutará si se cumple la expresión booleana y si no se ejecutará el Bloque Hijo 2. Además, hay que aclarar que la parte del **"else"** no es obligatorio ponerla, puede obviarse.

Las siguientes dos instrucciones son el **while** y el **for**, que van a tener una estructura bastante parecida y cuyo significado es el mismo que en C++ o Java. La primera la implementaremos de esta forma:

while (expresión booleana) do
[Bloque W]

Esta instrucción estará ejecutando el Bloque W hasta que la expresión booleana deje de cumplirse. Cada iteración del bucle se irá reiniciando las variables locales del Bloque W pero no las externas. El for tiene el mismo comportamiento en este aspecto. Escribiremos esta instrucción de la forma:

for (asignación; condición; asignación) do
[Bloque F]

Esta instrucción for es equivalente a un bloque tal que:

asignación;
while (expresión booleana) do [
[Bloque F]
asignación;
]

La última instrucción que vamos a poder utilizar es un **switch**, que será similar al de C++ pero sin usar la instrucción break, haremos un switch más sencillo. Tendrá la forma:

switch (idVarEntera) case {valores} : jump [B]
case {valores} : [B]
(...)
def : [B]

Entre paréntesis irá una variable entera con valor asignado y se ejecutarán todos los case cuyo valor coincida con el valor de esa variable, es decir, sus bloques. Además, en nuestro switch podremos poner una lista de valores entre llaves **"{}"** para todos los cuáles se entrará en el case correspondiente.

Por otro lado, introduciremos un nuevo término, que es **"jump"** y que se pondrá justo antes del bloque a ejecutar en el case correspondiente. Si entra en algún case que contiene esta palabra reservada, el switch dejará de comprobar el resto de case, ejecutará el bloque correspondiente y se terminará la instrucción.

Por último, aclaramos que existe un case especial que será el de por defecto y se ejecuta únicamente si no se ha podido ejecutar ningún otro case. Este empezaría con la palabra **"def"** en lugar de **"case"** y no es obligatorio introducirla.

4. Tipos de variables

En este apartado vamos a definir los tipos y las operaciones que podremos hacer con ellos. Disponemos de los tipos **int**, **float** y **double** para representar valores numéricos y **bool** para valores booleanos. También disponemos de los arrays de los tipos anteriormente nombrados y por consiguiente matrices, implementadas como arrays de arrays. Además, si queremos que una variable sea constante pondremos **"const nombreTipo idConstante = valor;"** y las constantes serán las únicas variables definidas fuera de cualquier bloque pues serán comunes a todo el programa.

Las variables de tipo se definen de la forma **"nombreTipo identificador"** y en caso de ser un array lo definiremos como **"nombreTipo[dimensión] identificador"**, todo ello en la parte reservada para la declaración de las variables. Si queremos declarar un array de varias dimensiones, declararemos cada una de las dimensiones que posea añadiendo corchetes. Por ejemplo, si queremos definir una matriz que tiene dos dimensiones pondremos **"nombreTipo[dimensión][dimensión] identificador"**. Si no se declara implícitamente el tipo de la variable se tomará el tipo **int** por defecto.

A continuación, vamos a ver cómo usar los operadores en las expresiones numéricas y booleanas en nuestro lenguaje. En las operaciones aritméticas, los operadores con mayor prioridad serán **"*"**, la multiplicación, y **"\"**, la división, y entre estos dos, elegiremos primero el que esté más a la derecha. El siguiente nivel de prioridad serán los operadores **"+"**, la suma, y **"-"**, la resta. No obstante, los paréntesis *"()"* tendrán prioridad sobre todos estos operadores.

Para los booleanos, disponemos de una gran variedad de operadores. Vamos a nombrar algunos de ellos teniendo en cuenta que donde ponemos entero puede ser una expresión entera y donde pone booleano puede ser una expresión booleana. Destacamos **"!!booleano"**, **"entero <= entero"**, **"booleano ^ / v booleano"**, **"XOR (booleano, booleano)"** y muchos más.

5. Funciones externas

Nos queda por ver cómo definiremos las funciones externas al programa principal. Las funciones serán llamadas por el programa principal y podrán devolver valores o no devolver nada. Las denotamos como **"fun nombreFunción recives [nombreTipo identificador, ...] returns nothing/nombreTipo]"**, y a continuación irán entre llaves **"{}"** las instrucciones que realizará la función.

La primera instrucción siempre será la declaración de las variables que utilizará y se realizará de la misma forma que en el programa principal, es decir, con el **"declare:"** entre llaves. Las funciones tendrán el mismo comportamiento que las funciones de C++. Una de las principales diferencias es que los valores que recibe la función comienzan con la palabra reservada **"recives"** y los tipos y nombres de las variables van a continuación entre corchetes **"[]"**. Además, el valor devuelto por la función se hará mediante la instrucción **"return idVariable;"** y será siempre la última instrucción de la función.

Por último, las funciones se llamarán desde el programa principal mediante **"call nombreFunción [idVariable, ...]"**. Si la función devuelve algún valor, este será asignado a una variable del programa principal y en caso contrario simplemente será llamada.

6. Ejemplos

En este apartado vamos a implementar dos pequeños ejemplos de dos posibles programas en nuestro lenguaje MiniJavaC.

Primer ejemplo:

```
const int A -> 100;
const bool B -> True;

fun suma recives [int a1, int a2] returns int{
 {declare: int res;}
   res -> a1 ++ a2;
   return res;
}

begin[
 {declare: int c; int d; int res; bool encontrado;}
   encontrado -> False; c -> 0; d -> 0;
   while (!!encontrado) do [
     c -> c * d ++ (2 ++ 3);
     d -> d ++ 1;
     c -> call suma [c,d];
     if (!! (c <= A)) do [
       encontrado -> B;
     ];
   ];
   res -> c/4;
]
end
```

Segundo ejemplo:

```
fun cuentaAtras recives [] returns nothing{
 {declare: int i; int c;}
   for (i -> 10, i > 0, i -> i -- 1) do [
     c -> i--1;
     i -> c;
   ];
}

begin[
 {declare: int x; int y;}
   for (x -> 0, x <= 2, x -> x ++ 1) do [
     switch (x) case 0: [y -> 5;]
                case 1: [y -> 3;]
                case 2: [y -> 2;];
   ];
   call cuentaAtras[];
]
end
```

II. Análisis de tipos

1. Cambios del lenguaje

A lo largo de la fase de análisis de tipos y del posterior paso a código máquina nuestro lenguaje ha sufrido varios cambios necesarios para simplificar ciertos aspectos. Vamos a proceder a mencionarlos y explicar dichas modificaciones.

En primer lugar, destacamos la desaparición de las instrucciones **"MAX"**, **"MIN"** y **"XOR"** cuya complejidad en código máquina nos han obligado a prescindir de ellas. Además, hemos cambiado las palabras clave utilizadas para los operadores **"AND"** y **"OR"** para simplificar los símbolos utilizados anteriormente.

Por otro lado, la instrucción **switch** (o case) ha sido modificada para poder ser compatible con la mencionada en los apuntes y tener más facilidades a la hora de representarla en código máquina. En efecto, hemos eliminado las instrucciones adicionales como **"jump"** o **"def"** y hemos impuesto que los valores que tome la variable del switch vayan de 0 a n consecutivos, es decir, no puede haber huecos entre los valores tomados. La instrucción quedaría así:

```
switch (idVarEntera) case 0 : [B]
                     case 1 : [B]
                     (...)
                     case n : [B]
```

Las funciones externas, al estar todas definidas dentro de lo que llamaríamos "Main Principal" están todas al mismo nivel y además tienen

la propiedad de poder llamarse las unas a las otras. También pueden ser recursivas, lo cual especificaremos más adelante.

Por último, hemos modificado la estructura de clases que teníamos anteriormente. En efecto, las expresiones y operadores van a ser representadas por dos clases del Árbol de tipos, **ExpresionUnaria** y **ExpresionBinaria**. La primera contendrá los tipos directos y los operadores unarios como la negación lógica. Por el contrario, la segunda representará todas las operaciones binarias y las expresiones que se sitúan en ambos lados.

2. Funcionamiento general

Vamos a proceder a explicar cómo realizamos el análisis de tipos. Para empezar, generamos la clase **FullBody** la cual se va a componer de las clases **FuncionesAndCtes** y **Programa** que representa al programa principal.

La primera clase tratará de crear dos listas. Una de ellas almacenará las funciones externas al programa principal junto con sus propiedades (parámetros, tipo a devolver,...) y sus correspondientes bloques internos con instrucciones. La segunda contendrá las variables constantes que junto con la expresión de su valor y el identificador correspondiente. Por otro lado, el programa principal constará únicamente de un bloque principal. En nuestro lenguaje, los bloques son conjuntos de instrucciones con entorno propio y heredable hacia arriba, por tanto no existen bloques vacíos sin instrucciones.

Además, en nuestro árbol, un bloque estará formado por un entorno local de variables que se definen al principio, tal y como indicamos en nuestra primera descripción del lenguaje. Por consiguiente, en dicho entorno se guardarán los identificadores de las variables locales. La otra parte del código es el conjunto de instrucciones y bloques hijos.

3. Herencia y redefinición de variables

Internamente, los bloques hijos reciben las variables heredadas del bloque padre que no estén redefinidas en los hijos. Además, también heredarán las variables que procedan de parámetros de funciones llamadas desde el bloque padre, salvo si igualmente han sido redefinidas. Exactamente lo mismo ocurre con la herencia de constantes. A continuación ilustramos la jerarquía previamente explicada:

Bloquehijo – hereda ->
BloquePadre – hereda ->
ParametrosFuncion – hereda ->
ConstantesGlobales

En el caso de que se redefina una variable que aparezca en todos los entornos se sigue la misma jerarquía:

Bloquehijo – redefineVariableDe ->
BloquePadre – redefineVariableDe ->

ParametrosFuncion – redefineVariableDe -> ConstantesGlobales

Sin embargo, la redefinición de variables solo se podrá realizar dentro del entorno local donde se defina. Cuando se abandone dicho bloque, la variable volverá a ser la del bloque padre o si se trataba de un parámetro de alguna función o quizás una constante, recuperará su valor inicial.

Nuestro análisis de tipos va a tener en cuenta todas estas posibles herencias y redefiniciones constantemente y representa una de las labores más costosas de la práctica. Al ser un analizador descendente, tendremos la suerte de que los primeros bloques creados serán los más profundos. Por lo tanto, las variables utilizadas en expresiones o instrucciones en un bloque dado serán la primera variable del entorno con la que se encuentren.

En efecto, empezamos con el análisis local y vamos subiendo hacia arriba hasta llegar a la clase **FullBody** que es la más alta. Esta clase se encargará de pasar las funciones y constantes menos prioritarias, es decir, las más globales con el fin de ir controlando los identificadores de las expresiones. Estos identificadores se retransmitirán hacia abajo para buscar algún bloque hijo que pueda heredarlas, es decir que las utilice y no las haya redefinido. Las funciones también se pasan hacia abajo y además, cuando se utiliza la instrucción **"call"** verifica si la función a la que llama tiene el tipo correcto y existe.

4. Resumen

En resumen, empezamos analizando los bloques más profundos, y vamos subiendo a través de los padres hasta llegar al bloque principal. En el caso de los bloques correspondientes a funciones externas, subimos hasta llegar al nivel menos profundo. Después volvemos a bajar realizando el paso de los parámetros a los bloques contenidos en la función.

Una vez llegado a este punto se hará el paso de las constantes y de los datos de las funciones externas hacia abajo, es decir del bloque principal del programa hasta los hijos más profundos. Este mismo trabajo se realiza también en todos los posibles bloques de las funciones externas ya que pueden usar constantes y llamar a otras funciones.

III. Paso a código máquina

1. Procedimiento general

Lo primero que se debe aclarar es que, en el caso de las funciones externas, la máquina-P asigna el mismo nivel de profundidad al ámbito de definición que al de uso. Por lo tanto, deducimos que nuestra máquina siempre le asignará la misma profundidad a dichos ámbitos, en cuánto a una función se refiere.

Una vez aclarado esto, vamos a proceder a explicar como se realiza la traducción a código máquina. Además trataremos de aclarar cómo, con

ayuda del árbol, somos capaces de lograr que las funciones puedan ser recursivas y llamarse entre ellas.

En primer lugar, comenzamos la traducción calculando el espacio que van a necesitar las variables de nuestro programa principal y ajustamos nuestro SP con ayuda de la instrucción **"ssp"**. Hacemos lo mismo con la longitud máxima de la pila de expresiones utilizando la instrucción **"sep"**. Después, definimos todas las constantes y les asignamos su valor correspondiente.

Una vez hecho esto, se marca una posición en la que se introduce el salto al cuerpo del programa principal. Tras dicho salto procedemos a poner los prólogos de cada una de las funciones externas al programa principal de forma consecutiva, es decir, introducimos su **"ssp"**, su **"sep"** así como el **"ujp"** al cuerpo de la función.

Gracias a esto, todas las funciones quedarán registradas y se podrán llamar las unas a las otras lo cual enriquece mucho nuestro lenguaje. A continuación, vendrán todos los códigos de las funciones de forma consecutiva y por último, el código del main. A este último se saltará directamente durante la ejecución del programa gracias al salto definido tras la declaración de constantes.

2. Registro de variables

Con respecto al registro de las variables, necesitamos ir desde el entorno más externo al más interno, es decir completamente al revés a lo realizado en el análisis de tipos. Iremos ajustando primero las direcciones de las variables más externas y posteriormente las de las más internas.

Además, a la hora de redefinir variables, lo que hacemos en realidad es crear en memoria una nueva variable completamente distinta pero con el mismo identificador. Cuando el programa utilice dicha variable accederá a la nueva en vez de a la exterior, quedando de este modo redefinida. Si la variable es exterior y no es redefinida se accederá directamente a la posición asignada en el entorno exterior.

Por otro lado, cuando se definen las constantes, su posición se encuentra en la parte inicial de la memoria a partir de la posición 5. Estas serán las primeras localizaciones del bloque principal y podrán acceder a ellas todas las funciones.

Por último, destacamos que el **case** ha sido definido usando la instrucción de salto **"ixj"** calculado tal y como se indica en los apuntes. En lo que respecta al resto de instrucciones, siguen el esquema mostrado en los apuntes, no hay nada que destacar.

3. Posibles cambios

Debido a la falta de tiempo y a ciertas complejidades, existen varios cambios que no hemos llegado a realizar pero que nos hubiese gustado con el único fin de dotar al código de funcionalidad, elegancia y legibilidad.

En primer lugar, cuando se accede a un **"array"**, nuestro código no admite en las dimensiones variables previamente definidas, sólo nos permite introducir una dirección con un número exacto. Por ejemplo, array[3][2]. En efecto, no podemos poner un identificador entre los corchetes debido a que en el cup no especificamos que la dimensión pudiera ser también una expresión, simplemente un integer.

Otra modificación interesante es añadir paso de variables singulares por referencia. Al implementar el paso de parámetros para los **"arrays"** hemos podido comprobar lo sencillo que podía llegar a ser y la cantidad de buenas propiedades que esto aporta a nuestro lenguaje.

4. Ejemplos

Vamos a mostrar el código máquina generado por uno de los programitas anteriores que pusimos de ejemplo para que quede reflejado todo lo dicho anteriormente.

Código de ejemplo:

```
const int A -> 100;
const bool B -> True;

fun suma recives [int a1, int a2] returns int{
 {declare: int res;}
   res -> a1 ++ a2;
   return res;
}

begin[
 {declare: int c; int d; int res; bool encontrado;}
   encontrado -> False; c -> 0; d -> 0;
   while (!!encontrado) do [
     c -> c * d ++ (2 ++ 3);
     d -> d ++ 1;
     c -> call suma [c,d];
     if (!! (c <= A)) do [
       encontrado -> B;
     ];
   ];
   res -> c/4;
]
end
```

Código máquina generado:

```
{0}  ssp 11;
{1}  sep 4;
{2}  ldc 5;
{3}  ldc 100;
{4}  sto;
{5}  ldc 6;
{6}  ldc true;
{7}  sto;
{8}  ujp 25;
{9}  ssp 8;
{10}  sep 3;
{11}  ujp 12;
```

```
{12}  lda 0 7;
{13}  lda 0 5;
{14}  ind;
{15}  lda 0 6;
{16}  ind;
{17}  add;
{18}  sto;
{19}  lda 0 0;
{20}  lda 0 7;
{21}  ind;
{22}  sto;
{23}  retf;
{24}  retp;
{25}  lda 0 9;
{26}  ldc false;
{27}  sto;
{28}  lda 0 8;
{29}  ldc 0;
{30}  sto;
{31}  lda 0 10;
{32}  ldc 0;
{33}  sto;
{34}  lda 0 9;
{35}  ind;
{36}  not;
{37}  fjp 75;
{38}  lda 0 8;
{39}  lda 0 8;
{40}  ind;
{41}  lda 0 10;
{42}  ind;
{43}  mul;
{44}  ldc 2;
{45}  ldc 3;
{46}  add;
{47}  add;
{48}  sto;
{49}  lda 0 10;
{50}  lda 0 10;
{51}  ind;
{52}  ldc 1;
{53}  add;
{54}  sto;
{55}  lda 0 8;
{56}  mst 0;
{57}  lda 0 8;
{58}  ind;
{59}  lda 0 10;
{60}  ind;
{61}  cup 2 9;
{62}  sto;
{63}  lda 0 8;
{64}  ind;
{65}  ldc 5;
{66}  ind;
{67}  leq;
{68}  not;
{69}  fjp 74;
{70}  lda 0 9;
{71}  ldc 6;
{72}  ind;
{73}  sto;
{74}  ujp 34;
{75}  lda 0 7;
{76}  lda 0 8;
{77}  ind;
{78}  ldc 4;
{79}  div;
{80}  sto;
{81}  stp;
```

Juan Vicente Guillén Casas

Isabel Pérez Pereda