java2cpp

Versión alpha 1.0

“De java a c++ en un solo paso”

Documentación

Trabajo final de compiladores

Universidad Católica "Nuestra Señora de la Asunción"

2021

Profesores: **Ernst Heinrich Goossen**
Luis Martínez

Code owners: **Lucas Martínez**
Joaquín Caballero

Tabla de contenido

Java2cpp

Java2cpp es un traductor dirigido por la sintaxis cuyo input es un texto en código de lenguaje Java y su output es la traducción de este código Java a código de lenguaje C++. Este traductor fue desarrollado utilizando las herramientas de LEX/FLEX y YACC/BISON.

Java2cpp cuenta con ciertas limitaciones por lo que se debe leer la sección de limitaciones (pág. 16).

Java y C++

Para elegir los lenguajes hemos considerado varios factores, como popularidad del lenguaje, similitudes entre ambos, complejidad y performance de estos.

Tanto Java como C++ son lenguajes bastantes populares y similares entre sí, si bien tienen sus diferencias, estas son ligeras variantes de sintaxis por lo que la dificultad de desarrollar el traductor no sería excesivamente mucha (al menos para la base del traductor). Dicho esto, cabe mencionar que encontramos algunas dificultades en el camino.

Se decidió hacer un traductor de Java a C++ y no viceversa ya que normalmente como C++ es un lenguaje de relativo bajo nivel, suele tener mejor performance que uno de más alto nivel como lo sería Java en este caso. Con esto en mente supusimos que sería más útil traducir de Java a C++.

Uso

Dependencias

Lex/Flex
Yacc/Bison
gcc (o similar)
libl.a
liby.a

Lex es un programa para generar analizadores léxicos (en inglés scanners o lexers). Lex se utiliza comúnmente con el programa yacc que se utiliza para generar análisis sintáctico.

El GNU Compiler Collection (colección de compiladores GNU) es un conjunto de compiladores y librerías creados por el proyecto GNU para C, C++, Objective-C, Fortran, Ada, Go , y D. Estos compiladores se consideran estándar para los sistemas operativos derivados de UNIX.

Libl.a es la biblioteca de Lex y liby.a es la biblioteca Yacc que contiene implementaciones de `yyerror` y funciones principales compatibles con Yacc.

Instalación de dependencias

Ubuntu:

Para instalar Bison ejecutar en la consola el comando:
`sudo apt-get install bison`

Para instalar Flex ejecutar en la consola el comando:
`sudo apt-get install flex`

Por defecto ubuntu ya cuenta con gcc instalado, para verificarlo ejecute `gcc --version`
En caso de que no lo tenga lo puede instalar ejecutando en la consola el comando:
`sudo apt-get install build-essential`

Ubuntu suele contar con esta librería.
Para instalar la librería libl.a (en caso de que sea necesario) ejecutar en la consola el comando:
`apt install libfl-dev`

Para instalar la librería liby.a ejecutar en la consola el comando:
`apt install libbison-dev`

Compilación

El archivo .zip ya cuenta con los archivos compilados por lo que usted ya puede ejecutarlo directamente si así lo desea. Este paso es solamente obligatorio si usted ha modificado algún archivo o cree usted que algún archivo se encuentra faltando o dañado.

Para su correcta compilación debe ejecutar lo siguiente en la consola o terminal de su computador:

```
lex java2cpp.l
yacc java2cpp.y
cc -o java2cpp y.tab.c -ly -ll
```

El primer comando (lex java2cpp.l) genera el código fuente C para el analizador léxico donde java2cpp.l es el archivo que contiene la especificación lex.

El segundo comando (yacc java2cpp.y) genera el código fuente C a partir del archivo java2cpp.y que contiene las especificaciones del traductor en gramáticas libres de contexto.

El tercer y último comando compila los archivos generados por los dos comandos anteriores utilizando el compilador gcc. La bandera -o NOMBRE significa que el archivo resultante tendra el nombre NOMBRE. El archivo y.tab.c es generado al ejecutar el segundo comando. La bandera -ly implica el uso de la librería liby.a y la bandera -ll el uso de la librería libl.a

Observación: si usted ha decidido utilizar un compilador diferente a gcc (como clang) aquí es donde debe compilar utilizando dicho compilador.

Ejecución

La ejecución es bastante sencilla, consiste en pasarle al traductor un archivo que contenga el código en lenguaje java (no es necesario que la terminación del archivo sea .java)
Para hacer esto, simplemente ejecutamos:

```
./java2cpp < FILE_PATH
```

Donde `java2cpp` es el archivo ejecutable compilado para el traductor java2cpp, y FILE_PATH es el path del archivo que contiene el código Java que queremos traducir.

Observación: usted debe cambiar FILE_PATH por el path del archivo que quiere traducir.
Por ejemplo: ./`java2cpp < ~/my_projects/cool_project/cool_file.java`

Si quiere probar puede usar el archivo `examples/alpha_input.java`

Resultado y traducción

El resultado de la traducción se guarda en el archivo java2cpp_translation.cc y en la consola se imprime si hubieron errores o no y alguna información adicional sobre java2cpp.

Ejemplo del mensaje en la consola de una traducción exitosa:

```
/*
*       ================================================================
*       Translated from java to c++ using java2cpp
*       Version:        alpha 1.0
*       Code owners:    https://github.com/martinezlucas98
*                       https://github.com/Joaquinecc
*       Translated on:  2021-06-18 04:01:52 (yyyy-MM-dd hh:mm:ss)
*       ================================================================
*/

Errors found: 0

Check the translation file: java2cpp_translation.cc

TRANSLATION SUCCESSFUL !!!
```

Ejemplo del mensaje en la consola de una traducción con errores:

```
/*
*       ============================================================
*       Translated from java to c++ using java2cpp
*       Version:        alpha 1.0
*       Code owners:    https://github.com/martinezlucas98
*                       https://github.com/Joaquinecc
*       Translated on:  2021-06-18 04:05:06 (yyyy-MM-dd hh:mm:ss)
*       ============================================================
*/

Errors found: 5

Check the translation file for more details: java2cpp_translation.cc

TRANSLATION FAILED !!!
```

Observación: cada vez que se ejecute el traductor java2cpp, se sobreescribirá el archivo java2cpp_translation.cc por lo que usted debe copiarlo y guardarlo donde desee y con el nombre que le guste.

Edición y archivos

Si usted se encuentra motivado para editar o modificar java2cpp, puede hacerlo editando los archivos java2cpp.l, java2cpp.y y type_conversion.h.

Recuerde que si usted modifica o altera algún archivo de java2cpp debe de compilarlo de nuevo como se explica en el apartado de Compilación (pág. 5)

A continuación explicamos brevemente la función de cada uno de los tres archivos mencionados.

java2cpp.l

Este es el archivo lex, aquí usted puede definir los tokens que quiera utilizando expresiones regulares (regex).

Este archivo contiene las especificaciones lex. Lex genera un escáner de lenguaje C a partir de una especificación fuente que usted escribe. Esta especificación contiene una lista de reglas que indican secuencias de caracteres (expresiones) que se deben buscar en un texto de entrada y las acciones que se deben realizar cuando se encuentra una expresión. Estas acciones van entre llaves.

Por ejemplo:

```
[$_a-zA-Z]+[$_a-zA-Z0-9]* {strcpy(yylval.var_name, yytext);return VAR;}
[0-9]+("."[0-9]+)?      {strcpy(yylval.var_name, yytext);return NUMBER;}
\"[^"]*\"         {strcpy(yylval.var_name, yytext);return QUOTED_STRING;}
```

La primera línea en la imagen define mediante expresión regular una combinación de caracteres que empiece con cualquier letra o con los símbolos $ _. Esto se almacena en `yyval.var_name` y se retorna el token `VAR`. Es decir, en esa primera línea define lo que es considerado una variable utilizando expresión regular.

Siguiendo la misma lógica, la segunda línea define lo que se considera un número (entero o con decimales) y le asigna esto al token `NUMBER`.

La tercera línea define una cadena de caracteres entre comillas dobles, `QUOTED_STRING`.

En este archivo nosotros decidimos definir los terminales que consideramos importantes, puede ver estos tokens en la sección de Anexos - Definiciones de los tokens lex (pág. 51).

java2cpp.y

Este es el archivo yacc, yacc lee una descripción de una gramática libre de contexto y escribe el código fuente C en un archivo de código. Aquí se definen mediante gramáticas libres de contexto la sintaxis del programa.

Por ejemplo, la definición de la sintaxis para el ciclo do-while:

```
DO_WHILE_LOOP : DO LC { write_to_file("do{\n"); tab_counter++;} STATEMENTS RC
WHILE LP {tab_counter--; print_tabs(); write_to_file("}while("); } DECL_EXPR
RP { write_to_file(")"); } MUST_SEMICOLON { write_to_file("\n"); }
              ;
```

También puede escribir funciones en el lenguaje C en este archivo, fuera del bloque marcado entre `%%  %%`

type_conversion.h

Este archivo de cabecera contiene las estructuras necesarias para la verificación de tipos. Es decir, contiene las tablas para validar conversiones de tipos con los operadores + - * / como también para validar la conversión implícita de un tipo a otro.

Para entender mejor, echemos una mirada a las definiciones de los tipos (y su equivalente en valor entero):

```
#define T_ERROR -1
#define T_INT 0
```

```
#define T_CHAR 1
#define T_FLOAT 2
#define T_DOUBLE 3
#define T_STRING 4
#define T_BOOL 5
```

A continuación se observa la matriz de conversión de tipos para el operador + (suma). La celda `[i][j]` representa el tipo de dato obtenido al evaluar una variable/constante del tipo `i` + una del tipo `j`:

```
struct T_table casting_table = {
        .add={
            { T_INT,    T_INT,      T_FLOAT,    T_DOUBLE,   T_ERROR,    T_INT     },
            { T_INT,    T_INT,      T_FLOAT,    T_DOUBLE,   T_STRING,   T_INT     },
            { T_FLOAT,  T_FLOAT,    T_FLOAT,    T_DOUBLE,   T_ERROR,    T_FLOAT   },
            { T_DOUBLE, T_DOUBLE,   T_DOUBLE,   T_DOUBLE,   T_ERROR,    T_DOUBLE  },
            { T_STRING, T_STRING,   T_STRING,   T_STRING,   T_STRING,   T_STRING  },
            { T_INT,    T_INT,      T_FLOAT,    T_DOUBLE,   T_ERROR,    T_INT     }
        },
```

En `[0][4]` se puede observar como la operación de un entero + string ocasiona un error ya que esta operación no es válida por motivos de conversiones de tipos.

Implementación

Features / características implementadas

A continuación se presentan las características o features implementados para la traducción.

También se presentan implementaciones simplificadas de sus gramáticas, estas implementaciones que se muestran en el documento son consideradas simplificadas ya que sólo se mostrarán lo que se considere relevante para su entendimiento. Esto debido a que si se muestra la implementación completa, estaríamos completando unas cuantas páginas solo con las gramáticas y funciones auxiliares.

Si usted se encuentra interesado en leer la implementación completa del código, puede leer los archivos `java2cpp.l`, `java2cpp.y` y `type_conversion.h`

Java2cpp utiliza tokens lex para representar a los terminales por lo que recomendamos leer el apartado de Definiciones de los tokens lex (pág. 51) antes de continuar.

Cabe mencionar que las librerías de `<iostream>`, `<string>` son automaticamentes agregadas mediante el `#include` al inicio del programa así como el `using namespace std;`

Identación / tabulación adecuada

Para una mejor pulcritud y entendimiento del output, implementamos una función para imprimir las tabulaciones necesarias `void print_tabs();`

Para calcular cuántas tabulaciones se deben imprimir, se creó una variable `int tab_counter` inicializada en cero. Esta variable aumenta y disminuye dependiendo de donde queremos agregar una tabulación extra o disminuir la cantidad de tabulaciones.

Por ejemplo, luego de abrir la llave '{' en una sentencia de ciclo `while` aumentamos esta variable `tab_counter` en uno, y cuando cerramos la llave de ese ciclo '}' disminuimos el valor de dicha variable en uno. Esto se puede ver a continuación.

```
WHILE_LOOP : WHILE LP {create_scope_name_and_push_it();write_to_file("while
("); } DECL_EXPR RP LC { tab_counter++; write_to_file("){\n"); } STATEMENTS
RC {pop_scope();tab_counter--; print_tabs(); write_to_file("}\n"); }
          ;
```

Declaraciones de variables y constantes (tabla de símbolos)

Las variables se copiaban directamente, excepto por algunas excepciones.
Para las declaraciones de variables `public` o `private` obviamos y lo considerábamos como variables globales del programa. Para las variables `final` se hizo la conversión de la palabra `final` a `const`.

Se mantiene una tabla de símbolos donde se almacenan las variables. Cada fila es una variable. Las columnas de la tabla son: tipo,nombre,scope, y si es constante. En cada asignación de variable se chequeaba si la variable a modificar era constante en caso de que si, se emite un mensaje de error.

Veamos también el valor numérico asignado a cada tipo de dato:

```
int = 0
char = 1
float = 2
double = 3
string = 4
bool = 5
```

Esto quiere decir que, por ejemplo, el número 4 representa al tipo de dato string.

Ciclos (Loops)

Se han implementado los ciclos `for`, `while` y `do-while`. Como estos ciclos son bastante similares entre java y c++, la traducción es bastante simple y directa.

Gramática simplificada (se omiten las funciones para mayor claridad):

```
WHILE_LOOP    : WHILE LP DECL_EXPR RP LC STATEMENTS RC
              ;

FOR_LOOP      : FOR LP FOR_PARAMS RP LC STATEMENTS RC
              ;

FOR_PARAMS    : DECL_EXPR SEMICOLON DECL_EXPR SEMICOLON EXPRESION
              | TYPE VAR COLON VAR
              ;

DO_WHILE_LOOP: DO LC STATEMENTS RC WHILE LP DECL_EXPR RP SEMICOLON
              ;

DECL_EXPR     : EXPRESION
              | TYPE VAR HAS_ASSIGNMENT
              | VAR HAS_ASSIGNMENT
              | /* */  { }
              ;
```

Estructuras condicionales

La estructura condicional `if-then` y sus derivaciones (`else`, `else if`) son similares entre java y c++. Esto implicó una traducción directa sin muchos inconvenientes.

Gramática simplificada (se omiten las funciones para mayor claridad):

```
IF_STATEMENT : IF LP MUST_EXPRESSION RP LC STATEMENTS RC ELSE_VARIATIONS
             ;

ELSE_VARIATIONS: ELSE LC STATEMENTS RC
               | ELSEIF LP MUST_EXPRESSION RP LC STATEMENTS RC
ELSE_VARIATIONS
               | /* */ { write_to_file("\n"); }
               ;
```

Estructuras anidadas

Las estructuras anidadas se garantizan utilizando recursividad en las gramáticas descritas en el archivo yacc java2cpp.y.

A continuación, algunos ejemplos (incompletos) de donde utilizamos recursividad.

```
EXPRESION    : EXPRESION LAND { write_to_file(" && "); } EXPRESION
             | EXPRESION LOR { write_to_file(" || "); } EXPRESION
```

```
            | EXPRESION LEQ { write_to_file(" <= "); } EXPRESION
            (etc etc)...
            | TERMINAL
            | VAR { write_to_file(yylval.var_name);
verify_scope(yylval.var_name);
add_exp_vect_var(48+lookup_in_table_alt(yylval.var_name)); }
           ;
```

En este primer ejemplo se ve como una expresión puede ver como utilizamos recursividad para construir las expresiones. Esto porque una expresión puede estar compuesta por otra.

```
STATEMENTS  : { print_tabs(); } DECLARATION STATEMENTS { }
            | { print_tabs(); } MAIN_METHOD_DECLARATION STATEMENTS { }
            | { print_tabs(); } COMMENT STATEMENTS { }
            | { print_tabs(); } IF_STATEMENT STATEMENTS { }
            (etc etc)...
            | /* */  { }
            ;
```

En este segundo ejemplo vemos la gramática de `STATEMENTS`, el cual tiene recursividad por la derecha. Esto permite que haya un `STATEMENT` tras otro. La gramática de `STATEMENT` define todo lo que puede ser parte del código, ya sean funciones, ciclos, condicionales, etc.

Procedimientos, métodos y funciones

Las funciones se copiaban directamente el nombre ,el tipo y los parámetros si este no eran arrays. Si tenían `private,static,public` obviamos ya que no teníamos en cuenta las clases no era un requisito).

Se mantiene una tabla de funciones. Cada fila representa una definición de una función o un llamado de una función. Las columnas son las siguientes: tipo (int), nombre, un array de los tipos de sus parámetros, un contador de este array y por último un flag (int) llamado `is_def` indica si fue una llamada de la función o una declaración de una función.

Al final de de leer el código input, se verifica que en la tabla de funciones todas las funciones con el flag `is_def=0` (es decir no una definición sino una llamada a una función) se hayan definido en alguna parte del código. Para ello se compara el nombre de la función, y los tipos de sus parámetros con las funciones de las tablas que tengan el flag `is_def=1` (es decir una declaración de una función). Si no se encuentra un match se emite un mensaje de error. Cabe mencionar que también se verifican definiciones de funciones repetidas.

Si todo sale correcto entonces se cargan los prototipos de todas las funciones al comienzo del archivo ouput.

Vectores

Para los vectores, es un poco más complicado. En java un vector se inicializa de la siguiente forma

```
double[] arr1 = new int[20]
```

Esto hay que convertir a la forma:

```
double arr1 [20];
```

Entonces se debe leer toda la linea para saber las dimensiones, en el caso de que sea una variable en vez de un número se copia la variable directamente.

En el caso de una inicialización. Ejemplo:

```
int[] arr4 = {5,4,3,2,1};
```

Se debe convertir en:

```
int arr4[] = {5,4,3,2,1};
```

La conversión es más simple, simplemente se pone los corchetes después del nombre. En caso de que los corchetes tengan una variable o un número se copia igualmente dentro de los corchetes después del nombre.

En caso de los parámetros, sería lo mismo, mover los corchetes después del nombre.

Arrays multidimensionales

Las inicializaciones y declaraciones son idénticas a los vectores unidimensionales mencionados arriba. La única diferencia es en los parámetros formales de las funciones. En java no hace falta indicar las dimensiones de los arrays multidimensionales en los parámetros formales. Como solución se propuso rellenar cada vez que se encuentra un array multidimensional en los parámetros como una constante definida.
Ejemplo:

```
private static void bar(int [][] num1, char num2)
```

Se traduce a:

```
void bar(int num1[][20],char num2)
```

Donde el número 20 se obtiene de la constante global `DIMENSION`.

Input/Output

Las funciones de input en Java que fueron implementadas son las funciones de `System.out.println()` y `System.out.print()`. La función de output de Java es `Scanner`.

Entonces para las funciones estándar de input y output, Java2cpp realiza las siguientes conversiones de Java (izquierda) a C++ (derecha):

`System.out.println(expr);`	`std::cout << expr << std::endl;`
`System.out.print(expr);`	`std::cout << expr ;`
`Scanner my_input = new Scanner(System.in);`	`string my_input; std::cin >> my_input;`

Líneas de comentarios:

Tanto Java como C++ tienen la misma sintaxis para comentarios de una línea o de múltiples líneas por lo que el mapeo de Java a C es bastante sencillo.

La definición de comentarios de una y múltiples líneas se encuentra en el archivo lex java2cpp.l y se puede ver a continuación.

```
"/*"(.|\n)*"*/"      {strcpy(yylval.var_name, yytext);return MLCOMMENT;}
"//"(.)*             {strcpy(yylval.var_name, yytext);return ILCOMMENT;}
```

Como se puede ver, al detectar mediante expresiones regulares el patrón del comentario, este retorna un token `ILCOMMENT` o `MLCOMMENT` dependiendo de si es una o múltiples líneas respectivamente.

Atención: el uso de comentarios de múltiples líneas `/* */` suele causar segmentation fault, aun así la traducción puede terminar exitosamente. De todos modos, recomendamos utilizar de ser posible solamente comentarios de una línea `//`.

Reglas de ámbito (scope)

La verificación de scope se hace al momento de leer la variable. Cada momento se mantiene una pila que contiene los scope. En el tope de la pila está el scope actual, y por debajo del tope se encuentran los antecesores.

La pila se inicializaba con el scope global. Cada vez que se entraba a una función, se insertaba en el tope de la pila y al finalizar se elimina del tope. En caso de if y loops, cada vez que se entraba se cargaba en el la pila con un nombre único y al finalizar el bloque se removía de la pila.

Para la verificación cada vez que se usa una variable, se busca en la tabla de símbolo, si se encuentra una variable con el mismo nombre se compara el scope actual con el scope de la variable de la tabla de símbolos. Si falla, se hace lo mismo solo que no se compara con el scope actual(el tope de la pila), si no con los antecesores antecesores. Es decir se busca en la tabla de símbolos una variable con el mismo nombre pero con el scope anterior, y así sucesivamente. Si no hubo ningún match correcto se emite un mensaje de variable no declarada.

Detección de múltiples declaraciones de la variable

Cada vez que se lee una variable, se inserta en la tabla de símbolos, no sin antes verificar que no se haya definido en el scope actual. En caso de si, se emite un mensaje de error “múltiples declaraciones de la variable [nombre de la variable] ”

Conversión y comprobación de tipos

Los tipos de datos fueron definidos por números enteros. Estos se mencionaron en la sección de estos en declaraciones de variables (pág. 10)

Muchas funciones están involucradas en la verificación y conversión de tipos. La lógica es la siguiente:

`int right_val_type` contendrá el tipo de dato del lado derecho de la expresión de asignación.

`int left_val_type` = el tipo de dato del lado izquierdo de la expresión de asignación o inicialización (es decir el tipo de dato de la variable). Para esto se busca la variable en la tabla de variables definidas y obtenemos su tipo de dicha tabla.

En el array/vector `expression_vect` se almacena la expresión del lado derecho, pero sustituyendo el nombre de las variables o las constantes con el número que representa a su tipo de dato.

Por ejemplo: `int a = 10 + 20` se almacenaría en `expression_vect` de la siguiente forma: `0+0` donde `0` representa el tipo de dato entero. Y el tipo de dato de la variable a es almacenada `en left_val_type`.

Luego se itera el vector `expression_vect` realizando las operaciones entre los tipos verificando con las tablas de matrices definidas en `type_conversion.h` hasta que quede un solo tipo de dato, el cual es el resultante de aplicar las diversas operaciones en el lado derecho. Si alguna operación no es compatible entre los tipos de datos, entonces se anuncia el error sin necesidad de realizar los pasos a continuación.

Ahora se asigna a `right_val_type` este entero que representa el tipo de dato del lado derecho y se compara con el valor almacenado en `left_val_type`. Si estos son iguales o es posible realizar una conversión implícita, entonces no se lanza un mensaje de error.

Para saber si es posible la conversión implícita se consulta con la matriz .implicit definida en la estructura `casting_table` en `type_conversion.h` que se ve a continuación.

```
// 0 = FALSE, 1 = TRUE, 2 = TRUE WITH WARNING
        .implicit={
            { 1, 1, 1, 1, 0, 1 },
            { 1, 1, 1, 1, 0, 1 },
            { 2, 2, 1, 1, 0, 2 },
            { 2, 2, 1, 1, 0, 2 },
            { 0, 0, 0, 0, 1, 1 },
            { 1, 1, 1, 1, 0, 1 }
        },
```

(Observación: los mensajes de warning, que en la tabla de arriba son representados con el número 2, generaban algunos segmentation fault por lo que fueron removidos o mejor dicho ignorados en esta versión)

Limitaciones

Una limitación importante es que javacpp solamente traduce archivos contenidos dentro de la clase Main y esta debe ser específicamente escrita de la siguiente forma:

```
public class Main { }
```

En Java hay varias maneras de declarar el método main, pero para que java2cpp funcione, este debe ser escrito como se ve a continuación:

```
public static void main(String[] args) { }
```

Y la declaración de arrays es de la forma:

```
int[] arr
```

No de la forma `int arr[]`

Una mayor limitación con la que cuenta el programa es que utiliza arrays o vectores con una longitud predeterminada. Esto puede causar que al llenarse el array se genere algún segmentation fault haciendo fallar al código.

Esto se puede solucionar aplicando las funciones de C de Malloc y Realloc más adelante.

```
int string_or_var[MAX_VARIABLES];
char syntax_errors[256] = "";
```

```
char type_cast_str_error[256] = "";
```

(arriba se ven algunos arrays con longitudes predefinidas)

También podemos considerar como una limitación el hecho de que para que java2cpp funcione se requiere obligatoriamente de la clase `Main`, esto se puede ver en la gramática inicial `program` en `java2cpp.y`

```
program : {fp_aux=fopen(AUXFILE,"w");print_init();} HAS_COMMENT MAIN_CLASS LC {push_s...
        | /* Empty file */  { write_to_file("\n"); exit(2); }
        ;
```

(AUXFILE se define como el nombre del archivo auxiliar donde se va escribiendo la traducción antes de juntarla con los prototipos de funciones necesarios).

Otra limitación que nos encontramos fue con los array multidimensionales en los parámetros formales de las funciones. En Java no se debe reservar memoria de antemano al usar array multidimensionales.

Ejemplo:

```
private static void bar(int [][] num1, char num2)
```

Como podemos notar los corchetes están vacíos, esto es un problema en C++, como solución se propuso rellenar los corchetes(excepto el primero) con una constante predefinida en el código.

Resultado:

```
void bar(int num1[][20],char num2)
```

Detección y recuperación de errores

El traductor java2cpp implementa dos tipos de detección y recuperación de errores para los errores léxicos, de modo pánico y producciones de error. Java2cpp utiliza el modo pánico para recuperarse de errores no definidos como producciones de error.

Modo pánico

Al descubrir un error, el analizador sintáctico desecha símbolos de entrada, de uno en uno, hasta que encuentra uno perteneciente a un conjunto designado de componentes léxicos de sincronización / delimitadores. Esto se conoce como recuperación en modo pánico.

BISON cuenta con un token especial llamado `error` que se utiliza para recuperarse de un error sintáctico mediante el modo pánico. A continuación de este token `error` debe haber un token delimitador, como por ejemplo el punto y coma.

A continuación se ve cómo utilizamos el token error:

```
STATEMENTS : { print_tabs(); } DECLARATION STATEMENTS { }
           | { print_tabs(); } MAIN_METHOD_DECLARATION STATEMENTS { }
           | { print_tabs(); } COMMENT STATEMENTS { }
           | { print_tabs(); } IF_STATEMENT STATEMENTS { }
           | { print_tabs(); } FOR_LOOP STATEMENTS { }
           | { print_tabs(); } WHILE_LOOP STATEMENTS { }
           | { print_tabs(); } DO_WHILE_LOOP STATEMENTS { }
           | { print_tabs(); } STDIO STATEMENTS { }
           | { print_tabs(); } BREAK_ST STATEMENTS { }
           | { print_tabs(); } RETURN_ST STATEMENTS { }
           | VAR_USE STATEMENTS { }
           | error DELIMITER STATEMENTS
           | /* */ { }
           ;
```

Donde `DELIMITER` es el componente léxico de sincronización o delimitador para el modo pánico.

Producciones de error

Consiste en aumentar la gramática del lenguaje con producciones que generen construcciones erróneas. Entonces se usa esta gramática aumentada con las producciones de error para construir el analizador sintáctico.

Si el analizador sintáctico usa una producción de error, se pueden generar diagnósticos de error apropiados para indicar la construcción errónea reconocida en la entrada.

Las producciones de error implementadas en java2cpp son las de:

- punto y coma faltante al final: alerta cuando faltó agregar el punto y coma al final de una sentencia.

```
MUST_SEMICOLON : SEMICOLON { write_to_file(";"); }
               | /*empty*/ { yyerror("Syntax error: expected ';'
at end of declaration"); }
               ;
```

- punto y coma esperado en vez de la coma: alerta cuando se utilizaron comas en vez de punto y comas en los parámetros del ciclo `for`

```
SEMICOLON_NOT_COMA : SEMICOLON { write_to_file(";"); }
                   | COMA { write_to_file(",");
strcat(syntax_errors,"Syntax error: expected ';' instead of
','\t"); }
                   ;
```

- signo == esperado en vez de = entre dos expresiones (ej: 1=1) : alerta cuando se encontró una asignación a una expresión en vez de una comparación en la condición del `if-then`

```
MUST_EXPRESSION : EXPRESION
                | VAR ASSIGNMENT { write_to_file(yylval.var_name);
write_to_file("="); } EXPRESION { }
                | EXPRESION ASSIGNMENT { write_to_file("="); }
EXPRESION { strcat(syntax_errors,"Syntax error: cannot assign to
an expression. Expected '==' operator\n"); }
                | /*empty*/ { strcat(syntax_errors,"Syntax error:
expected expression\n"); }
                ;
```

- expresión esperada: alerta cuando no hay ninguna expresión en los parámetros de la condición del `if-then`

```
MUST_EXPRESSION : EXPRESION
                | VAR ASSIGNMENT { write_to_file(yylval.var_name);
write_to_file("="); } EXPRESION { }
                | EXPRESION ASSIGNMENT { write_to_file("="); }
EXPRESION { strcat(syntax_errors,"Syntax error: cannot assign to
an expression. Expected '==' operator\n"); }
                | /*empty*/ { strcat(syntax_errors,"Syntax error:
expected expression\n"); }
                ;
```

Casos de prueba

Entre el contenido del archivo .zip se encuentran archivos .java para probar cada una de las funciones o características implementadas y también algunos códigos generales para probar como uno de busqueda linear, otro para hallar el MCD, etc. Puede leer más sobre estos en la sección de algoritmos de pruebas (pág.45).

Favor revisar el apartado de Ejecución (pág. 6) para entender cómo ejecutar java2cpp con un archivo de entrada.

A continuación se muestran los las traducciones de cada caso de prueba, señalando algunas cualidades. Para más detalles puede leer el apartado de Implementación (pág. 9)

Pruebas individuales por cada feature

Los archivos de pruebas específicos para las características implementadas se encuentran bajo el directorio de `examples/modules/`

En la tabla continuación puede ver el nombre de los archivos de prueba para cada característica.

Características / features	**Nombre del archivo de prueba***
Declaraciones de variables y constantes (tabla de símbolos)	types_test.java const_test.java
Ciclos (Loops)	do_while_test.java for_loop_test.java while_loop_test.java nested_for_test.java
Estructuras condicionales	if_else_if_else_test.java if_else_if_test.java if_else_test.java if_statement_test.java nested_if_test.java
Estructuras anidadas	nested_if_test.java nested_for_test.java function_call_inside_another_test
Procedimientos, métodos y funciones	methods_test.java function_call_inside_another_test
Vectores	array_test.java
Arrays multidimensionales	array_test.java
Input/Output	print_test.java input_test.java
Líneas de comentarios	comments_test.java
Reglas de ámbito (scope)	scope_if_statment_test.java scope_loop.java scope_var.java
Conversión y comprobación de tipos	type_error_test.java

* Recordar que estos archivos se encuentran en el directorio `examples/modules/` por lo que para ejecutar java2cpp con el archivo methods_test.java haríamos:
`./java2cpp < examples/modules/methods_test.java`

Pruebas individuales de errores

Los archivos de pruebas específicos para los errores detectables se encuentran bajo el directorio de `examples/error_tests/`

En la tabla continuación puede ver el nombre de los archivos de prueba para cada característica

Tipo de error	**Nombre del archivo de prueba****
Tipos no compatibles o no “casteable”	type_error_test.java
Error de sintaxis	syntax_error_test.java
Múltiples declaraciones de variable	multiple_declaration.java
Variable no declarada en el scope	scope_test.java
No se puede modificar una constante	const_test.java
Método / función no declarada	function_not_declared.java

** Recordar que estos archivos se encuentran en el directorio `examples/error/` por lo que para ejecutar java2cpp con el archivo methods_test.java haríamos:
`./java2cpp < examples/error_tests/type_error_test.java`

Ejemplos traducidos y explicados

Pruebas de características / features

types_test.java

Este test tiene la finalidad de probar si funcionan las declaraciones de variables con los tipos de datos simples implementados (hay un test aparte para vectores y matrices).

Input:

```
public class Main {
 public static void main(String[] args) {
   int var1;
   double var2;
   float var3;
```

```
   boolean var4;
   char var5;
   String var6;
   int var1_2 = 10;
   double var2_2 = 10.134;
   float var3_2 = 10.134;
   boolean var4_2 = true;
   boolean var4_2_2 = false;
   char var5_2 = 'c';
   String var6_2 = "hello";
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
     int var1;
     double var2;
     float var3;
     bool var4;
     char var5;
     std::string var6;
     int var1_2 = 10;
     double var2_2 = 10.134;
     float var3_2 = 10.134;
     bool var4_2 = true;
     bool var4_2_2 = false;
     char var5_2 = 'c';
     std::string var6_2 = "hello";
}
/* end Main Class */
```

do_while_test.java

Este test tiene la finalidad de probar la correcta traducción del ciclo do-while.

Input:

```java
public class Main {
 public static void main(String[] args) {
   do {
     // code block to be executed
     break;
   }while(true);
   int a;
 }
}
```

Output:

```cpp
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
   do{
       // code block to be executed
       break;
   }while(true);
   int a;
}
/* end Main Class */
```

for_loop_test.java

Este test tiene la finalidad de probar la correcta traducción del ciclo for y su variante for-in.

Input:

```java
public class Main {
 public static void main(String[] args) {
   for (int i = 0; i < 5; i++) {
       // System.out.println(i);
   }
   for (String i : cars) {
       // System.out.println(i);
   }
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
   for (int i = 0;i < 5;i++) {
       // System.out.println(i);
   }
   for (std::string i : cars) {
       // System.out.println(i);
   }
}
```

while_loop_test.java

Este test tiene la finalidad de probar la correcta traducción del ciclo while.

Input:

```
public class Main {
 public static void main(String[] args) {
   while (true) {
     // code block to be executed
     break;
   }
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
   while (true){
       // code block to be executed
       break;
   }
}
```

if_else_if_else_test.java

Este test tiene la finalidad de probar la correcta traducción de la variante de la sentencia if que contiene if-elseif-else.

Input:

```
public class Main {
 public static void main(String[] args) {
   if (20 > 18) {
     // code block to be executed
   }else if (30 > 25) {
     // code block to be executed
   }else {
     // code block to be executed
   }
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
   if (20 > 18) {
       // code block to be executed
   } else if (30 > 25)) {
       // code block to be executed
   } else {
       // code block to be executed
   }
}
/* end Main Class */
```

if_else_if_test.java

Este test tiene la finalidad de probar la correcta traducción de la variante de la sentencia if que contiene if-elseif.

Input:

```
public class Main {
 public static void main(String[] args) {
   if (20 > 18) {
```

```
    // code block to be executed
  }else if (30 > 25) {
    // code block to be executed
  }
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
  if (20 > 18) {
      // code block to be executed
  } else if (30 > 25)) {
      // code block to be executed
  }
}
/* end Main Class */
```

if_else_test.java

Este test tiene la finalidad de probar la correcta traducción de la variante de la sentencia if que contiene if-else.

Input:

```
public class Main {
 public static void main(String[] args) {
  if (20 > 18) {
    // code block to be executed
  }else {
    // code block to be executed
  }
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
```

```
/* start Main Class */
int main(int argc, char **argv){
   if (20 > 18) {
       // code block to be executed
   } else {
       // code block to be executed
   }
}
/* end Main Class */
```

if_statement_test.java

Este test tiene la finalidad de probar la correcta traducción de la variante de la sentencia if-then.

Input:

```
public class Main {
 public static void main(String[] args) {
   if (20 > 18) {
     // code block to be executed
   }
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
   if (20 > 18) {
       // code block to be executed
   }
}
/* end Main Class
```

nested_if_test.java

Este test tiene la finalidad de probar la correcta traducción de varias sentencias del tipo if-then y sus variantes anidadas.

Input:

```
public class Main {
 public static void main(String[] args) {
```

```
   if (20 > 18) {
     // code block to be executed
   }else if (30 > 25) {
     // code block to be executed
     if (20 > 18) {
       // code block to be executed
     }else if (30 > 25) {
       // code block to be executed
     }
   }
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
   if (20 > 18) {
       // code block to be executed
   } else if (30 > 25)) {
       // code block to be executed
       if (20 > 18) {
           // code block to be executed
       } else if (30 > 25)) {
           // code block to be executed
       }
   }
}
/* end Main Class */
```

nested_for_test.java

Este test tiene la finalidad de probar la correcta traducción de sentencias del tipo for anidadas.

Input:

```
public class Main {
 public static void main(String[] args) {
   for (int i = 0; i < 5; i++) {
       for (int i = 0; i < 5; i++) {
           // System.out.println(i);
```

```
        }
    }
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
    for (int i = 0;i < 5;i++) {
        for (int i = 0;i < 5;i++) {
            // System.out.println(i);
        }
    }
}
/* end Main Class */
```

methods_test.java

Este test tiene la finalidad de probar la correcta traducción de las declaraciones de métodos. También traduce los parámetros formales de las funciones.

Input:

```
public class Main {
 public static void main(String[] args) {

 }
 public static void funbar(int num1, char num2) {
    int x;
}

 private static void bar(int [][] num1, char num2) {
     int [][] arr=new int [10][2];
     int x;
     int c= 1+ 2;
     c=1+2;
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
int funbar(int,char);
int bar(char);
/* start Main Class */
int main(int argc, char **argv){

}
void funbar(int num1,char num2){
   int x;
}
void bar(int num1[][20],char num2){
   int arr[10][2];
   int x;
   int c = 1 + 2;
   c = 1 + 2;
}
/* end Main Class */
```

Explicación del output:
Los parámetros que no son array se copian igual. Los arrays multidimensionales rellena los corchetes(excepto el primero) con una constante, en este caso 20.

array_test.java

Este test tiene la finalidad de probar la correcta traducción de las declaraciones de vectores y matrices multidimensionales.

Input:

```
public class Main {
 public static void main(String[] args) {
   int m =5;
   int n =6;
   double[] arr1 = new int[m];
   double[] arr2 = new int[m][n];
   double[] arr3 = new int[5];
   int[] arr4 = {5,4,3,2,1};
   double[][] arr5 = new int[3][5];
   int[][] arr6 = {{1,2,3},{4,5,6},{7,8,9}};
   arr1[0] = 10;
   arr2[0][1] = 90;
 }
```

```
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
     int m = 5;
     int n = 6;
     double arr1[m];
     double arr2[m][n];
     double arr3[5];
     int arr4[] = {5,4,3,2,1};
     double arr5[3][5];
     int arr6[][] = {{1,2,3},{4,5,6},{7,8,9}};
     arr1[0] = 10;
     arr2[0][1] = 90;
}
/* end Main Class */
```

print_test.java

Este test tiene la finalidad de probar la correcta traducción de las funciones clásicas de output.

Input:

```
public class Main {
 public static void main(String[] args) {
   System.out.print("hello");
   System.out.println(" world");
   System.out.println("hello" + " world");
   System.out.print("hello" + " world");
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
```

```
int main(int argc, char **argv){
   std::cout << "hello";
   std::cout << " world" <<  std::endl;
   std::cout << "hello" << " world" <<  std::endl;
   std::cout << "hello" << " world";
}
/* end Main Class */
```

Explicación:
Como se puede observar, para el println java2cpp agrega el caracter de nueva linea al final de std::cout. También otra cosa llamativa es que para concatenar Java utiliza el signo +, pero en C++ usamos simplemente <<

input_test.java

Prueba la correcta traducción de la función Scanner de Java a la función std::cin de C++

Input:

```
public class Main {
 public static void main(String[] args) {
   Scanner my_input = new Scanner(System.in);
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
   std::string my_input;
   std::cin >> my_input;
}
/* end Main Class */
```

comments_test.java

Este test tiene la finalidad de probar la correcta traducción de los comentarios.

Input:

```
// comment out
public class Main {
 public static void main(String[] args) {
     // inline comment
```

```
     /*
     multiple lines
     comment
     */
 }
}
//comment out 2
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
// comment out
/* start Main Class */
int main(int argc, char **argv){
     // inline comment
     /*
```

function_call_inside_another_test.java

De que trata el test: Verificar que todas las funciones llamadas, están previamente declaradas.

Input:

```
public class Main {
 public static void main(String[] args) {
 }
 public static void funbar(int num1, char num2) {
   int x;
}
 private static void bar(int num1, char num2) {
       int x;
       x=2;
       funbar(x,'a');
       funTest("tesst");
   }
 public static void funTest(String test){
 }
}
```

Output:

```
#include <iostream>
```

```
#include <string>
using namespace std;
int funbar(int,char);
int bar(int,char);
string funTest(string);
/* start Main Class */
int main(int argc, char **argv){

}
void funbar(int num1,char num2){
   int x;
}
void bar(int num1,char num2){
   int x;
   x = 2;
   funbar(x,'a');
   funTest("tesst");
}
void funTest(std::string test ){
}
/* end Main Class */
```

Explicación del output:
No se emite mensaje de error porque todas las funciones llamadas dentro de la función bar están definidas en alguna parte del código

scope_if_statment_test.java

De que trata el test: Demostrar el scope de los bloques if.

Input:

```
public class Main {
 public static int p;
 public static int g;
 public static void funasd(){
   int x;
   if (20 > 18) {
     int x;
     int c=x+1;
   }else if (30 > 25) {
     int c=3;
     int h=c+2;
     int d=x;
```

```
   }
   if(100>10){
     int d=x+g;
   }
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
int funasd();
/* start Main Class */
int p;
int g;
void funasd( ){
   int x;
   if (20 > 18) {
       int x;
       int c = x + 1;
   } else if (30 > 25)) {
       int c = 3;
       int h = c + 2;
       int d = x;
   }
   if (100 > 10) {
       int d = x + g;
   }
}
/* end Main Class */
```

Explicación del output:
No se emite mensaje de error porque las variables utilizadas son definidas previamente en un scope alcanzable.

scope_loop.java

De que trata el test:Demostrar el scope de los bloques de loop (wihle, for).

Input:

```
public class Main {
 public static    int x;
 public static void main(String[] args) {
```

```
   while (true) {
     int d=x;
   }

   for (int i = 0; i < 5; i++) {
       int d=x+i;
   }

 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int x;
int main(int argc, char **argv){
   while (true){
       int d = x;
   }
   for (int i = 0;i < 5;i++) {
       int d = x + i;
   }
}
/* end Main Class */
```

Explicación del output:
No se emite mensaje de error porque las variables utilizadas son definidas previamente en un scope alcanzable.

scope_var.java

De que trata el test: Demostrar el correcto funcionamiento de los alcanze de las variables

Input:

```
public class Main {
 public static int x;
 public static void funbar(int num1, char num2) {
   int x;
```

```
 }
public static void funbar2(int d) {
   int c=d+2;
 }
 private static void bar() {
     int c=x+1;
}
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
int funbar(int,char);
int funbar2(int);
int bar();
/* start Main Class */
int x;
void funbar(int num1,char num2){
   int x;
}
void funbar2(int d ){
   int c = d + 2;
}
void bar( ){
   int c = x + 1;
}
/* end Main Class */
```

Pruebas de detección de errores

type_error_test.java

Este test tiene la finalidad de probar el correcto funcionamiento de la comprobación de tipos.

Input:

```
public class Main {
 public static void main(String[] args) {
   int a = 12;
```

```
   String b = "10";
   int c = a + b;
   int x = ((1 + 'c')-("string"+2)/(3+2));
   int y = "fdfd";
   float z = a + b;

   a = "hola";
   int myvar;
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
     int a = 12;
     std::string b = "10";
     int c = a + b;
^^^^^^^^
Error on java file line [5] :: Error: No conversion for INT, STRING
with "+" operator
     int x = ((1 + 'c') - ("string" + 2) / (3 + 2));
^^^^^^^^
Error on java file line [6] :: Error: No conversion for INT, STRING
with "-" operator
     int y = "fdfd";
^^^^^^^^
Error on java file line [7] :: Error: implicit cast: Cannot cast from
STRING to INT
     float z = a + b;
^^^^^^^^
Error on java file line [8] :: Error: No conversion for INT, STRING
with "+" operator
     a = "hola";
^^^^^^^^
Error on java file line [10] :: Error: implicit cast: Cannot cast from
STRING to INT
     int myvar;
}
/* end Main Class */
```

Explicación del output:
El primer error detectado se da en la línea 5 de nuestro archivo de input (Java). Este error nos dice que no se puede convertir un tipo `int` a un tipo `string` utilizando el operador +, es decir no es posible hacer: entero + cadena.
Esto ocurre en la declaración `int c = a + b;` donde `a` es de tipo entero y `b` es una cadena de caracteres.

El siguiente error ocurre en la línea 6 de Java, similar al anterior pero con relación al operador -. Se puede observar aquí que la detección de tipos funciona en expresiones un poco más complejas.

El tercer error, que ocurre en la línea 7 del archivo Java, alerta que no existe una conversión implícita de un tipo de dato `string` a `int`.

El cuarto error ocurre en la línea 8 del archivo de entrada. Este error es similar al primero detectado, pero con una variable de tipo float.

El último error es similar al tercero mencionado y ocurre en la línea 10 del archivo Java.

syntax_error_test.java

Este test tiene la finalidad de probar el correcto funcionamiento de la detección de errores en la sintaxis (modo pánico y producciones de error).

Input:

```
public class Main {
 public static void main(String[] args) {
   int a =3+6
   int b = 2;
   int c = 100+b;
   if(1=1){
     // do something
     int s =
   }
   int i;
   for(i=0, i<90, i++){
     //something
   }
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
     int a = 3 + 6
^^^^^^^^
Error on java file line [4] :: Syntax error: expected ';' at end of
declaration
     int b = 2;
     int c = 100 + b;
     if (1=1) {
^^^^^^^^^^^^^^^^
Error on java file line [6] :: Syntax error: cannot assign to an
expression. Expected '==' operator
          // do something
          int s =
^^^^^^^^^^^^^^^^
Error on java file line [9] :: Syntax error: expected ';' at end of
declaration
^^^^^^^^^^^^^^^^
Error on java file line [9] :: Syntax error: expected expression
     }
     int i;
     for (i = 0,i < 90,i++) {
^^^^^^^^^^^^^^^^
Error on java file line [11] :: Syntax error: expected ';' instead of
','     Syntax error: expected ';' instead of ','
          //something
     }
}
/* end Main Class */
```

Explicación del output:
El primer error detectado se da en la línea 3 del archivo de input (Java) donde efectivamente falta el punto y coma al final de la declaración de la variable `int a`. Esto se detecta usando la producción de error adecuada. Se alerta que el error ocurrió en la línea 4 porque necesita el punto y coma antes de esa línea.

El siguiente error detectado es que en la condición del if se intenta una asignación a una expresión en vez de una comparación entre ellas.

Otro error, ocurrido en la línea 8 del archivo Java, nos dice que está esperando una expresión para la asignación de `int s` y que también espera un punto y coma al final. Se alerta que el error ocurrió en la línea 9 porque necesita el punto y coma antes de esa línea.

El último error detectado en este archivo de prueba se da en los parámetros del ciclo for donde en el archivo Java en la línea 11 se han escrito comas en vez de punto y coma. Este error es detectado y anunciado exitosamente. Cómo pudo haberse dado cuenta, este error es anunciado dos veces, uno por cada coma que ha puesto en vez de un punto y coma.

multiple_declaration.java

De que trata el test: Verificación de uso repetitivo de nombres de variables en el mismo scope.

Input:

```
public class Main {
  public static int x;

  public static void funbar(int d) {
     int x;
     int d;
     int x;
   }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
int funbar(int);
/* start Main Class */
int x;
void funbar(int d ){
   int x;
   int d;
^^^^^^^^
Error on java file line [6] :: Multiple declaration of variable d
   int x;
^^^^^^^^
Error on java file line [7] :: Multiple declaration of variable x
}
/* end Main Class */
```

Explicación del output:
Se declara múltiples veces la variable x en el mismo scope, esto ocurre en la línea 7 del archivo de entrada Java. También se vuelve a declarar la variable d en la línea 6, está ya fue definida dentro de los parámetros de la función.

scope_error.java

De que trata el test: Uso de una variable no definida en el scope o scope ancestrales.

Input:

```
public class Main {
 public static int x;
 private static void bar() {
     int c=x+1;
     c=d+2 ;
     if(100>10){
       int d;
       if(true){
         int g;
       }
       d=x+g;
     }
}
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
int bar();
/* start Main Class */
int x;
void bar( ){
   int c = x + 1;
   c = d
^^^^^^^^
Error on java file line [5] :: Variable d was not declared in the scope
^^^^^^^^
Error on java file line [5] :: Variable d was not declared in the scope
+ 2;
   if (100 > 10) {
```

```
       int d;
       if (true) {
           int g;
       }
       d = x + g
^^^^^^^^^^^^^^^^
Error on java file line [11] :: Variable g was not declared in the
scope
^^^^^^^^^^^^^^^^
Error on java file line [11] :: Variable g was not declared in the
scope
;
   }
}
/* end Main Class */
```

Explicación del output:
En la línea 5 del archivo de entrada Java se utiliza una variable `d` no declarada.
En la línea 11 del Java se quiere acceder a la variable `g`, pero no está en un scope alcanzable.

const_test.java

De que trata el test: Verificación de una variable constante no sea modificada

Input:

```
public class Main {
 public static final int x;
 public static void funbar(int num1, char num2) {
   int d=x+1;
   x=2;
 }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
int funbar(int,char);
/* start Main Class */
const int x;
void funbar(int num1,char num2){
```

```
   int d = x + 1;
   x = 2;
^^^^^^^^
Error on java file line [5] :: Cannot modify variable's value. Variable
x was declared as const
}
/* end Main Class */
```

Explicación del output:
La variable `x` se definió como una constante global, y se intenta asignarle un valor posteriormente en la línea 8 del archivo de entrada.

function_not_declared.java

De que trata el test: Llamada a funciones que no se han declarado.

Input:

```
public class Main {
 public static void main(String[] args) {
 }
 public static void funbar(int num1, char num2) {
   int x;
}
 private static void bar(int num1, char num2) {
       int x;
       x=2;
       funbar(x,'a');
       funTest("tesst");
   }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
int funbar(int,char);
int bar(int,char);
/* start Main Class */
int main(int argc, char **argv){

}
void funbar(int num1,char num2){
```

```
   int x;
}
void bar(int num1,char num2){
   int x;
   x = 2;
   funbar(x,'a');
   funTest("tesst");
}
/* end Main Class */
Error on java file :: Function char funTest(string) called but not
declared
```

Explicación del output:
Se llama en la línea 11 (del archivo de entrada) a la función `funTest` no declarada en ninguna parte. Esto se alerta al final del programa.

Algoritmos de pruebas

Tambíen el .zip cuenta con unos algoritmos de prueba más útiles que los tests, son ejemplos de código sencillo de java para probar su traducción. Estos archivos se encuentran bajo el directorio `examples/`
Estos son:

- **fibonacci.java:** algoritmo que calcula los diez primeros números de la secuencia fibonacci.
- **linear_search.java:** realiza una búsqueda lineal sobre un array buscando el número 50.
- **mcd.java:** algoritmo para calcular el MCD (o GCD en ingles) de dos números x=8, y=12.
- **reverse_num.java:** agortimo para invertir un numero. En este caso el 987654.

Por ejemplo si quiere traducir mcd.java debe ejecutar en la consola:
`./java2cpp < examples/mcd.java`

A continuación se muestra cada uno de estos algoritmos (input) y su traducción a C++ (output).

fibonacci.java

Input:

```
public class Main{
   public static void main(String[] args){
       int n1=0;
       int n2=1;
```

```
        int n3;
        int i;
        int count=10;

        System.out.print(n1+" "+n2);

        for(i=2;i<count;i++) {
            n3=n1+n2;
            System.out.print(" "+n3);
            n1=n2;
            n2=n3;
        }
    }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
    int n1 = 0;
    int n2 = 1;
    int n3;
    int i;
    int count = 10;
    std::cout << n1 << "-" << n2;
    for (i = 2;i < count;i++) {
        n3 = n1 + n2;
        std::cout << " " << n3;
        n1 = n2;
        n2 = n3;
    }
}
/* end Main Class */
```

linear_search.java

Input:

```
public class Main{
      public static int len = 6;
      public static int[] arr= {10,20,30,50,70,90};
```

```
public static int linearSearch(int key){
       int r = -1;
       for (int i = 0;i < len;i++) {
           if(arr[i] == key){
               r = i;
               break;
           }

       }
       System.out.println(key+" is found at index: "+r);
       return r;
   }
   public static void main(String[] args){
       int key = 50;
       linearSearch(key);
   }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
int linearSearch(int);
/* start Main Class */
int len = 6;
int arr[] = {10,20,30,50,70,90};
int linearSearch(int key ){
   int r = -1;
   for (int i = 0;i < len;i++) {
       if (arr[i] == key) {
           r = i;
           break;
       }
   }
   std::cout << key << " is found at index: " << r <<  std::endl;
   return r;
}
int main(int argc, char **argv){
   int key = 50;
   linearSearch(key);
}
/* end Main Class */
```

mcd.java

Input:

```
public class Main {
   public static void main(String[] args)   {
   int x = 12; int y = 8; int gcd = 1;
   for(int i = 1; i <= x && i <= y; i++)  {
       if(x%i==0 && y%i==0){
           gcd = i;
       }
   }
   System.out.print("GCD of "+x+" and "+y+" is: "+gcd);
   }
}
```

Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
   int x = 12;
   int y = 8;
   int gcd = 1;
   for (int i = 1;i <= x && i <= y;i++) {
       if (x % i == 0 && y % i == 0) {
           gcd = i;
       }
   }
   std::cout << "GCD of " << x << " and " << y << " is: " << gcd;
}
/* end Main Class */
```

reverse_num.java

Input:

```
public class Main  {
```

```
   public static void main(String[] args)  {
        int number = 987654;
        int number_backup = number;
        int reverse = 0;
        while(number != 0){
            int remainder = number % 10;
            reverse = reverse * 10 + remainder;
            number = number/10;
        }
        System.out.println("The reverse of the number " + number_backup
+ " is: " + reverse);
   }
}
```

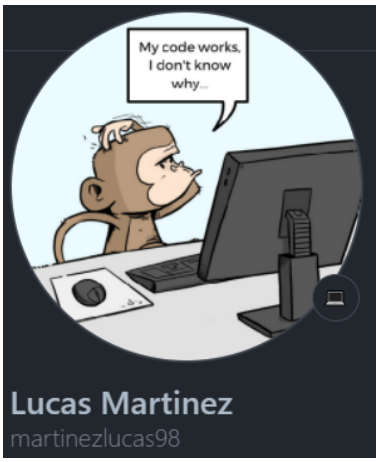
Output:

```
#include <iostream>
#include <string>
using namespace std;
/* start Main Class */
int main(int argc, char **argv){
   int number = 987654;
   int number_backup = number;
   int reverse = 0;
   while (number != 0){
        int remainder = number % 10;
        reverse = reverse * 10 + remainder;
        number = number / 10;
   }
   std::cout << "The reverse of the number " << number_backup << " is:
" << reverse <<  std::endl;
}
/* end Main Class */
```

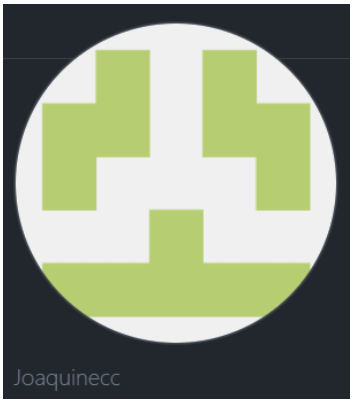
Aspectos Legales

Información sobre los code owners

Java2cpp fue creado y desarrollado por:



Lucas Martínez
Github: [martinezlucas98]



Joaquin Caballero
Github: [Joaquinecc]

Licencia de java2cpp

Java2cpp se encuentra licenciado bajo Apache License 2.0

Anexo

Definiciones de los tokens lex

Tokens lógicos y matemáticos.

Token	**Significado**
LAND	Operador lógico AND
LOR	Operador lógico OR
GEQ	Mayor o igual
LEQ	Menor o igual
NOT	Operador lógico de negación
GT	Mayor que
LT	Menor que
NEQ	No igual (diferente a)
DEQ	Doble igual ==
ASSIGNMENT	Símbolo de asignación
PLUS	Símbolo de suma
MINUS	Símbolo de resta
MUL	Símbolo de multiplicación
DIV	Símbolo de división
MOD	Símbolo de módulo

Otros tokens

Token	**Significado**
LP	Símbolo paréntesis izquierdo (
RP	Símbolo paréntesis derecho)
LC	Símbolo llave izquierda {

RC	Símbolo llave derecho }
LB	Símbolo corchete izquierdo [
RB	Símbolo corchete derecho]
COMA	Símbolo de coma
SEMICOLON	Símbolo de punto y coma
COLON	Símbolo dos puntos