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.

Cambios con respecto a la entrega anterior: Con repecto a la última entrega, es importante señalar que hemos realizado ciertas modificaciones en las restricciones de nuestro lenguaje:

- Hemos ajustado las estructuras de control switch para que puedan operar tanto con valores booleanos como enteros. Además, cada caso dentro de un switch será tratado como un bloque independiente, permitiendo así la declaración de nuevas variables dentro de cada *case*. Las variables declaradas solo podrán usarse en el propio *case*.
- Hemos introducido la funcionalidad de importación (import) para facilitar la incorporación de archivos externos en nuestro programa.
- Las constantes ahora solo podrán ser de tipo entero.
- Hemos quitado la inicialización de arrays como Int array[i] = [1..i] y el operador concatenación. También hemos cambiado la sintaxis de declaración de arrays para que sea de una forma más intuitiva.
- Para evitar problemas con el tabulador y las guardas hemos cambiado la sintaxis del switch y del valuefor.
- Permitimos alias de cualquier tipo.

2. Estructura básica de un programa en HaskPlus

Como nos hemos basado en una mezcla entre *C++* y *Haskell*, la sintaxis será muy intuitiva. Nos podremos encontrar declaraciones de Struct, typedef , contantes, variables globales, import y definiciones de funciones, en cualquier orden. Como inicio de la ejecución del programa es importante destacar el *main*(), que será de la siguiente forma:

```
1 main() -> Int {
2     // codigo
3     return 0;
4 }
```

Cabe destacar que todos los programas deben contener una función *main*(). 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.

Observación 3.2. Cuando imprimimos un valor Bool mediante la función show() (que veremos más adelante), los booleanos se representan como 1 si es $True$ y 0 si es $False$. (Cualquier entero distinto de 1 se interpreta como $False$ en booleano).

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:

```
1 Struct nReg {
2     tipo1 var1;
3     tipo2 var2;
4     //lista de variables
5 };
```

Listing 1: nReg denota el nombre del nuevo tipo Struct definido

Para inicializar un Struct debemos ir inicializando componente a componente.

3.2.2. Tipo array

En cuanto a los arrays, en *HaskPlus* se usa la sintaxis List<tipo>[$TamanoArray$] $nombreArray$, donde primero se declara el identificador List, luego el tipo que tendrá el array entre < y >, y después el tamaño del mismo entre corchetes, que debe ser un número entero, seguido del nombre del array.
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 List<List<tipo>[$TamanoArray1$] >[$TamanoArray2$] $nombreArray$.
Los arrays pueden ser inicializados mediante cualquiera de los bucles que se presentarán en el apartado 5.5. Esto permite asignar un valor al array posición por posición. Dependiendo de las dimensiones del array, la sintaxis será la siguiente:

- dimensión 1: $nombreArray[i1] = valor$.
- dimensión 2: $nombreArray2[i1][i2] = valor$.
- dimensión n: $nombreArrayn[i1][i2][i3]...[in] = valor$.

A su vez, también pueden ser inicializados al declararlos. Por ejemplo, la sintaxis a seguir para el caso de 1 y 2 dimensiones sería:

- List<tipo>[5] $nombreArray$ = {1,2,3,4,5}
- List<List<tipo>[2] >[2] $nombreArray$ = {{1,2},{2,3}}.

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 &$nombrePuntero$.
Además, incluimos la instrucción *new* para reservar memoria dinámica. Un ejemplo de sintaxis sería:
tipo * $nombrePuntero$ = new tipo . Cabe destacar que a pesar de tener la instrucción *new*, no crearemos la instrucción *delete*, ya que no vamos a implementar un recolector de basura. Además, se permiten punteros a cualquier tipo.

4. Expresiones, operadores y bloques

4.1. Expresiones

En *HaskPlus*, las expresiones pueden ser:

- **Constantes de tipo**:
  - Int: número entero.

  Recordemos que las constantes no se pueden modificar.

```
1 const Int nEntero = valor1;       //declaracion de constantes para el tipo Int
```

- **Accesos a**:

- Variables: mediante el nombre o identificador de la variable, que estará formado por caracteres alfanuméricos. 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:

```
List<tipo>[tamanoArray] nArray;  //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 = new tipo; //declaramos el puntero reservando memoria
*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, es decir, unos dentro de otros. Por ejemplo, puede haber un Struct donde uno de sus campos sea un array, un array de Struct, un array de punteros, un puntero a un array, un Struct en el que uno de sus campos sea un puntero u otro Struct, un puntero a 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

- 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	Sí	6

- Operadores en Arrays:

Operador	Identificador	Tipo	Asociatividad	Prioridad
Sumatorio	sum	Unario prefijo	Sí	7
Productorio	prod	Unario prefijo	Sí	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;

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 el operador suma o resta, previamente definidos, 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, ya sea *i = i+1*, *i = i-1* o *i += 1* y *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 hacen falta más casos, se podrán hacer ifs anidados.

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 $\oplus = /, mod$.

- **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 las operaciones sumatorio y productorio sobre listas de tipo Int.

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
}
```

La *declaración* de la variable contador del bucle deberá ser de tipo Int, al igual que la *asignación*. La *expresión* evaluada será un Bool.

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, siendo n una variable de tipo Into un número entero.

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

5.6. Switch

HaskPlus implementa los *Switch-case* de una forma ligeramente distinta a la que acostumbra $C++$. En nuestro switch, cada caso será un bloque distinto, por lo que permitimos declaración de nuevas variables en cada caso del switch, que sólo podrán ser usadas en el mismo bloque en el que han sido declaradas. También implementaremos el caso por defecto así como el *break*. Sin embargo, no todos los switches deben incluir el caso $default$. Tanto en cualquier *case* como en el caso $default$ hay que poner *break* al final de cada caso.

```
switch(exp){
    case 1: //codigo
        break;
    ...
    case n: //codigo
        break;
    default: //codigo
        break;
}
```

La expresión que estudia el switch (*exp*) siempre deberá ser de tipo Int o Bool. Por otro lado, las sentencias *case* estarán acompañadas siempre de sus correspondientes Int (o Bool) que numerarán los casos. Además, los casos siempre tendrán que estar ordenados por orden ascendente. Mediante la implementacion del *brtable* conseguimos que el switch se haga en tiempo constante.

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 (opcional) y el *break*, que será necesario incluirlo al final de todos los casos (incluido el $default$).

```
valuefor(var) {
    case e1 = a1;
        break;
    ...
    case en = an;
        break;
    default = a;
        break;
}
```

La variable (*var*) a evaluar en el $valuefor$ debe ser de tipo Int o Bool. Esta variable y las expresiones a_i con las que se le asocia deben compartir el mismo tipo.

5.8. Typedef

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

typedef nombre = tipo ;

Cabe destacar que se acepta anidamiento de typedef , es decir, la definición de un typedef a partir de otro typedef ya definido.

5.9. Entrada y Salida

Considerando la fusión de nuestros respectivos lenguajes, hemos establecido que la impresión en pantalla se realice mediante la función show, mientras que la lectura se efectúe mediante read. Se admitirán tanto valores

Int como Bool para ambas funciones. Recordemos que en el caso de los valores booleanos, se representarán como 0 si son falsos y como 1 si son verdaderos.
Para mostrar por pantalla expresiones, se utilizará la sintaxis show(*expresion*). Por otro parte, la sintaxis de la lectura será: tipo *nombrevar* = read().

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

Listing 2: Impresión por pantalla

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

Listing 3: Lectura

Observación 5.3. Para verificar la funcionalidad del método read(), hemos creado un archivo (*input.txt*) con valores predefinidos para facilitar la verificación de ejemplos, tal y como fue sugerido por el profesor en clase.

5.10. Imports

Hemos incorporado la instrucción import para facilitar la incorporación de archivos externos en nuestro programa. La sintaxis de esta instrucción es relativamente simple y se compone de la palabra clave import, seguida de la ruta de acceso donde se ubica el archivo que deseamos importar dentro de nuestra práctica. Es importante mencionar que el nombre del archivo debe incluir su extensión correspondiente para que el sistema pueda ubicarlo correctamente.

```
import examples/media.txt;
```

Listing 4: Imports

Es crucial resaltar que, al importar un archivo, este no debe contener una función principal *main*(). La razón detrás de esta especificación es mantener la modularidad y la claridad en el diseño del programa. Al excluir una función *main*() en los archivos importados, evitamos conflictos y confusiones con la función principal del programa principal.

Observación 5.4. Nuestro lenguaje no permite la declaración de variables globales, constantes, typedefs, structs o funciones con el mismo nombre ni en el propio archivo ni en el archivo que se está importando. Esta restriccion garantiza la consistencia y evita conflictos entre los diferentes componentes del programa.

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 5: Paso por referencia

6.1. Llamadas a función

Para las llamadas a las funciones aplicaremos la sintaxis ya existente de *C++*. Si por ejemplo queremos guardar el valor de retorno de la función haremos:

tipo nombre = funcion (a,b);

Es importante destacar que la llamada a función no necesariamente debe asignarse siempre a una variable. Puede ocurrir en cualquier lugar, ya sea como un término dentro de cualquier expresión booleana o entera, así como en cualquier línea de código si la función es de tipo Void o presenta parámetros por referencia.

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. Cabe destacar que solo debe haber un *return* y debe estar ubicado en el ámbito 0 del cuerpo de la función, es decir, no puede estar dentro de condicionales, switches, bucles...

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

Es obligatorio incluir al final del cuerpo de todas las funciones la expresión *return*, acompañada del valor que se vaya a devolver, y seguido de un punto y coma, pero con una excepción. En el caso de las funciones de tipo Void, se debe incluir tan solo la instruccion *return* seguido de punto y coma, ya que no debe devolver ningún valor.

Es importante resaltar que se permite que se devuelvan expresiones de cualquier tipo, lo que significa que no estamos limitados a tipos básicos como Int o Bool, sino que también permitimos devolver listas, structs,

punteros, etc. Si lo que se devuelve no es de tipo básico, la expresión a retornar deberá ser guardada en una variable previamente.

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. Gestión de errores

Vamos a explicar como hemos realizado la gestión de errores sintácticos y léxicos.

- **Errores sintácticos**: A grandes rasgos, cuando se produce un error sintáctico, imprimimos un mensaje donde se especifica cuál es el causante de ese error, así como la fila y columna donde se ha producido el mismo. Se produce la recuperación del error tras el primer punto y coma que se encuentre. Los principales errores sintácticos que hemos trabajado han sido:
  - Declaración de variables: se produce un error sintáctico cuando falta el identificador de una variable o hay un error en su inicialización.
  - Declaración de funciones: se produce un error sintáctico cuando falta el símbolo -> en la declaración de una función, no se especifica el tipo de retorno, o hay errores en los parámetros de la función, ya sea por falta de : o del tipo de parámetro.
  - Instrucciones: se produce un error sintáctico si una instrucción no es reconocida.
  - Read y show: se produce un error sintáctico en las funciones *read* y *show* en caso de tener parámetros incorrectos.
    - En el caso de la función *read*, recordemos que no debe tener ningún parámetro.
  - Condiciones: se detecta error sintáctico en los *if*, *while*, *for*, *repeat*, *switch* y *valuefor* en caso de que haya un error en las condiciones de los mismos.
    - En el caso del *for*, también se detecta cualquier error ya sea al declarar o aumentar el contador del bucle.

  También se detectará cualquier falta de punto y coma en cualquier punto del código del programa. Cabe destacar que la falta de llaves será un error del que no nos recuperaremos y el prgrama fallará. Además, en el caso de que nos recuperemos de un error, no se creará correctamente la instrucción asociada a ese error.
- **Errores léxicos**: Cualquier elemento no declarado léxicamente se detecta como un error, mostrando la fila y columna donde se ha producido.

8. Vinculación

La vinculación, también conocida como binding, es un concepto fundamental en la programación que se refiere a la asociación entre un identificador y el objeto que representa o designa. Aunque el proceso de vinculación comienza en un punto específico del texto del programa, es durante la ejecución cuando los identificadores se vinculan realmente a objetos concretos.

En nuestro contexto, este proceso se lleva a cabo mediante una lista de HashMap, donde cada posición de la lista representa un nivel de ámbito (comenzando desde el nivel 0). Cada posición de esta lista contiene un HashMap donde cada clave es un identificador y cada valor es el ASTNode asociado a ese identificador. Esta estructura nos permite gestionar el ámbito de las variables y garantizar la coherencia en su uso dentro del programa.

La vinculación desempeña un papel crucial en varios aspectos de la programación. Por ejemplo, nos aseguramos de que no se declaren variables con el mismo identificador en el mismo ámbito, lo que evitaría confusiones y errores de interpretación. Además, al acceder a cualquier variable, podemos estar seguros de que ha sido previamente definida en el mismo ámbito. Del mismo modo, los valores de retorno deben estar vinculados a las funciones correspondientes, y las llamadas a funciones deben estar asociadas a funciones que hayan sido definidas previamente.

Para lograr esto, recorremos la lista de nodos del árbol de sintaxis abstracta (AST) que hemos construido durante el análisis del programa. Cada nodo del árbol tiene su correspondiente a *bindNode*, que contiene la información necesaria para establecer estas asociaciones de manera coherente y precisa.

En caso de que ocurra un fallo en el proceso de vinculación, como intentar acceder a una variable no declarada previamente en el mismo ámbito, se lanzará una excepción de tipo BindingException. Esta excepción proporcionará un mensaje explicativo que se imprimirá en la consola, ayudando así a identificar y corregir rápidamente el problema.

9. Tipado

En esta parte nos encargamos del correcto uso de los tipos. En *HaskPlus* se lleva a cabo un tipado que presenta las siguientes características:

- *Declaración de tipos explícita*: se debe indicar explícitamente el tipo de cada variable.
- *Tipado fuerte*: se asegura de que a cada objeto solo se le puedan aplicar exclusivamente las operaciones relacionadas con su tipo. En caso contrario, se lanzará una excepción de tipo TypingException con un mensaje explicativo acerca de la razón del fallo.
- *Equivalencia de tipos por nombre*: Dos tipos son iguales si tienen el mismo nombre.

Al igual que con el proceso de vinculación, en el análisis de tipos recorremos los diversos nodos del árbol de sintaxis abstracta (AST). Durante este proceso, registramos el tipo de cada nodo para poder realizar comparaciones posteriormente.

En el análisis de tipos, nos aseguramos de varios aspectos cruciales, como que las asignaciones tengan el mismo tipo tanto en el lado izquierdo como en el derecho, además de que las condiciones de los bucles y las estructuras condicionales sean del tipo adecuado. A su vez, en las funciones, los tipos de los argumentos deben coincidir con los tipos de los parámetros, y que el tipo de la expresión asociada a la instrucción return sea compatible con el tipo de retorno declarado por la función.

Además, en el proceso de tipado, también verificamos que al acceder a un Struct, el campo al que intentamos acceder realmente existe. Esta verificación se realiza durante el análisis de tipos y no durante la vinculación,

como se explicó en clase. Hablando de accesos, también nos aseguraremos de que al acceder a la posición de un array, el índice que representa la posición y que va entre corchetes sea de tipo Int.

Es importante destacar que esta sección del proceso se alcanza únicamente si no se han detectado fallos en el proceso de vinculación. Esto subraya la importancia de un análisis previo riguroso para garantizar la coherencia y la integridad del programa durante la etapa de tipado.

10. Generación de código

Una vez que hemos completado las etapas anteriores del proceso de compilación sin errores, procedemos a la generación de código. Sin embargo, antes de comenzar esta fase, realizamos algunas tareas de preparación importantes.

En primer lugar, asociamos a cada identificador (designador) un número que indica su posición relativa al inicio de un bloque de código (delta). Esto es fundamental para la gestión de la memoria y la asignación de direcciones en tiempo de ejecución. La primera variable dentro de un bloque tendrá la posición 0, y las siguientes tendrán la posición de la anterior más el tamaño de la anterior. Para facilitar este proceso, necesitamos una función llamada $getSize()$, que nos devuelve el tamaño del tipo en concreto.

Para los structs y las funciones, inicializamos la cuenta de los deltas a 0. Al abrir otro tipo de bloques como el de lo bucles, los condicionales, los switches o los valuefor, no reiniciaremos el delta. Cada vez que cerramos un bloque, descontamos al delta el tamaño de todas sus variables.

También contamos con la función 'calculateAdress()', que nos ayuda a determinar la posición en memoria de un objeto específico, lo cual es crucial durante la generación de código.

Además, debemos calcular el tamaño máximo de la memoria estática necesaria para nuestro programa. Para ello, utilizamos la función '$getMaxMemoryGlobal()$'.

Al iniciar la generación de código, primero nos enfocamos en generar el código para variables globales y constantes mediante la función $preMaingenerateCode()$. Posteriormente, nos ocupamos de generar el código para el resto del árbol.

11. Ejemplos

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

- **01_dec_var.txt**: Declaración y asignación de valores a variables de tipos Int y Bool.
- **02_op_arit.txt**: Ejemplos de expresiones con operadores aritméticos.
- **03_op_bool.txt**: Ejemplos de expresiones con operadores booleanos.
- **04_op_array.txt**: Ejemplos de expresiones con operadores de arrays.
- **05_bloques.txt**: Ejemplo de bloques anidados y del cambio de ámbito del programa en cada uno de ellos.
- **06_punteros.txt**: Punteros.

- **07_arrays.txt**: Arrays.
- **08_structs.txt**: Structs.
- **09_encuentraMax.txt**: Consiste en encontrar el máximo en un *array* de enteros.
- **10_factorial.txt**: Función que calcula el factorial de un número dado.
- **11_media.txt**: Cálculo de la media de un array.
- **12_fibonacci.txt**: Función que calcula el *nesimo* número de la serie de Fibonacci.
- **13_if.txt**: Ejemplos del uso del $if - else$ con una o dos ramas.
- **14_while.txt**: Ejemplos de *while* con condiciones booleanas y comparaciones de enteros.
- **15_for.txt**: Ejemplo del típico uso del for como incrementador.
- **16_repeat.txt**: Ejemplo de *repeat* para dar un número n de vueltas.
- **17_switch.txt**: Ejemplo del *switch* donde la variable a diferenciar casos es un Int o Bool.
- **18_valuefor.txt**: Asignamos valor a una variable de tipo Int usando el $valuefor$.
- **19_typedef.txt**: Ejemplo de alias y operación con ellos. Se muestra typedef de $typedef$, y $typedef$ a tipos no básicos.
- **20_read.txt**: Lectura y escritura de Int y Bool.
- **21_valor_referencia.txt**: Diferencias entre el paso por valor o por referencia de los parámetros de una función.
- **22_errores_sintax.txt**: Fichero que contiene ejemplos de errores sintácticos de HaskPlus.
- **23_errores_tipado.txt**: Fichero que contiene ejemplos de errores de tipado de HaskPlus.
- **24_constantes.txt**: Fichero que contiene declaraciones de contantes.
- **25_producto_matrices.txt**: Fichero que contiene productos de matrices. Es importante destacarlo porque muestra el paso por referencia de matrices, y un ejemplo de retorno de una lista de listas.
- **26_imports.txt**: Ejemplo de programa con una instrucción import. Calcula los 5 primeros números de la serie de Fibonacci y luego hace la media de ellos. Hace uso de los archivos *media.txt* y $fibonacci.txt$.
- **27_producto_matrices_read.txt**: Ejemplo igual que el archivo $25_producto_matrices.txt$ pero que lee por teclado las matrices.
- **media.txt**: Ejemplo de fichero a importar. Contiene una función que calcula la media de un array. Tiene el mismo contenido que $11_media.txt$ pero sin *main* para poder importarse.
- **fibonacci.txt**: Ejemplo de fichero a importar. Contiene una función que calcula el n-ésimo término de la serie Fibonacci. Tiene el mismo contenido que $12_fibonacci.txt$ pero sin *main* para poder importarse.