PROCESADORES DEL LENGUAJE

Doble grado de Ingeniería Informática y Matemáticas

HaskPlus

Lucía Alonso Mozo

Javier Amado Lázaro

Curso 2023/24

Índice

1. Introducción

El documento se centra en definir la sintaxis de un nuevo lenguaje de programación llamado *HaskPlus*, el cual fusiona los paradigmas de *Haskell* y *C++*. Esta combinación busca aprovechar las ventajas tanto de la programación funcional como de la orientada a objetos. El objetivo es proporcionar a los programadores una herramienta versátil que combine la elegancia funcional con la eficiencia de *C++*. En este estudio, exploraremos la estructura sintáctica de *HaskPlus*, identificando sus elementos esenciales y delineando las reglas gramaticales para su implementación y uso en el desarrollo de aplicaciones modernas. Además, se proporcionarán ejemplos prácticos para ilustrar cómo utilizar el lenguaje de manera efectiva.

2. Estructura básica de un programa en HaskPlus

Como nos hemos basado en una mezcla entre *C++* y *Haskell*, la sintaxis será muy intuitiva. En las primeras líneas del código se encontrarán las declaraciones de los alias y los Struct, seguidas de las definiciones de las funciones. Como inicio de la ejecución del programa es importante destacar el *main*, que será de la siguiente forma:

```
main() -> Int {
    // codigo
    return 0;
}
```

La forma de hacer comentarios será usando // y después un texto detrás. Si queremos hacer comentarios de mayor longitud o más de una línea usaremos /**\ .

3. Tipos

En esta sección vamos a introducir los tipos que tiene a su disposición el usuario en *HaskPlus*, así como la sintaxis necesaria para declararlos. Cabe destacar que se deben declarar los tipos de manera explícita.

3.1. Tipos básicos

Los tres tipos básicos predefinidos que presenta el lenguaje de *HaskPlus* son:

- Int: representa los números enteros.
- Bool: puede tomar los valores $True$ o $False$.
- Void: para representar el tipo vacío.

Observación 3.1. El tipo Void se utiliza exclusivamente para funciones que no devuelvan ningún valor, y por tanto devolverán el tipo Void.

3.2. Tipos definidos por el usuario

3.2.1. Tipo struct

En *HaskPlus* se permite crear estructuras o registros para albergar distintas variables específicas dentro de una variable más genérica:

- Struct: estructura de datos formada por distintos campos que son variables de uno o distintos tipos.

La sintáxis para declarar este tipo sería la siguiente:

```
Struct nReg {
    tipo1 var1;
    tipo2 var2;
    //lista de variables
};
```

3.2.2. Tipo array

En cuanto a los arrays, en *HaskPlus* se usa la sintaxis tipo $nombreArray[TamanoArray]$, donde primero se declara el tipo que tendrá el array, y después el nombre del mismo seguido de corchetes entre los que se especifica el tamaño que va a tener el array, que debe ser un número entero.
Es importante señalar que los arrays en *HaskPlus* son homogéneos, es decir, todos los elementos de un array deben tener el mismo tipo. A su vez, se permiten arrays de varias dimensiones, es decir, arrays cuyos elementos son arrays a su vez. Un ejemplo de array de más de una dimensión son las matrices, cuya representación sería tipo $matriz[Tamano1][Tamano2]$.
Los arrays de una dimensión se pueden inicalizar de las siguientes formas en *HaskPlus*:

- Utilizando alguno de los bucles que presentaremos a continuación para inicializar posición por posición el array de la siguiente manera $array[i] = valor$.
- Int array[5] = [1,2,3,4,5]. Asignando a cada posición del array una serie de valores escritos entre corchetes y separados por comas.
- Int array[5] = [1..5]. Esta forma solo es aplicable cuando el array es de enteros y es útil cuando se desea inicializar un array con una serie extensa de valores comprendidos entre dos enteros.

Sin embargo, los arrays de dos dimensiones o más, se inicializan mediante el primer formato descrito anteriormente para los arrays de una dimensión, es decir, mediante un bucle.

3.2.3. Tipo puntero

En *HaskPlus*, los punteros representan una dirección de memoria que almacena un valor. Para declarar un puntero, se utiliza la sintaxis tipo * $nombrePuntero$. Posteriormente, para acceder al valor almacenado en la dirección de memoria apuntada por el puntero, se emplea $*nombrePuntero$. Por ejemplo, mediante la expresión $*nombrePuntero = 3$, inicializamos el valor asociado a un puntero de tipo Int a 3. Por otro lado, si se desea obtener la dirección de memoria del puntero, se utiliza $\&nombrePuntero$.

4. Expresiones, operadores y bloques

4.1. Expresiones

En *HaskPlus*, las expresiones pueden ser:

- **Constantes de tipo**:
  - Int: número entero.
  - Bool: cierto o falso.

```
const Int nEntero = valor1;       //declaracion de constantes para el tipo Int
const Bool nBoolean = valor2;     //declaracion de constantes para el tipo Bool
```

- **Accesos a**:
  - Variables: mediante el nombre o identificador de la variable, que estará formado por caracteres alfanuméricos y guiones bajos. Los identificadores de las variables deben empezar siempre por una letra ya sea mayúscula o minúscula.
  - Arrays: usaremos la siguiente sintaxis para acceder al elemento del array situado en la posición determinada por los índices:

```
tipo nArray[tamanoArray];  //declaramos el array
nArray[ind1];              //accedemos a la posicion ind1 del array
```

  - Structs: para acceder a los distintos campos de un registro utilizaremos el '.' de la siguiente manera:

```
Struct nReg{    //declaramos el struct
    tipo1 campo1;
    tipo2 campo2;
};

nReg var;       //declaramos una variable de ese tipo
var.campo1;     //para acceder al campo1 del struct
var.campo2;     //para acceder al campo2 del struct
```

  - Punteros: mediante * accedemos al valor asociado que guarda el puntero y mediante & accedemos a la dirección de memoria del mismo.

```
tipo * nPuntero;       //declaramos el puntero
*nPuntero;             //para obtener el valor asociado al puntero
&nPuntero;             //para obtener la direccion de memoria del puntero
```

Cabe destacar que los distintos tipos de accesos que hemos presentado anteriormente pueden estar anidados. Por ejemplo, puede haber un Struct donde uno de sus campos sea un array, o también puede haber un array de Struct.

4.2. Operadores

En cuanto a los operadores de *HaskPlus*, distinguimos tres tipos, que aparecen clasificados segun su identificador, tipo, asociatividad y prioridad (siendo 0 el menos prioritario y 7 el más prioritario):

- Operadores Aritméticos:

Operador	Identificador	Tipo	Asociatividad	Prioridad
Suma	+	Binario infijo	Por la izquierda	4
Resta	-	Binario infijo	Por la izquierda	4
Multiplicación	*	Binario infijo	Por la izquierda	5
División	/	Binario infijo	Por la izquierda	5
Módulo	mod	Binario infijo	Por la izquierda	5
Division entera	div	Binario infijo	Por la izquierda	5

- Operadores Booleanos:

Operador	Identificador	Tipo	Asociatividad	Prioridad
Mayor	>	Binario infijo	Por la izquierda	3
Menor	<	Binario infijo	Por la izquierda	3
Mayor o igual	>=	Binario infijo	Por la izquierda	3
Menor o igual	<=	Binario infijo	Por la izquierda	3
Igual	==	Binario infijo	Por la izquierda	2
Distinto	! =	Binario infijo	Por la izquierda	2
And	&	Binario infijo	Por la izquierda	1
Or	|	Binario infijo	Por la izquierda	0
Not	not	Unario prefijo		6

- Operadores en Arrays:

Operador	Identificador	Tipo	Asociatividad	Prioridad
Sumatorio	sum	Unario prefijo	Por la izquierda	7
Productorio	prod	Unario prefijo	Por la izquierda	7
Concatenación	++	Binario infijo	Por la izquierda	7

4.3. Bloques

En *HaskPlus*, los bloques se encuentran delimitados por llaves { y }. Al ingresar en un bloque, el programa ajusta su ámbito al del bloque en el que se ha adentrado. Dentro de un bloque, únicamente se permite el acceso a las variables que se hallan dentro de dicho bloque o en bloques más externos. Las variables declaradas dentro de un bloque reemplazan a aquellas declaradas en bloques externos en caso de coincidir en nombre. Es factible que varios bloques estén anidados, y en tal situación, el ámbito del bloque más profundo prevalecerá. La aparición de bloques estará invariablemente precedida por una estructura de control, como por ejemplo while, if...

5. Instrucciones

En esta sección se exponen las instrucciones fundamentales que caracterizarán el lenguaje HaskPlus. Es oportuno señalar que hemos determinado que la terminación de cada instrucción se represente mediante el símbolo ; con el objetivo de favorecer la simplicidad y promover la uniformidad con otros lenguajes prominentes.

5.1. Declaración

La forma de declarar una variable tendrá dos posibilidades.

- **Sin inicializar**: tipo variable;
  Bool encontrado;
- **Inicializadas**: tipo variable = valor/expresión.
  Int num = 1;

Observación 5.1. Es importante destacar que las variables de tipo entero Int, si no están inicializadas, adquieren por defecto el valor 0. Del mismo modo, las variables de tipo Bool adoptan el valor $False$ si no se les ha asignado ningún valor inicial.

5.2. Asignación

La metodología de asignación seguirá el estándar establecido en los principales lenguajes de programación. La variable que se desea definir se ubicará en el lado izquierdo del signo de igualdad, mientras que en el lado derecho se especificará el valor que se le asignará. Es crucial que ambas expresiones posean el mismo tipo de dato.

$$variable = expresión.$$

Además, considerando $\oplus$ como un operador aritmético previamente definido, se ofrece la opción de asignar valor a una variable de la siguiente manera:

$$variable\_entero \oplus= variable\_entero2$$

En secciones posteriores del documento se analizará la correcta utilización y las restricciones asociadas al empleo de estos operadores.

Observación 5.2. No se permitirá el uso de operaciones como *i++* o *i- -*. En tales casos, se deberá escribir la expresión completa (*i = i+1* o *i = i-1*), o emplear el método abreviado recién explicado (*i += 1* o *i -= 1*), que ofrecerá la misma funcionalidad.

5.3. Condicional

Para las condiciones, hemos decidido poner el tope en un máximo de dos únicas ramas por condición.

Si es de una rama sería:

```
if(exp) {
 //codigo
}
```

O de dos ramas, en los que se considerará que si $[\![exp]\!] = true$ se hará $codigo1$ y si $[\![exp]\!] = false$, se hará $codigo2$.

```
if(exp) {
 //codigo1
} else{
 //codigo2
}
```

5.4. Operaciones con operadores

Como se ha definido previamente en la sección 4.2, se presentan distintos tipos de operadores, incluyendo aritméticos, booleanos y aquellos aplicados a arrays.

- **Operadores Aritméticos**: Las operaciones se realizan entre números enteros (Int). Considerando n_1, n_2 como enteros y $\oplus$ como un operador aritmético, todas las expresiones adoptarán la forma $n1 \oplus n2$. Es importante destacar que $n2 \neq 0$ cuando $\oplus = /, div$.
- **Operadores Booleanos**: Los operadores And, Or y Not operarán sobre valores booleanos (Bool). Los demás operadores booleanos compararán enteros.
- **Operadores en Arrays**: Los operadores sum y $prod$ aplicarán operaciones sobre listas, mientras que la concatenación se llevará a cabo mediante el operador ++. Es decir, si l_1 y l_2 son dos listas, la concatenación se expresará como l_1 ++ l_2.

Observación 5.3. Cuando se utiliza el operador de concatenación de listas, es esencial verificar el tamaño de ambas listas que se desean concatenar. Esta verificación es necesaria para poder declarar el nuevo array con un tamaño que sea el resultado de la concatenación de los dos arrays anteriores.

5.5. Bucles

5.5.1. Bucle While

El bucle while consistirá en poner una expresión exp de forma que $mientras$ $[\![exp]\!] = true$ se dará vueltas, haciendo el código que esté entre llaves. Como condición de terminación tenemos que $[\![exp]\!] = false$.

```
while(exp){
 //codigo
}
```

5.5.2. Bucle For

El bucle *for* es la misma idea de ejecutar un código un número de veces dependiendo de las condiciones de parada (*expresión*).

```
for(declaracion; expresion; asignacion){
 //codigo
}
```

5.5.3. Bucle Repeat

Hemos añadido una ampliación a los tipos de bucle llamada *repeat*. Este consistirá en un bucle para dar n vueltas sin necesidad de definir un iterador. La condición de parada será cuando se hayan dado las n vueltas. Si queremos hacer un bucle infinito bastará con poner $n = Inf$.

```
repeat(n){
 //codigo
}
```

5.6. Switch

HaskPlus implementa también los *Switch-case* de forma parecida a *Haskell* usando las guardas. También implementaremos el caso por defecto así como el *break*.

```
switch(var){
| e1: //codigo
...
| en: //codigo
| default: //codigo
}
```

5.7. ValueFor

HaskPlus implementa una combinación entre los *if's* y los *switch* a los que llamaremos *valuefor*. Este consistirá en una asignación donde *var* pasará a valer un a_i dependiendo del e_i que sea cierto. Como en el *switch* implementaremos un valor por defecto y el *break*.

```
valuefor(var) {
| e1 = a1;
...
```

```
| en = an;
| default = a;
}
```

5.8. Alias

Permitimos el uso de alias igual que en *C++* y *Haskell*. La sintaxis será de la siguiente forma:

alias nombre = tipo ;

5.9. Entrada y Salida

Considerando la fusión de nuestros respectivos lenguajes, hemos establecido que la impresión en pantalla se realice mediante el comando show, mientras que la lectura se efectúe mediante read. Se restringe la lectura a valores del tipo Int, mientras que en la impresión se admitirán tanto valores Int como Bool. En el caso de los valores booleanos, se representarán como 0 si son falsos y como 1 si son verdaderos.

```
show(3); //mostrar  un 3 por pantalla.
show(True); //mostrar  un 1 por pantalla.
```

Listing 1: Impresión por pantalla

```
Int n = read();
```

Listing 2: Lectura de entero

6. Funciones

Para definir una función aplicaremos el siguiente esquema.

```
nombre (a:tipoA, b:tipoB) -> tipoC {
 //codigo
}
```

La función se denominará nombre y aceptará parámetros, en este caso, a y b, los cuales serán de los tipos tipoA y tipoB, respectivamente. El tipo de retorno de la función será tipoC.

Siguiendo la convención de *C++* respecto al paso de parámetros por valor o por referencia, distinguiremos entre estos dos métodos al pasar argumentos a la función. El paso por valor se llevará a cabo según el esquema general proporcionado previamente, mientras que el paso por referencia se indicará mediante el uso del símbolo &.

```
nombre (a:&tipoA, b:&tipoB) -> tipoC {
 //codigo
}
```

Listing 3: Paso por referencia

También permitiremos que las funciones retornen más de un tipo a la vez, esto se hará sustituyendo tipoC por (tipoC, tipoD).

6.1. Llamadas a función

Para las llamadas a las funciones aplicaremos la sintaxis ya existente de *C++*. Podemos realizar las llamadas a funciones en cualquier parte de una expresión. Si por ejemplo queremos guardar el valor de retorno de la función haremos:

tipo nombre = funcion (a,b);

6.2. Retorno de función

Para hacer el retorno de la función usaremos la sintaxis que nos dice la intuición, *return*. Este deberá ser la última línea que queremos que se ejecute de la función.

```
nombre (a:tipoA, b:tipoB) -> tipoC {
 //codigo
 tipoC n = a1;
 return n;
}
```

Observación 6.1 Es relevante destacar que en *HaskPlus* se admite la recursividad. En este contexto, una función puede invocarse a sí misma dentro de su propio cuerpo.

7. Ejemplos

A continuación se proporciona una lista de ejemplos, así como una descripción de lo que se evalúa en cada uno de ellos.

```
main() -> Int{
    Int varI;           //declaracion de variables enteras
    varI = 2;           //asignacion de un valor a una variable entera
    Bool varB,varB2;    //declaracion de variables booleanas
    varB = False;       //asignacion de un valor a una variable booleana
    varB2 = varB;       //asignacion del valor de la variable varB a la variable varB2
    return 0;
}
```

Listing 4: Declaración y asignación de valores a variables de tipos Int y Bool

```
main() -> Int{
    Int var1 = 2 * 3 + 6 / 6;       //var1 = 6 + 1 = 7
    Int var2 = (2 * 3 + 6) / 6;     //var2 = 12 / 6 = 2
    Int var3 = (2 * (3 + 6)) / 6;   //var3 = 18 / 6 = 3
    Int var4 = (4 * 5) mod 8;       //var4 = 4
```

```
    Int var5 = 4 + 5 mod 8;          //var5 = 4 + 5 = 9
    Int var6 = -10 div 5;            //var6 = -2
    Int var7 = 3 - 20 div 5;         //var7 = -1
    return 0;
}
```

Listing 5: Ejemplos de expresiones con operadores aritméticos.

```
main() -> Int{
    Int var1 = 4;
    Int var2 = 3;
    Bool varB1 = False & False | True                    //varB1 = True
    Bool varB2 = (False & False) | True                  //varB2 = True
    Bool varB3 = var1 > var2 | var1 == var2              //varB3 = True
    Bool varB4 = var1 <= var2 & True                     //varB4 = False
    Bool varB5 = var1 != var2 | var1 > var2 & False      //varB5 = False
    Bool varB6 = var1 != var2 | (var1 > var2 & False)    //varB6 = True
    Bool varB7 = not True == False & var2 < var1         //varB7 = True
    Bool varB8 = not (True != False) & var2 < var1       //varB8 = False
    return 0;
}
```

Listing 6: Ejemplos de expresiones con operadores booleanos.

```
main() -> Int{
    Int array1[5] = [1,2,3,4,5];
    Int array2[4] = [2,1,3,2];
    Int var1 = sum array1;                      //var1 = 15 (sumatorio).
    Int var2 = prod array1;                     //var2 = 12 (productorio).
    Int var3 = sum array1 + prod array2;        //var3 = 15 + 12 = 27
    Int  array3[9] = array1 ++ array2;          //[1,2,3,4,5,2,1,3,2].
    return 0;
}
```

Listing 7: Ejemplos de expresiones con operadores de arrays.

```
main() -> Int{
    Int x = 5;              //Variable en el ambito global
    show(x);                //Se imprimira un 5
    while (x > 0) {         //Comienza un bloque con el bucle
        Int y = 10;         //Variable en el ambito del bucle
        Int x = 6;
        show(x);            //Se imprimira un 6
        if (y > 5) {        //Comienza un bloque con el condicional
            Int z = 20;     //Variable en el ambito del condicional
        }                   //Fin del bloque del condicional
    }                       //Fin del bloque del bucle
    return 0;
}
```

Listing 8: Ejemplo de bloques anidados y del cambio de ámbito del programa en cada uno de ellos

```
main() -> Int{
    Int var1 = 2;
    Int * var2 = &var1; //asignamos a la direccion de memoria del puntero var2 la de la
    variable var1
    Int var3 = *var2; //var3 = 2
    Int * var4 = var2; //direcci n de memoria del puntero var4 la de var2
    Bool varB1 = False;
    Bool * varB2 = &varB1; //asignamos a la direccion de memoria del puntero varB2 la de la
     variable varB1
    Bool * varB3 = varB2; //asignamos a la direccion de memoria del puntero varB3 la de la
    variable varB2
    Bool varB4 = not(*varB2) & not(*varB3)  //varB4 = True
    return 0;
}
```

Listing 9: Punteros.

```
main() -> Int{
    Int array1[6] = [1..6]; //inicializamos el array con valores del 1 al 6
    Int array2[5] = [1,4,6,7,8];
    Int array3[7];  //inicializamos el array mediante un bucle
    for(Int i = 0; i < 7; i = i + 1){
        array3[i] = i;
    }
    Int var1 = array1[2] + array2[4];           //var1 = 3 + 8 = 11
    Bool arrayB1[2] = [True,False];
    Bool varB1 = arrayB1[1];                    //varB1 = False
    Int arrayMult[7][6];                        //declaraci n de un array de 2 dimensiones
    for(Int i = 0; i < 7; i = i + 1){ //usamos dos bucles anidados
        for(Int j = 0; j < 6; j = j + 1){
            arrayMult[i][j] = j;
        }
    }
    Int arrayMult3[7][6][5]; //declaraci n de un array de 3 dimensiones
    for(Int i = 0; i < 7; i = i + 1){ //usamos tres bucles anidados
        for(Int j = 0; j < 6; j = j + 1){
            for(Int k = 0; k < 5; k = k + 1){
                arrayMult3[i][j][k] = k;
            }
        }
    }
    return 0;
}
```

Listing 10: Arrays.

```
Struct tInfo{                    //declaramos el struct
    Int var1;
    Int var2;
    Bool varB1;
    Bool varB2;
};
main() -> Int{
    Int var3 = 3;
    Bool varB4 = True;
    tInfo str;
    str.var1 = var3;                //asignamos la var3 al campo var1 de str
    str.varB1 = not varB4;          //asignamos not varB4 al campo varB1 de str
    tInfo str2 = str;               //asignamos la variable entera str a str2
    return 0;
}
```

Listing 11: Structs

```
calculaMax(arr:Int[], length:Int) -> Int{
    Int max = arr[0];
    for(Int i = 0; i < length; i = i + 1){
        if(arr[i] > max){
            max = arr[i];
        }
    }
    show(max);                   //mostramos el maximo
    return max;
}
main() -> Int{
    Int length = 5;
    Int array[5] = [1,2,3,4,5];
    Int sol = calculaMax(array,length);
    return 0;
}
```

Listing 12: Consiste en encontrar el máximo en un *array* de enteros.

```
factorial (n:Int) -> Int {
    Int fact = 1;
    for (Int i=1 ; i<=a ; i = i + 1) {
        fact=i*fact;
    }
    show(fact);
    return fact;
}
main() -> Int{
    Int n = 5;
    Int sol = factorial(n);
    return 0;
```

```
}
```

Listing 13: Función que calcula el factorial de un número dado.

```
calculaMedia(arr:Int[], n:Int) -> Int{
    Int i = 0;
    Int suma = 0;
    if (n > 0) {
        repeat n {
            suma = suma + arr[i];
            i = i + 1;
        }
        return suma / n;
    } else {
        return 0;
    }
}
main() -> Int{
    Int length = 5;
    Int array[5] = [1,2,3,4,5];
    Int sol = calculaMedia(array,length);
    show(sol);
    return 0;
}
```

Listing 14: Cálculo de la media de un array.

```
calculaFibonacci(n: Int) -> Int{
    Int res;
    if (n <= 1) {
        res = 1;
    } else {
        res = fibonacci(n - 1) + fibonacci(n - 2);
    }
    return res;
}
main() -> Int{
    Int n = 5;
    Int sol = calculaFibonacci(n);
    show(sol);
    return 0;
}
```

Listing 15: Función que calcula el $n - ésimo$ número de la serie de Fibonacci.

```
main() -> Int {
    Int x = 3; //inicializamos x a 3
    if(x == 3){
        x = 5;
```

```
    }
    show(x);
    Bool ok = False;
    if(ok){
        x = 7;
    }
    else{ //debe entrar aqui y cambiar el valor de x a 1
        x = 1;
    }
    show(x);
    if(not ok) { //debe entrar aqui y  cambiar el valor de x a 2
        x = 2;
    }
    show(x);
    return 0;
}
```

Listing 16: Ejemplos del uso del $if - else$ con una o dos ramas.

```
main() -> Int {
   Int x = 0;
   Int maximo = 10;
   Bool siempre = True;
   show(x);
   while(x < 10 & siempre){ //el bucle debera dar 10 vueltas
      x = x+1;
   }
   show(x); //debe imprimir como valor de x 10
   return 0;
}
```

Listing 17: Ejemplos de *while* con condiciones booleanas y comparaciones de enteros.

```
main() -> Int {
   Int N = 5;
   Int x = 0;
   for(Int i = 0; i < N; i = i +1 ){
     x = x +1;
   }
   show(x); //debe mostrar como valor de x el 5.
   return 0;
}
```

Listing 18: Ejemplo del típico uso del for como incrementador.

```
main() -> Int {
   Int x = 10;
   repeat(10){
     x = x - 1;
```

```
    }
    show(x); // x debe valer 0
    return 0;
}
```

Listing 19: Ejemplo de *repeat* para dar un número n de vueltas.

```
main() -> Int {
    Int x = 2;
    Int resultado;
    switch(x) {
     |0: resultado = 0;
         break;
     |1: resultado = 1;
         break;
     |2: resultado = 2;
         break;
     |default: resultado = 10;
    }
    show(resultado); //resultado debe valer 2.
    return 0;
}
```

Listing 20: Ejemplo del *switch* donde la variable a diferenciar casos es un Int.

```
main() -> Int {
    Int x = 2;
    Int resultado;
    valuefor(resultado){
     | (x < 2) = 0;
     | (x > 2) = 1;
     | (x == 2) = 2;
     | default = 3;
    }
    show(resultado); //resultado debe valer 2.
    return 0;
}
```

Listing 21: signamos valor a una variable de tipo Int usando el *valuefor*.

```
alias Enterito = Int; //Enterito es un sinonimo de Int
suma(a:Enterito, b:Enterito) -> Enterito {
    return a + b;
}
main() -> Int{
    Int a = 3;
    Enterito b = 4;
    show(suma(a,b)); //debe imprimir 7
    return 0;
```

```
}
```

Listing 22: Ejemplo de alias y operación con ellos.

```
main()-> Int {
    // Ambos deberan imprimir los valores que el usuario introduzca por teclado.
    Int a = read();
    show(a);
    Int b = read();
    show(b);
    return 0;
}
```

Listing 23: Lectura y escritura de Int.

```
// Paso por valor
incrementarPorValor(num:Int) -> Void {
    num = num + 1;
}
// Paso por referencia
incrementarPorReferencia(num:&Int) -> Void{
    num = num + 1;
}
main() -> Int{
    Int numero = 1;
    show(numero);
    // Paso por valor
    incrementarPorValor(numero);
    show(numero); //el numero debe mantenerse igual
    // Paso por referencia
    incrementarPorReferencia(numero);
    show(numero);//el n mero debe cambiar (haberse incrementado)
}
```

Listing 24: Diferencias entre el paso por valor o por referencia de los parámetros de una función.

```
//Errores Sintacticos
Struct tTiempo{
    Int grados;
    Bool soleado;
}                       //Error de sintaxis: falta el ; al final
calculaKelvin(grados) -> Int{     //Error de sintaxis: falta especificar el tipo del
    parametro grados mediante :Int
    Int k = grados + 273;
    //Error de sintaxis: falta la sentencia return k; para devolver un valor de retorno
}
main() -> Int{

    Int var1 = 5 + 4) * 3;  //Error de sintaxis: falta (
```

```
13      Bool;           //Error de sintaxis: hay que asignar un identificador a la variable
14      Int x = 2 //Error de sintaxis: falta el ;
15      if(){           //Error de sintaxis: falta especificar la condicion del if
16          var1 = var1 + 1;
17      }
18      for(Int i = 0; i < 5, i=i+1){           //Error de sintaxis: en el for se pone ; en vez
        de ,
19          var1 = var + 1;
20      }
21      valuefor(var1){
22          | False = 5;
23          | 4 = 2; //La primera expresion debe ser de tipo booleano.
24      }
25      Int x = 0;
26      valuefor(var1){ //falla porque no hemos puesto el default y no se cumple ni la
        condicion x > 0 ni la x < 0.
27          | (x > 0) = 1;
28          | (x < 0) = -1;
29      }
30      Int y;
31      switch x{ //Error de sintaxis: faltan los parentesis.
32          | 0: y = 1;
33          break;
34          | default: y = 2;
35          break;
36      }
37      while(){} //Error de sintaxis: falta condicion del while.
38
39      return 0;
40  }
```

Listing 25: Fichero que contiene ejemplos de errores sintácticos de HaskPlus.

```
1  //Errores Tipado
2  funcProbando(v1:Int, v2:Bool) -> Int{
3      //codigo de la funcion
4      return 0;
5  }
6  Struct tInfo{
7      Int v1;
8      Bool v2;
9  }
10 main() -> Int{
11     Int var1 = 3;
12     Bool varB1 = False;
13     Int var2 = var1 + varB1;    //Error de tipos: varB1 no es de tipo Int
14     Bool varB2 = var1 & varB1;  //Error de tipos: var1 no es de tipo Bool
15     Int var1 = funcProbando(var1, var2)     //Error de tipos: var2 no es una variable de
       tipo Bool
16     tInfo variable;
```

```
17      variable.v1 = varB1;            //Error de tipos: varB1 no es de tipo Int
18      variable.noExiste = True; //Error de tipos: noExiste no es un campo del struct tInfo
19      if(var1){
20          //Error de tipos: en la condicion no hay una expresi n booleana
21      }
22      Int array[5] = [1..5];
23      array[3] = varB1;       //Error de tipos: varB1 no es de tipo Int
24      Int var3 = sum var1;    //Error de tipos: var1 necesita ser un array para poder
        aplicarle el operador sum
25      Int x;
26      valuefor(x){
27          | True = True; //Error de tipos: la x es de tipo Int.
28          | default = 0;
29      }
30      return 0;
31  }
```

Listing 26: Fichero que contiene ejemplos de errores de tipado de HaskPlus.