Especificacion del Lenguaje *BLA*

Fernández, K. & Gyomrey, A.

El presente documento muestra la especificación del lenguaje *BLA*. *BLA* es un lenguaje imperativo fuertemente tipado. Su sintáxis y operaciones tienen elementos inspirados en C y otros lenguajes tales como Haskell, Python y R.

Consideraciones Léxicas

Las siguientes palabras son reservadas del lenguaje:

> `alias` `type` `union` `as`, `enum`, `fun`, `return`, `none`, `int`, `float`, `boolean`, `char`, `string`, `for`, `if`, `elif`, `else`, `while`, `in`, `continue`, `break`, `const`, `read`, `print`, `true`, `false`, `fill`, `map`, `foldr`, `foldr`

Estas palabras corresponden con las funciones nativas de *BLA* explicadas más adelante.

Un idenficador es una secuencia de letras y dígitos, empezando por una letra. *BLA* es sensitivo a capitalización, es decir, el uso de de mayúsculas y minúsculas hace variar un identificador. Por ejemplo, `foo` y `Foo` son identificadores distintos.

En *BLA* es posible escribir comentarios en bloque y de línea basados en la sintáxis de Haskell. Un comentario de línea empieza con dos guiones consecutivos `--` y finaliza con un fin de línea. Los comentarios de bloque empiezan con `{-` y terminan con `-}`. Cualquier símbolo es permitido en un comentario exceptuando la secuencia `-}`, la cual tiene la función única de finalizar el comentario actual.

Las constantes booleanas son `true` y `false` correspondientes respectivamente a los valores de verdadero y falso. Los booleanos se representan en un byte.

Un entero puede ser especificado con o sin su base. Un entero con base es una secuencia de dígitos decimales (0–9) y caracteres (a-zA-Z), seguido de un guión bajo (_) y de la base especificada en formato decimal. Los carácteres son válidos siempre que la base lo requiera. La base es un valor entre 1 y 36, ambos inclusive. Los enteros son de 32 bits.

En caso de no indicar la base, se considera que el entero está expresado en base decimal. Los siguientes son ejemplos válidos de enteros:

> 10_2, 124_7, 2A_16, 2br_20, 00000_1

Un número flotante es una secuencia de dígitos, un punto, seguido de cualquier secuencia de dígitos. La segunda secuencia puede estar vacía. Adicionalmente, se pueden escribir en notación científica. El signo del exponente es opcional, en caso de no especificarse se asume positivo y `E` puede estar expresado en mayúsculas o minúsculas. Los flotantes son de precisión de 32 bits.

Son ejemplo de flotantes válidos:

4.67, 0.42, 3., 37.0, 23.4E+2, 42.8e–6

Una constante `string` es una secuencia de caracteres delimitados por comillas dobles. Los strings pueden contener cualquier caracter. Una cadena de caracteres puede empezar y terminar en líneas diferentes. Dos cadenas de caracteres contínuas se concatenan automáticamente. Es decir, el siguiente es un `string` válido.

```
"Este es un string válido. Nótese que las comillas que lo
delimitan estan
en líneas diferentes." "Este string es continuación del anterior."
"Y este también."
```

Los operadores y caracteres de puntuación usados por el lenguaje son los siguientes:

```
+ - * / % < <= > >= = == != && || ! ==> <== ; , . [ ] ( ) { } : @ ^
```

Gramática de *BLA*

Input ::= Declaration | Input Declaration

TypeDef ::= `ident`

TypeDef ::= `^` TypeDef

TypeDef ::= `[` Expression `]` TypeDef

TypeDef ::= `[` Expression `:` Expression `]` TypeDef

TypeDef ::= `^` TypeDef

Declaration ::= VariableDeclaration

Declaration ::= `fun` Type `ident` (ArgList) Block

Declaration ::= `fun none ident` (ArgList) Block

Declaration ::= `alias` TypeDef `ident ;`

Declaration ::= `type ident {` StructFields `}`

Declaration ::= `enum ident {` EnumValues `}`

VariableDeclaration ::= `const` Type `ident` = Expression `;`

VariableDeclaration ::= Type `ident` = Expression `;`

VariableDeclaration ::= Type `ident ;`

StructFields ::= lambda | StructFields TypeDef `ident ;`

EnumValues ::= `ident`

EnumValues ::= `enumConstant`

EnumValues ::= EnumValues `, ident`

EnumValues ::= EnumValues `, enumConstant`

ArgList ::= lambda | NonEmptyArgList

NonEmptyArgList ::= `const` Type `ident` | Type `ident`

NonEmptyArgList ::= NonEmptyArgList `,` Type `ident`

NonEmptyArgList ::= NonEmptyArgList `, const` Type `ident`

Block ::= `{` BlockStatement `}` | `{ }`

BlockStatement ::= Statement | BlockStatement Statement

Statement ::= VariableDeclaration | `ident =` Expression `;` | `;`

Statement ::= `if (` Expression `)` Block ElseStatements

Statement ::= `while (` Expression `)` LoopBlock

Statement ::= `for ident int (` Expression `,` Expression `)` LoopBlock

Statement ::= Block

Statement ::= `return ;` | `return` Expression `;`

Statement ::= `ident (` ParametersInstance `)`

Statement ::= `read ident ;` | `print` ListIdent `;`

ListaIdent ::= Expression | ListIdent `,` Expression

LoopBlock ::= `{` LoopBlockStatement `}` | `{ }`

LoopBlockStatement ::= LoopStatement | LoopBlockStatement LoopStatement

LoopStatement ::= Statement | `break ;` | `continue ;`

ElseStatements ::= lambda | `elif (` Expression `)` Block ElseStatements

ElseStatements ::= `else` Block

Expression ::= `intConstant` | `floatConstant` | `charConstant` | `booleanConstant` | `stringConstant` | `enumConstant`

Expression ::= `(` Expression `)`

Expression ::= Lvalue

Expression ::= `ident` | `ident (` ParametersInstance `)`

Expression ::= Expression `as` TypeDef

Expression ::= AritmeticExpression | BooleanExpression

AritmeticExpression ::= `-` Expression

AritmeticExpression ::= Expression `+` Expression

AritmeticExpression ::= Expression `-` Expression

AritmeticExpression ::= Expression `*` Expression

AritmeticExpression ::= Expression `/` Expression

AritmeticExpression ::= Expression `%` Expression

BooleanExpression ::= Expression `&&` Expression

BooleanExpression ::= Expression `||` Expression

BooleanExpression ::= Expression `==>` Expression

BooleanExpression ::= Expression `<==` Expression

BooleanExpression ::= `!` Expression

BooleanExpression ::= Expression `==` Expression

BooleanExpression ::= Expression `!=` Expression

BooleanExpression ::= Expression `<` Expression

BooleanExpression ::= Expression `<=` Expression

BooleanExpression ::= Expression `>` Expression

BooleanExpression ::= Expression `>=` Expression

String ::= `string` | String `string`

Lvalue ::= `ident`

Lvalue ::= Lvalue `[` Expression `]`

Lvalue ::= Lvalue `.` `ident`

Lvalue ::= Lvalue `^`

Lvalue ::= `@` Lvalue

ParametersInstance ::= lambda | ParametersInstanceNonEmpty

ParametersInstanceNonEmpty ::= Expression

ParametersInstanceNonEmpty ::= ParametersInstanceNonEmpty `,` Expression

Estructura de un Programa

Un programa en *BLA* consiste en una secuencia de declaraciones, donde cada declaración establece una variable, función o un nuevo tipo. En el caso de las funciones, las declaraciones deben incluir el cuerpo de éstas. Para las variables, puede o no incluir su inicialización.

Todo programa debe tener una función denominada `main` con retorno `none` y que no tome parámetros. Esta función sirve de punto de entrada a la ejecución del programa.

Alcance

BLA soporta niveles anidados de alcance. El alcance es estático. Una declaración en el nivel más alto tiene alcance global. Cada función tiene su propio alcance local para sus parámetros y otras variables locales que pueden ser declaradas en el cuerpo de estas. Un par de llaves de la forma `{ ... }` define un nuevo alcance anidado dentro de un alcance local. Los alcances locales opacan u ocultan los alcances más externos, es decir, una variable `a` definida en una función oculta cualquier otra variable con el mismo nombre definida en un alcance externo.

- Todos los identificadores deben ser declarados.
- Una variable es visible posterior a su declaración. Es decir, no puede ser utilizada como *lvalue* ni *rvalue* previamente a su declaración.
- Las funciones son visibles desde cualquier punto del programa, incluso previo a su declaración.
- Los identificadores para variables dentro de un mismo alcance deben ser únicos. Igualmente, una variable no puede tener el mismo nombre que una función.
- Se considera que los parámetros formales de una función se encuentran en el primer nivel de alcance local, por lo que no es posible crear variables locales con el mismo nombre de algún argumento en el primer nivel de alcance. Es posible hacerlo en alcances anidados.
- Las funciones se encuentran identificadas unívocamente por su nombre.
- Las declaraciones en alcance global son accesibles a lo largo de todo el programa para funciones y posterior a su declaración para variables, a menos que sean ocultadas por variables del mismo nombre en alcances internos.
- Las declaraciones en alcances cerrados no son accesibles.

Tipos

Los tipos nativos del lenguaje son `int`, `float`, `boolean`, `string` y `none`.

Los arreglos pueden ser multi-dimensionales y de cualquier tipo distinto a `none`.

BLA permite crear nuevos tipos de datos. Y generar referencias a ellos.

Variables

Las variables pueden ser declaradas de cualquier tipo no vacío, es decir, de cualquier tipo distinto a `none`. Esto incluye arreglos y tipos definidos por el usuario. Las variables declaradas fuera de alcances locales tienen por defecto alcance global.

Las variables declaradas en la lista de parámetros formales o cuerpo de la función tienen alcance local.

Las variables globales pueden ser inicializadas con expresiones complejas siempre que éstas no incluyan llamadas a funciones definidas por el usuario. La siguientes son inicializaciones válidas:

```
int a; int a = 3; int a = 3 * (4-5 * 3);
```

Mientras que las siguientes son inicializaciones inválidas, considerando `foo` una función definida por el usuario que recibe un entero y retorna un entero.

```
int a = foo(1); int a = 3 + 4*foo(5);
```

Las variables son inicializadas por defecto con el valor 0 para los tipos `int` y `float`, con el caracter nulo para caracteres, `false` para booleanos. Se cumple la regla recursivamente para los tipos compuestos definidos por el usuario. Análogamente para los elementos de un arreglo. Para el caso de los apuntadores se inicializan automáticamente con el apuntador nulo.

Si una variable está declarada con inicialización constante no puede ser pasada por referencia ni se puede obtener la dirección de esta. Dado que las estructuras, arreglos y uniones se pasan por referencia, no es posible pasar un objeto de este tipo a una función si éste fue declarado como constante.

Arreglos

En *BLA* los arreglos son homogéneos, indexados linealmente y de tamaño constante.

- Los arreglos pueden ser de cualquier tipo primitivo distinto a `none` o tipo declarado por el usuario.
- Pueden establecerse arreglos de varias dimensiones.
- Los arreglos son inicializados por defecto con el valor 0 para los tipos `int` y `float`, y `false` para los `boolean`. Los arreglos de registros son inicializados rellenando con ceros los bytes que esta contenga.
- El número de elementos en el arreglo no puede cambiarse una vez declarado.
- Los arreglos son indexados desde cero hasta una posición antes de su longitud, a menos que se especifique lo contrario. Es decir, si el rango de un arreglo va de `a` a `b`, se puede acceder a cualquier posición mayor o igual a `a` y estrictamente inferior a `b`.
- El índice utilizado en la selección debe ser de tipo entero.
- Ocurre un error a tiempo de ejecución si se intenta acceder a una posición fuera de los límites de un arreglo.
- Los arreglos pueden ser pasados a las funciones por referencia.
- La asignación de arreglos es profunda, es decir, se copia elemento a elemento. Ambos arreglos deben tener el mismo tipo, tamaño y límites.
- La declaración de arreglos se efectúa primero las dimensiones y seguidamente el tipo base, más detalles acerca de esto se definirán más adelante.

Strings

Se pueden incluir secuencias de caracteres constantes.

- La asignación de strings es superficial.
- Los strings pueden ser pasados como parámetros a las funciones.
- La comparación de strings con `==` y `!=` compara las secuencias caracter por caracter.

Funciones

La declaración de una función incluye el nombre de esta y su firma asociada, es decir, tipo de retorno y número y tipo de parámetros formales.

- Las funciones tienen alcance global, no se permiten funciones anidadas.
- Las funciones pueden tener cero o más parámetros.

- Los parámetros formales pueden ser de cualquier tipo distinto a `none`, arreglos o tipos definidos por el usuario.
- Los identificadores utilizados en los parámetros formales deben ser distintos.
- Los arreglos y tipos estructurados se pasan por referencia. Los tipos básicos se pasan por valor.
- Los parámetros formales se encuentran en alcance local, por lo que no es posible definir una variable con el mismo identificador que un parámetro formal en el primer nivel de anidamiento de una función. Sin embargo, en niveles de anidamiento mayor es posible la redefinición.
- Los tipos de retorno pueden ser únicamente tipos primitivos escalares del lenguaje. En caso de querer retornar un tipo compuesto, este debe ser pasado por referencia a la función.
- Si el tipo de retorno de una función es `none`, no es posible colocar en su cuerpo retornos con parámetros, es decir, sólo es posible colocar `return` vacíos.
- Se permiten funciones recursivas directa o indirectamente.
- El cuerpo de las funciones debe estar contenido en un bloque de anidamiento `{ ... }`. Dicho bloque puede a su vez contener un número arbitrario de instrucciones, así como nuevos niveles de anidamiento.

Invocación de funciones

La invocación de funciones incluye el pasaje de argumentos del llamador al llamado, la ejecución del cuerpo del llamado y el retorno del control al llamador, posiblemente con un resultado. Los tipos primitivos son pasados por defecto por valor mientras que los tipos estructurados son pasados por referencia.

- El número de argumentos pasados a una función en su invocación debe coincidir con el número de parámetros formales.
- El tipo de cada argumento real en la invocación de una función debe coincidir con el tipo de cada parámetro formal.
- Los parámetros son evaluados de izquierda a derecha.
- Una función retorna el control al llamador una vez alcanzado el final de su cuerpo o al alcanzar una instrucción `return`. Si el tipo de retorno es no vacío la función debe incluir un retorno por cada posible punto de salida.

Apuntadores

BLA permite el uso de apuntadores. Para esto se utilizan dos operadores `^` y `@` que devuelven respectivamente el valor resultante de desreferenciar un apuntador y la dirección de memoria de un elemento.

No es posible obtener la dirección de memoria de un elemento declarado como constante.

Tipos compuestos

BLA permite la creación de tipos compuestos, equivalentes a `struct` de C. Los tipos compuestos se declaran utilizando la palabra reservada `type`, seguida del nombre del tipo y de la lista de campos que este contiene. Los campos deben estar definidos entre llaves, separados por punto y coma. La siguiente es una declaración válida de un tipo:

```
type Circulo { int x; int y; float r; }
```

Los tipos compuestos se alínean por palabra y se realiza padding para alinear los campos internamente. Los tipos compuestos se pasan por referencia a las funciones.

Uniones

BLA permite la creación de uniones. Dichas uniones son seguras, por lo que se verifica a tiempo de ejecución que la unión se esté utilizando en un contexto adecuado. La declaración de las uniones es similar a la declaración de los tipos compuestos, exceptuando el uso de la palabra clave `union` en vez de `type`. La siguiente es una declaración válida de una unión:

```
union intYFloat { int x; float y; }
```

Enumeraciones

BLA soporta tipos enumerados. Se especifica un identificador y entre llaves el conjunto de identificadores que contiene. Estos valores automáticamente pasan a ser contantes del lenguaje y no pueden ser usados como valores en situaciones posteriores.

- Las constantes de enumeración son sensitivas a mayúsculas y minúsculas

- Las constantes hacen inaccesible cualquier variable declarada previamente con el mismo nombre.
- Las constantes deben ser únicas, es decir, no pueden haber dos constantes con el mismo identificador declaradas tanto en el mismo tipo enumeración como en enumeraciones distintas.

```
enum colores {amarillo, azul, rojo}
```

Lectura del Nombre de un Tipo

El nombre de un tipo se lee de izquierda a derecha.

- El caso más sencillo se trata de los tipos primitivos, estructuras y uniones, que vienen definidos por su identificador. Por ejemplo: `int`, `float`, `char`, `string`.
- Para los apuntadores, se utiliza la declaración `^T` y se lee "un apuntador al tipo T".
- Para el caso de los arreglos, se escribe primero las dimensiones y seguidamente el tipo base, por ejemplo `[E]T` es un arreglo de `E` posiciones de `T`. `[E1:E2]T` es un arreglo cuyos índices se encuentran en el intervalo `[E1,E2)` de tipo `T`.
- Todos los otros tipos surgen de aplicar recursivamente estas reglas.

Los siguientes son ejemplos de declaraciones de tipo y su significado:

- `^int` : un apuntador a entero.
- `^[3]int` : un apuntador a arreglo de tamaño 3 de enteros.
- `[3]^int` : un arreglo de tamaño 3 de apuntadores a enteros.
- `[3][4]Circulo` : un arreglo de tamaño 3 de arreglos de tamaños 4 de `Circulo`.
- `[42]^[69]float` : un arreglo de tamaño 42 de apuntadores a arreglos de tamaño 69 de `float`.

Aliases

Un tipo puede ser utilizado en varios lugares usando un alias si necesidad de escribir su estructura completa. Un ejemplo de definición es:

alias [8][8]^pieza tablero;

Las siguientes dos declaraciones generan exactamente el siguiente efecto:

tablero t1;

[8][8]^pieza t1;

Equivalencia de Tipos, Compatibilidad

BLA es un lenguaje fuertemente tipado. La equivalencia de tipos se da por nombre. Por lo tanto, dos tipos definidos por el usuario son equivalentes si y sólo si tienen el mismo nombre, no serán equivalentes si tienen distintos nombres e iguales estructuras.

Todas las operaciones se deben ejecutar con argumentos del mismo tipo, en caso de necesitar realizar conversiones estas deben ser declaradas explícitamente, de la forma `A as T`, donde `T` es el tipo destino y `A` es el elemento original. Sólo se permiten conversiones entre tipos primitivos.

Asignación

BLA efectúa copia por valor tanto para los tipos básicos como para los compuestos. La instrucción `LValue = RValue` copia el valor del lado derecho en la dirección indicada en el lado izquierdo.

- La asignación de tipos no primitivos es profunda. Esto significa que las referencias también se copian.
- No es permitido utilizar constantes en el lado izquierdo de una asignación, exceptuanto en la asignación de inicialización.
- Ambos lados de la asignación deben tener el mismo tipo.
- Los parámetros formales de una función son considerados variables a menos que se especifique lo contrario con el uso de la palabra reservada `const` previo a su tipo.

Estructuras de Control

Las estructuras de control de *BLA* están basadas en las de *Python*.

- Una cláusula `else` o `elif` siempre se asocia al `if` más cercano aún sin cerrar.
- Las expresiones condicionales de las declaraciones `if`, `elif` y `while` deben ser de tipo `boolean`.
- La variable de iteración de un ciclo acotado `for` debe ser de un valor escalar discreto: `int`, `char`, `boolean`
- Una declaración de tipo `break` o `continue` sólo pueden aparecer en el cuerpo de ciclos `while` o `for`.
- La variable de iteración en un ciclo acotado `for` no puede ser utilizada como `Lvalue`, así como no puede ser pasada por referencia en funciones que se encuentren en el cuerpo del ciclo.
- El valor de una declaración `return` (en caso de ser funciones que retornen algún tipo no vacío) deben ser del mismo tipo especificado en la firma de la función.
- Las cláusulas `if`, `elif`, `else` deben ser precedidas por una instrucción de tipo bloque. Esta puede contener a su vez más instrucciones

Expresiones

Por simplicidad, *BLA* no permite la conversión automática de tipos. Esto significa que aún cuando la expresión equivalga entre los tipos, no es posible la interacción entre ellos. Para realizar esta operación se debe usar la conversión explícita.

- Las constantes evalúan a su valor interno (true, false, enteros, flotantes, cadenas, etc.).
- Ambos operandos en las operaciones aritméticas (+, -, *****, / y %) deben ser ambos enteros o flotantes, pero del mismo tipo.
- El operador unario `-` aplica sobre enteros y flotantes y devuelve el mismo tipo del operando.
- Ambos operandos en las comparaciones aritméticas (<, >, <= y >=) deben ser enteros, flotantes o caracteres, pero del mismo tipo. El resultado de la operación es `boolean`. En caso de caracteres, se evalúa la precedencia según su ocurrencia en la tabla ASCII.

- Ambos operandos deben ser del mismo tipo para las operaciones de igualdad (==, !=). Al comparar dos arreglos, se hará de forma profunda. El resultado de la operación será de tipo `boolean`.
- Los operandos para las expresiones booleanas (!, &&, ||, ==> y <==) aceptan sólo argumentos de tipo `boolean` y devuelven `boolean`.
- Las operaciones booleanas || y && se evalúan haciendo corto circuito.
- Para convertir explícitamente un flotante, caracter o booleano en un entero se utiliza la sintáxis `E as int`, donde E es la expresión a convertir. En caso de convertir un flotante a entero se realiza redondeo, en caso de conversión de caracter a entero se utiliza su valor en la tabla ASCII. Para convertir un entero, caracter o booleano a flotante se utiliza `E as float`. Para convertir de entero a booleano se utiliza `E as boolean`, el cual devolverá `false` en caso de ser cero y `true` en caso contrario.
- Es posible realizar conversiones explícitas para todos los tipos básicos, utilizando la forma `E as T`, donde `T` es el tipo destino y `E` la expresión a convertir.

Precedencia de Operadores

La precedencia de los operadores, de menor a mayor es:

- `== !=` (igualdad, asocia a izquierda)
- `<==` (consecuencia, asocia a izquierda)
- `==>` (implicación, asocia a derecha)
- `&& ||` (operadores lógicos y, o. Asocian a izquierda)
- `< <= > >=` (relacionales, no asocian)
- `- +` (suma, resta, asocian a izquierda)
- `* / %` (multiplicación, división entera o con decimales, módulo, asocial a izquierda)
- `- !` (negación unaria aritmética y booleana)
- `as` (conversión explícita, asocia a izquierda)
- `@` (dirección, asocia a derecha)
- `^` (desreferencia, asocia a izquierda)

La razón de esta precedencia es respetar la la precedencia natural de expresiones como `true && false == false`, considerando la igualdad como equivalencia, dicha expresión debe evaluar `(true && false) == false`, a diferencia de lenguajes como C que evalúa `true && (false == false)`.

Conversiones Explícitas

Todos los tipos básicos son convertibles a cualquier otro tipo básico. A continuación se muestran las convenciones de conversión:

- `int`: Si desea convertir una expresión de tipo `boolean` a `int` el resultado será cero en caso de `false` y uno en caso contrario. Para la conversión desde `char` el resultado será el índice correspondiente del caracter en la tabla ASCII. Al convertir de `float` a `int`, se truncará la mantisa, en caso de overflow se asignará el máximo entero o el mínimo entero según sea el caso.
- `float`: Las conversiones para `char` y `boolean` son análogas a los casos de conversión a entero. La conversión de `int` a `float` es la conversión real del valor, no la copia directa de bits.
- `char` : Puede asumir que la conversión a `char` se efectúa para los casos `boolean` y `float` como si realizara la conversión inicial a `int` y luego utilizar dicho valor como índice en la tabla ASCII. En caso de que el índice sea negativo o supere el rango permitido, se utilizarán los caracteres extremos en la tabla.
- `boolean` : Al igual que las conversiones a `char`, puede asumir que las conversiones a `boolean` son el equivalente a convertir inicialmente a `int`, luego, si el valor resultante es cero, el booleano será `false`, en caso contrario será `true`.

Las conversiones se realizan utilizando el operador binario `as`. La siguiente es una conversión válida:

```
3 as float
```

Verificaciones a Tiempo de Ejecución

Algunas verificaciones no pueden conocerse estáticamente y deben esperar hasta la ejecución para indicar la falla:

1. El acceso a índices fuera de la definición de un arreglo terminará la ejecución del programa.
2. El uso correcto de las uniones, es decir, el acceso a la parte activa de la unión.