Y_Compiler

El funcionamiento del compilador esta dividido en varias fases para ayudar al usuario a estar consciente de todas las fases de un compilador y poder observar con detalle como estas se llevan a cabo y de los distintos tipos de errores que pueden ocurrir durante cualquiera de estas fases.

El orden de eventos normal para un proceso de compilacion exitoso es el siguiente:

1. **Obtener el codigo fuente.**

   Este paso puede ser conseguido de dos formas: subiendo un archivo de texto a la aplicacion o escribiendo el nombre de un nuevo archivo, seleccionando la opcion “nuevo” y escribiendo a mano el codigo fuente.

2. **Compilar el codigo fuente**

   Al hacer click sobre el boton “compilar” dara comienzo el proceso de compilacion. La primer etapa es el analisis lexico, luego se realiza el analisis sintactico, luego se construye el AST (Abstract Syntax Tree) , y finalmente se emplea este arbol para extraer toda la informacion util de la entrada, se procesa y compila. Como resultado se obtiene:

   1. **Tabla de Simbolos**

      Una tabla empleada por el compilador en la siguiente fase que contiene toda la informacion necesaria sobre todas las funciones/metodos/constructores/sub-bloques, etc (Scopes para abreviar), detalla todos los Scopes, sus tamanios y el offset de cada variable dentro del sistema relativo a su Scope. Esta tabla tambien muestra varios otros detalles importantes para la compilacion. Las clases no se listan aqui.

   2. **Lista de Clases**

      Las clases se compilan y almacenan en una estructura diferente de la Tabla de Simbolos. Esto porque todos los campos de una clase son relativos a esa instancia de clase, no aun scope determinado (como es el caso de las variables locales) Al finalizar la compilacion se debe mostrar una lista de todas las clases ingresadas en el sistema junto con toda la informacion perteneciente a estas, incluyendo las clases nativas (Object y String) asi como toda la informacion obtenida de un proceso de heredacion.

   3. **AST**

      El AST (Abstract Syntax Tree) es una representacion en forma de Arbol de la sintaxis de la entrada, es util para procesar toda la informacion contenida en esta.

   Todas y cada una de estas 3 estructuras es mostrada al usuario si la compilacion fue exitosa, cada una se encuentra ubicada en su propia tabla y se puede navegar de una a otra con total libertad. Tras compilar todas estas estructuras exitosamente puede comenzar la siguiente fase.

3. Generar Codigo Intermedio

El codigo intermedio empleado por este compilador es Codigo 3 Direcciones. (Llamado asi porque solo se puede referenciar a 3 posiciones de memoria al mismo tiempo). Si ocurre un error durante la etapa de compilacion, se le notificara del error y no se le permitira continuar a esta fase. Sin embargo, si el proceso es exitoso entonces el nombre del boton cambia: “Finish compilation” . Y el usuario podra dar inicio a esta fase. Tras finalizar esta fase de manera exitosa, la compilacion puede considerarse terminada.

Generacion de Codigo Intermedio

La generacion de codigo intermedio toma lugar luego de haber compilado exitosamente 3 estructuras importantes: El AST, la Tabla de Simbolos y la lista de Clases.

Empleando estas 3 estructuras, se vuelve a recorrer el AST, pero esta vez se emplea la informacion extraida durante la primera fase para poder generar el codigo intermedio. El codigo intermedio es una representacion del programa escrito en lenguaje de alto nivel en codigo de bajo nivel en el que cambia totalmente el paradigma y no existen muchos de los conceptos que conocemos en alto nivel. Este codigo puede ser empleado posteriormente para traducir a codigo maquina, traducir a codigo Assembler o ser ejecutado directamente por un interprete.

Si la generacion del codigo intermedio es exitosa, sera redirigido automaticamente a la tab de ejecucion. En el text area de la izquierda se puede observar el codigo 3D generado, y a la derecha se puede observar la consola. Sobre la consola se encuentra el boton para comenzar la ejecucion del codigo.

Consideraciones:

- Si ocurre un error durante la fase de generacion de codigo intermedio, de todos modos sera redireccionado a la tab de ejecucion, sin embargo si se le notificara del error. Los detalles del/los error(es) estan contenidos en una tab aparte nombrada Errores.

- Si realiza un cambio al codigo fuente tras haber finalizado la primer etapa, pero antes de comenzar la segunda y selecciona “Finish compilation” el cambio efectuado no se vera reflejado. Para aplicar el cambio debera resetear el ciclo de compilacion de regreso al principio, puede hacer esto seleccionando el titulo que dice “Alto Nivel” Si lo hace, observara como cambia el texto del boton de regreso a la opcion de compilar.
- Para generar codigo intermedio de manera exitosa se debe saber donde comienza la ejecucion del programa. Por defecto, este buscara la clase Main y posteriormente el metodo: static void main(). Si alguno de estos no se encuentra en el programa fuente se informara al respecto y no sera posible continuar. Sin embargo, el suario es capaz de seleccionar cualquier metodo estatico valido como punto de inicio de ejecucion. Para seleccionar un metodo distintinto al default como punto de entrada, simplemente dirijase a la tab de clases y seleccione cualquier metodo estatico que NO tome ningun parametro. Al hacerlo, este metodo se coloreara verde, podra regresar a la tab de alto nivel y generar codigo intermedio empleando el metodo seleccionado como punto de entrada.

Tras haber seleccionado un punto de comienzo valido, y si no hay errores, el compilador generara el codigo intermedio correspondiente al programa y sera redirigido a la seccion de ejecucion.

En este punto puede considerarse la compilacion como terminada y exitosa. Para poder emplear el codigo intermedio generado, puede hacer uso del interprete de 3D adjunto en esta misma aplicacion.

El codigo 3D a ejecutar por el interprete puede ser el codigo generado por el compilador o puede ser escrito manualmente por el usuario en el textarea. El resultado se puede observar en la consola.

Para dar comienzo a la ejecucion, simplemente presione el boton verde “Iniciar” y la ejecucion dara comienzo.

La forma en que funciona el interprete de 3D puede ser customizada por el usuario para ajustarse a distintas necesidades. Para configurarlo se hace uso de directivas espceciales que pueden o no ser incluidas en el codigo fuente del 3D. De no ser incluidas se toman los valores por default.

Directivas soportadas por el Interprete de 3D

- **#UNCAP_HEAP**

  Por default el interprete designa un maximo numero de elementos para el Heap. Por default este maximo es de 150,000 celdas. Ya que este compilador no implementa un garbage collector, un programa relativamente grande podria sobrepasar este limite con cierta facilidad. Si ocurre esto, se le alertara al usuario sobre el Heap Overflow y se detendra la ejecucion. Sin embargo, tambien se le presentara la opcion para continuar la ejecucion. Si selecciona continuar la ejecucion, el tamanio maximo del heap es doblado y se ejecutaran el resto de instrucciones hasta alcanzar el nuevo limite o finalizar la ejecucion. Este limite se encuentra presente con el objetivo de proteger la maquina contra una alocacion de memoria excesivamente grande. Sin embargo, si el usuario esta seguro que su maquina puede alocar tanta memoria como se necesite sin ningun problema, puede emplear esta directiva para remover el limite del heap.

- **#UNCAP_HEAP_DISPLAY**

  Esta directiva tiene efecto unicamente si esta empleando el debugger. El debugger muestra en pantalla el valor y la posicion de todos los elementos del heap con forme se ejecuta el codigo. Sin embargo, este no muestra todo el heap. Este display muestra unicamente un segmento: 0 – 2000. Cualquier indice mayor a 2000 no podra ser visualizado, ya que requeriria renderizar demasiado espacio en la pagina y muchas computadoras pueden verse sobrecargadas con facilidad. Sin embargo, si el usuario esta seguro que su maquina puede manejar una renderizacion de tantas celdas tenga el heap, puede hacer uso de esta directiva. Al alcanzar el display 2000, el debugger doblara el tamanio del display y seguira adelante. NO se recomienda nunca su uso.

- **#UNCAP_STACK_DISPLAY**

  Igual que la directiva #UNCAP_HEAP_DISPLAY excepto que esta aplica al Stack. El Stack generalmente no tiende a crecer demasiado y aun si lo hace, este eventualmente vuelve a disminuir. Por esta razon el interprete no maneja ningun cap para el Stack, sin embargo esto es diferente para el debugger cuando grafica el Stack display. El Stack display puede graficar valores del Stack entre los indices 0 – 1000. Generalmente esto es suficiente para la mayoria de programas, sin embargo, si desea remover este limite y esta seguro que su computador puede manejarlo, puede hacer uso de esta directiva. NO se recomienda nunca su uso.

- **#UNCAP_INSTRUCTION_EXECUTION**

  Con el objetivo de proteger su maquina contra loops infinitos o recursividad demasiado profunda, el interprete maneja un limite de instrucciones que pueden ser ejecutadas de corrido. Este limite es de 150,000 instrucciones. Alcanzar este limite podria ser el resultado de una recursion descrontolada, un loop infinito por accidente o tambien podria simplemente ser un programa grande. Sea cual sea el caso, se le alerta al usuario sobre el limite de instrucciones excedido y se detiene la ejecucion.

Sin embargo, si el usuario esta seguro que su programa no tiene ningun problema y simplemente 150,000 instrucciones no alcanzan para su ejecucion completa puede seleccionar la opcion “continuar” visible unicamente si llega a este limite. Al seleccionar esta opcion el interprete ejecuta todas las siguientes instrucciones hasta alcanzar nuevamente el limite o terminar la ejecucion del programa. Si el usuario esta seguro que su programa no tiene ningun problema y le gustaria remover las “pausas” que genera el interprete por seguridad, puede utilizar esta directiva. No es recomendable su uso.

- **#MAX_HEAP** num

  La directiva #MAX_HEAP es una alternativa mucho mas segura a su version UNCAP. Si desea emplear un heap mas grande que el default pero no le gustaria remover el limite totalmente puede emplear esta directiva para establecer manualmente el limite del heap. La plabra num en la directiva indica que puede ser reemplazada por cualquier numero.

- **#MAX_HEAP_DISPLAY** num

  Su uso es similar a su version UNCAP pero mucho mas segura. Esta directiva hace posible editar manualmente el limite del heap display.

- **#MAX_STACK_DISPLAY** num

  Igual al #MAX_HEAP_DISPLAY pero este limite aplica para para el Stack display.

- #MAX_INSTRUCTION_EXECUTION num

  Igual a su version UNCAP pero mucho mas segura. Esta directiva hace posible editar manualmente el numero maximo de instrucciones que se pueden ejecutar antes de pausar la ejecucion.

- **#HIDE_NATIVES**

  Una funcionalidad explicada a detalle mas adelante es el Stack Trace. En resumen el Stack Trace muestra todas las llamadas realizadas antes de encontrar una Runtime Exception y esto incluye tanto llamadas a metodos nativos del interprete 3D como llamadas a funciones declaradas por el usuario. Si el usuario considera que las llamadas a metodos nativos pueden confundir un poco al ser mostrados junto con las llamadas a metodos declarados por el usuario, puede usar esta directiva y el Stack Trace mostrara unicamente las llamadas a metodos declarados por del usuario.

- **#FORCE_ENTRY_PROC** ID

  Por default, para que el interprete o el debugger comienzen su ejecucion del codigo 3D el procedimiento `___MAIN___` es empleado como punto de entrada siempre. Sin embargo, si el codigo 3D fue escrito a mano, o proviene de alguna otra fuente que no es el Y_Compiler, lo mas seguro es que este procedimiento no exista o tenga alguna otra funcion. Para poder usar el interprete con codigo 3D de terceros simplemente especifique con esta directiva el nombre del procedimiento que le gustaria emplear como punto de inicio de ejecucion. La palabra ID en esta directiva indica que puede ser reemplazada por cualquier nombre de procedimiento valido.

- **#FORCE_ENTRY_POINT** num

  Si por alguna razon, el usuario desea comenzar la ejecucion en cierto numero de linea en vez del comienzo de un procedure, puede emplear esta directiva. Donde num puede ser reemplazado por algun numero de linea del codigo 3D. NO es recomendado su uso.

- **#ACCURACY** num

  Esta directiva hace posible editar manualmente la exactitud con que el compilador hace ciertas operaciones. Por ejemplo, un numero double para ser representado en String el compilador necesita saber cuantos decimales va a mostrar en pantalla. Este numero es decidido por la ACCURACY default del compilador. Otro uso tambien es en el calculo de raices. Las raices son calculadas empleando el metodo de Newton y la accuracy con que se ejecuta este metodo es dependiente de la accuracy del compilador. La accuracy que el compilador utiliza se expresa en multiplos de 10. 10 es la accuracy por default del compilador. Si desea usar una mayor accuracy puede incrementar este numero a 100, 1000, etc. Mientras mayor sea la accuracy mas decimales se mostraran en pantalla y el calculo de raices sera mas exacto. Por default, el compilador muestra 5 decimales (accuracy * 10,000) Si incrementa la accuracy a 100 mostrara un decimal mas y asi sucesivamente.

- **#MAX_CACHE** num
  El compilador emplea un segmento reservado del Stack nombrado Cache. Este segmento se emplea para el calculo de expresiones. Este segmento se encuentra reservado en los indices 0-400 del Stack. Donde 400 es el limite del Cache. Si este limite llegara a ser excedido el programa entero fallaria completamente. Geralmente es casi imposible alcanzar este limite. Para alcanzar este limite se tendria que evaluar una expresion extremadamente grande o emplear recursividad muy profunda dentro de una sola expresion (como es el calculo de un Fibonacci muy grande por ejemplo) Aun con esto 400 es generalmente mas que suficiente para el cache, pero si por alguna razon este limite llegara a ser excedido durante la ejecucion puede emplear esta directiva para cambiar manualmente el limite del cache.

Consideraciones:

El interprete emplea ciertas palabras reservadas: Stack, Heap, P, H y C. Si esta escribiendo codigo 3D a mano o proviene de alguna otra fuente, asegurese que estas palabras cumplan con su funcion predeterminada. Las funciones de estas palabras son:

- **Stack**
  Empleada para acceder al Stack. Tanto para obtener un valor (t1 = Stack[t]) O para escribir un valor (Stack[t1] = 5).
- **Heap**
  Empleado para acceder al Heap.
- **P**
  P es empleado como el puntero del Stack y maneja su posicion actual.
- **H**
  H es empleado como el puntero del Heap y maneja su posicion y tamanio actuales. Si H llegara a ser mayor al limite del Heap se lanza una Heap Overflow exception.

- **C**
  C es empleado como puntero del Cache, si llegara a ser menor que 0 o mayor al limite del Cache (por default 400) Se lanza un Cache Underflow exception o un Stack Overflow exception segun sea el caso. Si su codigo no utiliza cache en lo absoluto favor de NO emplear C en ninguna parte del codigo.

El resto de palabras pueden ser empleadas como nombres de temporales sin ningun problema. Siempre que no sean palabras reservadas de la sintaxis del codigo (goto, if, proc, exit, write, read, print,input)
La sintaxis detallada del codigo 3D puede ser encontrada en otro documento.

La sintaxis del codigo 3D es case insensitive, es decir que Stack es lo mismo que STACK por ejemplo.

Puede utilizar todas las directivas en conjunto que desee y no hay ningun orden determinado sobre el orden que en que estas deben venir. Sin embargo, es una buena practica establecerlas al principio del codigo.

DEPURADOR

El depurador de 3D cumple con una funcion bastante similar el interprete y utilliza todas las directivas que el interprete utiliza. La diferencia es que provee varios metodos distintos para la ejecucion del 3D a diferencia del interprete que solo tiene la opcion “iniciar”. El depurador muestra en todo momento el valor de los temporales y el contenido del Stack y el Heap; siempre que sus indices no sobrepasen su maximo.

Las opciones que presenta el depurador son:

- **Iniciar**

  Su funcion es exactamente igual a la funcion del interprete con el mismo nombre: Ejecuta todas las instrucciones sin detenerse hasta terminar la ejecucion o sobre pasar el limite de instrucciones. La diferencia es que todos los displays son actualizados con forme se ejecute el codigo. Asi que al finalizar la ejecucion puede observar como terminaron las diferentes estructuras.

- **Siguiente Linea**

  Ejecuta unicamente la siguiente linea en la lista de instrucciones, muestra su resultado y se detiene.

- **Saltar**

  Su funcion es la misma que Siguiente Linea, excepto cuando la instruccion es un Call. Ya que Siguiente Linea entra al call y ejecuta la siguiente instruccion del procedimiento llamado y se detiene. Saltar por el contrario, al momento de ejecutar un Call efectua todas las instrucciones del procedimiento llamado sin detenerse hasta regresar del procedimiento.

- **Siguiente Break Point**

  El depurador presenta la opcion de establecer breakpoints sobre el codigo 3D. Para establecer un breakpoint simplemente seleccione alguna linea del editor y automaticamente se registrara el breakpoint. Esta funcionalidad es muy sensible, el simple hecho de navegar sobre el codigo 3D con las flechas del teclado anade breakpoints automaticamente por cada linea navegada. Para desabilitar esta sensibilidad y anadir breakpoints uncicamente cuando se este totalmente seguro que se desea anadir, puede dar click en el titulo “Depurar” literalmente a la palabra. Al hacerlo se desactivara la sensibilidad a breakpoints y podra navegar sobre el codigo 3D sin anadir breakpoints cada vez que se mueva sobre el editor. Cuando este listo para agregar un breakpoint puede seleccionar “Depurar” nuevamente para volver a re-habilitar la sensibilidad del editor a los breakpoints. En resumen la palabra “Depurar” produce un toggle de opciones entre habilitar y desabilitar la sensibilidad a breakpoints.

  Si selecciona un breakpoint especifico de la lista de breakpoints puede removerlo de la lista, tambien puede seleccionar “Clear All” para remover todos los breakpoints ingresados hasta el momento.

  Tras haber establecido todos los breakpoints de interes, la opcion “Siguiente Break Point” ejecutara todas las instrucciones siguientes sin detenerse hasta encontrar algun Breakpoint o finalizar la ejecucion.

- **Continuar**

  Esta opcion reanuda la ejecucion y ejecuta todas las instrucciones siguientes sin detenerse hasta finalizar la ejecucion.

- **Detener**

  Esta opcion detiene el flujo de ejecucion. Si se selecciona alguna de las opciones anteriores tras haber seleccionado esta opcion, estas se ejecutaran desde el principio.

Funciones nativas del Interprete

El codigo 3D es bastante universal y consistente en tanto a su sintaxis principal y a la idea fundamental de que unicamente se puede acceder a 3 posiciones de memoria al mismo tiempo. Sin embargo, aparte de las instrucciones basicas del codigo, como lo son: call, goto, if, ifFalse, proc { … } establecer labels, temp = temp op temp, etc. Existen tambien funciones nativas del interprete cuya sintaxis y numero de funciones soportadas puede variar de acuerdo a la implementacion. A continuacion se listan todas las funciones nativas que soporta el interprete y que pueden ser empleadas como cualquier otra instruccion de 3 Direcciones.

- **print**

  La sintaxis de esta funcion es la siguiente: print(‘%c’,valor)

  Donde ‘%c’ indica el formato, puede ser reemplazado por ‘%e’ o ‘%d’ Las comillas simples son siempre requeridas, y la letra luego del porcentaje indica el formato. %c indica que el valor sera interpretado como un codigo ASCII e imprimira el caracter correspondiente, %e indica un numero entero y %d indica un numero con punto decimal.

  Valor puede ser el nombre de algun temporal o puede ser algun numero escrito directamente.

- **exit**

  La sintaxis de esta funcion es la siguiente: exit (code)

  Donde code debe ser algun numero entero. Esta funcion detiene totalmente la ejecucion del codigo 3D. Esto debido a que esta funcion se utiliza para lanzar una exception en tiempo de ejecucion.

  Code indica que exception fue lanzada. Y actualmente puede ser:

  - 0 : Null Pointer Exception
  - 1 : Array Index out of bounds.
  - 2 : Downcast Exception.
  - 3 : Impossible to parse String to int. Invalid String.
  - 4 : Invalid Slice method arguments

- **write**

  La sintaxis de esta funcion es la siguiente : write ()

  Los parametros que esta funcion recibe son tomados del Cache, por esta razon no es necesarios expresarlos explicitamente en los parentesis. Debido a que esta aplicacion es puramente web, escribir un archivo a disco es imposible por motivos de seguridad. Por esta misma razon, es que esta aplicacion implementa un sistema de arhivos virtual puramente simbolico y bastante simple. Esta funcion recibe los parametros sobre la ruta y el contenido del archivo desde el Cache, por eso es que no es necesario enviarlos como parametros dentro del parentesis. El top del cache debe ser un puntero a un charArray que contiene el contenido del archivo.

Debajo debe encontrarse otro puntero a un charArray que indica el path del archivo. Ambos parametros son recibidos del Cache, se extraen sus valores del Heap y se escribe un archivo correspondiente. El archivo es cargado y creado en el sistema virtual de Archivos de la aplicacion. El usuario puede posteriormente guardarlos para descargar una copia de ellos.

- **read**

  La sintaxis de esta funcion es la siguiente : read ()

  Esta funcion no toma ningun parametro explicito en los parentesis porque los parametros son recibidos directamente del cache. Esta funcion espera unicamente un parametro del Cache:

  Un puntero a algun charArray indicando el path del archivo. Este parametro es recibido del Cache, se extrae su valor del heap, se obtiene el path del archivo y se busca en el sistema de archivos virtual de la aplicacion. Si no se encuentra lanza una exception. De lo contrario, compila el contenido del archivo a un nuevo charArray en el heap, sube al Cache la direccion del nuevo charArray y regresa.

- **input**

  La sintaxis de esta funcion es : input ()

  Esta funcion detiene la ejecucion del codigo. Para reanudar la ejecucion simplemente presione enter en la consola, al hacerlo el interprete obtendra el texto escrito en la ultima linea, lo compilara a una representacion de charArray en el heap, subira la direccion de la nueva cadena al cache y reanuda la ejecucion.

Lenguaje de Alto Nivel

El lenguaje de alto nivel aceptado por el Y_Compiler es denominado Yaba. Este lenguaje esta fuertemente influenciado por el lenguaje Java. Toda la sintaxis y funcionalidad del lenguaje es extraida directamete de Java, con algunas excepciones. Por lo tanto, podria decirse que si alguna entrada de Java compila en Java tambien deberia compilarse en el Y_Compiler, sin embargo hay algunas excepciones que el Y_Compiler implementa de forma diferente a Java. A continuacion se indican estas excepciones.

- **Paquetes (packages)**

  En Java una de las funcionalidades mas importantes y clave de todas es la implementacion de paquetes. En java se pueden agrupar clases en diferentes paquetes, y se puede indicar si la clase es public o protected. Si es protected solo puede ser instanciada y accedida desde otras clases dentro del mismo paquete. Ademas, si existen dos o mas clases con el mismo nombre dentro del mismo programa, esta ambiguedad puede ser resuelta empleando paquetes, donde el nombre simple de la clase se refiere a la clase en el paquete actual, si se desea utilizar la clase con el mismo nombre en otro paquete, debe ser completado el acceso a la clase como: nombre_paquete.nombre_clase

  Esta sintaxis puede aceptar una cadena de nombres, donde el final de la cadena representa el nombre de la clase y los otros miembros representan una ruta dentro del arbol de paquetes del programa. Esta funcionalidad NO esta soportada en lo absoluto en Y_Compiler. Por lo tanto, todas las clases podria decirse que estan en el mismo paquete. La sentencia import importa archivos no paquetes. Por esta misma razon, no pueden existir dos o mas clases con el mismo nombre en ningun lado del programa. De ser asi, se lanza una repeated class Exception. Aun si estan diferentes archivos y/o directorios.

- **Import**

  La sentencia import en Java importa paquetes completos o clases especificas dentro de los paquetes y utiliza una cadena de nombres indicando el path de la clase/paquete objetivo dentro del arbol de paquetes de la aplicacion. Y_Compiler como ya fue escrito mas arriba, no soporta paquetes. Por lo tanto la sentencia import importa archivos. La sintaxis es un tanto diferente:

  import “path” ;

  Donde path indica la direccion del archivo objetivo relativa al archivo que realiza el import. La direccion se resuelve en base al sistema de archivos virtual de la aplicacion. Por lo tanto el archivo tuvo que haber sido subido con anterioridad al sistema. Si el archivo no se encuentra, se reportara el error pero unicamente se ignorara la sentencia y se tratara de continuar la compilacion.

- **Inner classes**

  En Java es posible declarar una clase como private dentro de otra clase. De ser asi, esta clase puede ser instanciada unicamente dentro de la clase donde fue declarada. Esta funcionalidad NO esta soportada en lo absoluto por Y_Compiler. Intentar declarar clases anidadas sera reportado como error sintactico desde el principio.

Por esta misma razon, los modificadores de visibilidad aplicados a clases: public, protected o private no tienen ninguna funcionalidad real dentro del Y_Compiler. Sin embargo, para ayudar a varios programadores acostumbrados a declarar clases como public o protected (e incluso private en algunos casos) La sintaxis del modificador de visibilidad de clases es aceptado por el Y_Compiler, sin embargo estos modificadores aplicados a clases no tienen ningun efecto, todas las clases son manejadas como publicas en todo momento por el Y_Compiler. Si no se declara ningun modificador siempre se aplica public por default.

- **protected**

  Como fue escrito mas arriba, este modificador no tiene ningun uso aplicado a clases. Sin embargo, puede ser util al emplearlo como modificador de un miembro de clase (campo o metodo). En Java la funcionalidad de este modificador aplicado a miembros de clase funciona de tal manera que no es posible acceder a determinado miembro desde alguna clase que pertenece al mismo paquete. Debido a que los paquetes han sido totalmente removidos de esta leguaje, la visibilidad protected se maneja un tanto diferente:

  Un miembro declarado como protected puede ser accedido unicamente por la clase que define el miembro o alguna clase que extienda dicha clase. Es decir que uincamente la clase que lo declara o subclases de esta pueden acceder a dichos miembros.

- **private**

  Al igual que protected, este modificador tiene uso unicamente si se emplea como modificador de algun miembro de clase. Al definir un miembro como private no puede ser accedido por ninguna clase mas que la clase que define el miembro. Una subclase tampoco puede acceder al miembro, es mas las sub-clases nunca heredan los miembros private.

- **abstract**

  Se puede declarar un metodo como abstract y su funcionalidad es igual que en Java. No es posible declarar un metodo como abstract y private al mismo tiempo puesto que su implementacion nunca se podria completar.

- **final**

  Es posible emplear el modificador final tanto para metodos como campos. Su funcionalidad varia dependiendo del tipo de miembro a que es aplicado.

  Para campos, el modificador final implica que el campo es constante y por lo tanto se le puede asignar un valor una vez. Intentar modificar su valor resultara en un error de compilacion.

  Para metodos, un metodo declarado como final implica que el metodo no puede ser sobreescribido por una subclase. Si se intenta declarar un metodo con la misma signature que un metodo final de la super-clase, se lanzara un error de compilacion.

- **Uso de null como parametro**

  En Java, puede ser util en algunos casos enviar null como parametro a alguna funcion. Sin embargo, en Yaba no es posibe enviar directamente null como parametro, por ejemplo: doSomething(null); No es posible, ya que null podria representar cualquier objeto, y Y_Compiler no podria completar la signature del metodo especifico que se esta intentando llamar con null. Para poder obtener la misma funcionalidad, simplemente hay que castear null a la clase que espera el metodo recibir. Por ejemplo, en el ejemplo anterior, su version funcional en Yaba seria: doSomething((Something)null); Al indicar que ese null pertenece a la clase Something Yaba puede completar la signature del metodo y resolver la ambiguedad.

- **Diamond operator <>**

  Muchos programadores estaran familiarizados con el diamond operator de Java. Un ejemplo comun seria: Stack<Something> stack; El diamond operator en Java no esta reservado a las librerias, el usuario tambien puede hacer uso del diamond operator para implementar varias estructuras interesantes. Sin embargo, en Yaba el diamond operator no esta soportado de ninguna manera.

- **Interface**

  En java existe cierto concepto llamado Interface, cuyo funcionamiento es similar a una clase abstracta. La existencia de Interfaces implica la existencia y el uso de otra keyword: implements. Las interfaces son bastante famosas entre los programadores de Java y resultan sumamente utiles al implementar ciertas estructuras de datos interesantes. Sin embargo, las interfaces en su totalidad no estan soportadas por Yaba.

- **Exception, try and catch.**

  En Java es posible recuperarse de errores en tiempo de ejecucion, media vez se encuentren dentro de un bloque try-catch. La exception actua como un return especial que sale del programa a menos que sea atrapado por un bloque try-catch. De ser asi el conjunto de instrucciones en el catch se ejecuta tras haberse disparado la exception y recibe como parametro toda la informacion del evento que sucedio y su exception asociada. En java incluso es posible extender la clase Exception para enviar custom exceptions empleando la keyword throw. Todo este engine de Exceptions, throw, throws, try, catch, etc. No esta soportado en lo absoluto en Yaba. En caso de ocurrir una exception en tiempo de ejecucion es imposible recuparse del fallo y la aplicacion termina. Sin embargo se muestra informacion detallada sobre el Stack Trace en el momento en que fue lanzada la exception.

- **Uso de function calls del lado izquierdo de una asignacion.**

  En Java puede ser util en algunos casos muy especificos utilizar function calls del lado izquierdo de una asignacion, generalmente esta accion no tiene sentido porque indica que se intenta obtener una referencia de un valor puro al que aun no se le ha asignado ninguna direccion (ni de Stack ni de Heap). Sin embargo, si la funcion es un getter y el valor devuelto es algun objeto u arreglo entonces y solo entonces tendria sentido emplearlo del lado izquierdo siempre que el objetivo final de la asignacion no es el valor de la function call en si, sino algun campo o posicion del valor devuelto. Como por ejemplo: getMaestro().nombre = “nombre”; En este caso tendria sentido utilizar la function call del lado izquierdo. Sin embargo estos casos son muy especificos y es posible obtener el mismo resultado de una forma mas convencional: Maestro m = getMaestro(); m.nombre = “nombre”; Esta forma de realizar la asignacion es la forma convencional y es demas decir que esta totalmente soportada. Sin embargo el shortcut que se obtiene al realizar la function call directamente del lado izquierdo no esta soportada. En caso de intentarlo se lanzara una semantic exception: Impossible to retrieve reference from function call. Se recomienda tenerlo presente.

- **super**

  En java la keyword super puede utilizarse para hacer referencia a la clase padre de una sub-clase. Esta keyword esta implementada exitosamente en su totalidad en Yaba. La diferencia con super de Java original, es que en Java no es necesario utilizar super para llamar al constructor default de la clase padre. Por el contrario, en Yaba al instanciar una clase se llama unicamente su constructor nativo y posteriormente algun constructor definido por el usuario en caso de aplicar. Si se desea llamar al constructor de la clase padre en el constructor defecto de la clase, es necesario realizarlo explicitamente. Es decir: SubClass () { super(); } la keyword super tambien puede utilizarse para acceder a ciertos miembros de la clase padre que han sido sobreescritos por la sub-clase. Por ejemplo: super.doSomething(); ejecutara el metodo de la super-clase aun si la clase que realiza la llamada haya sobreescrito este metodo.

Clases Nativas y Metodos Nativos

En Java existen varias librerias “nativas” como lo son Java’s IO, Java’s utils, Java’s swing, etc.

Por el mismo hecho que los paquetes han sido removidos, esto implica que ninguna de las librerias nativas de Java estan implementadas o soportadas de ninguna manera.

Ademas de las librerias nativas, en Java tambien existen varias clases completamente nativas (No se necesita importarlas de ningun paquete) algunas son Object, String, System, etc. El Y_Compiler implementa unicamente las cases Object y String. Todas las demas clases como System no son implementadas. A continuacion se describe como fueron implementadas las clases nativas del Y_Compiler.

- **Object**
  - **String getClass()**

    Devuelve un String con el nombre de la clase del objeto
  - **boolean equals(Object)**

    Devuelve true/false si ambos objetos son iguales. Sin embargo, criterio que utiliza para verificar si son iguales su direccion de memoria. Si se desea alguna funcionalidad diferente se puede sobreescribir. La clase String por default sobreescribe este metodo y devuelve true/false si todos los caracters en ambos Strings son iguales.
  - **String toString()**

    Este metodo devuelve una representacion en forma de String del objeto. Esta representacion unicamente indica que es una instancia, su clase y su direccion de memoria. Este metodo puede ser sobreescrito por cualquier clase para darle una mejor funcionalidad mas especifica.
- **String**
  - **protected char[] char_array**

    Este campo almacena el charArray que representa el String.
  - **int length()**

    Este metodo retorna el tamanio actual del String.
  - **char[] toCharArray()**

    Este metodo retorna el charArray que representa el String.
  - **String toUpperCase()**

    Este metodo retorna el mismo String pero todas las minusculas se cambian por mayusculas. Este String es un nuevo String, no modifica el original.

- **String toLowerCase()**

  Este metodo funciona igual que toUpperCase() pero al reves, todas las mayusculas se cambian por minusculas.

- **Array**

  Un array en el Y_Compiler es un hibrido entre una Clase y un tipo puro. Un Array en el Y_Compiler no posee un constructor nativo, y tampoco esta registrado como Class dentro de la lista de clases. Por lo tanto un intento como: Array a = new Array(); Resultara en error a menos que el usuario implemente su propia clase Array. Sin embargo, es hibrida porque si implementa campos y metodos nativos de todos las arreglos. A continuacion se describen los campos y metodos que soportan todos los Arrays.

  - **int length**

    Un campo publico que contiene la longitud del array.

  - **boolean isNull()**

    Esta funcion devuelve true si el arreglo es nulo o false de lo contrario. Esta funcion es necesaria ya que una expresion del tipo: array == null es un error de semantica. Si se necesita verificar si un arreglo es nulo, se puede emplear este metodo NO hacer la verificacion directa como es el caso de los Objetos.

  - **int indexOf (int v)**
  - **int indexOf (char v)**
  - **int indexOf (double v)**

    Estas funciones devuelven el index del elemento buscado dentro del arreglo. Si el index no se encuentra se devuelve -1.

  - **?[] copy()**

    Esta funcion genera un nuevo arreglo cuyo contenido es una copia del contenido del array que llama la funcion. El tipo del arreglo retornado es el mismo el del arreglo que realiza la llamada. Esta funcion realiza una copia lineal, es decir que si se emplea con arreglos multidimensionales no va a realizar una copia recursiva. Sino simplemente un array con las mismas referencias que el array original. No se recomienda su uso con arreglos tipos Objeto o Array.

  - **?[] slice (int lowerLimit, int upperLimit)**

    Esta funcion devuelve un nuevo arreglo que contiene una copia de los elementos del arreglo original que realiza la llamada entre los indices lowerLimit (inclusive) y upperLimit (exclusive). Si los limites enviados no son validos (upperLimit > array.length, lowerLimit < 0, etc) Se lanza una exception en tiempo de ejecucion y es imposible recuperarse. Se recomienda ser cuidadoso con su uso.

- **String toString()**

  Esta funcion envuelve el array enviado dentro de una clase String y lo utiliza como su campo char_array. Si el caller de esta funcion no es un arreglo de caracteres se lanza una excepcion.

Funciones Nativas

Ya que varias clases nativas como System, Integer, Double, etc. No estan implementadas en el Y_Compiler las funciones de cierta utilidad general han sido implementadas como funciones estaticas que no pertenecen a ninguna clase. Por lo tanto para llamarlas se puede hacer uso del nombre de la funcion sin ningun otro contexto. Estas funciones toman precedencia sobre cualquier otra funcion con el mismo nombre de existir alguna. A continuacion se detallan las funciones nativas implementadas por el Y_Compiler.

- **void print(String message)**

  Esta funcion imprime en consola el mensaje enviado en la linea actual de la consola.

- **void println(String message)**

  Esta funcion imprime el mensaje en consola en la linea actual de la consola y agrega un salto de linea al final.

- **void write_file (String path, String content)**

  Esta funcion escribe un archivo en el sitema virtual de archivos de la aplicacion usando el path como nombre y content como el contenido del archivo.

- **String read_file (String path)**

  Esta funcion lee un archivo del sistema virtual de archivos de la aplicacion cuyo path coincida con el parametro y devuelve un String con el contenido. El path debe existir, de lo contrario se lanza una runtime exception.

- **String input ()**

  Esta funcion detiene la ejecucion para que el usuario puede ingresar texto en la consola. Al presionar enter se toma el texto en la ultima linea de la consola y se devuelve en un String.

- **String str (? value)**

  Esta funcion puede tomar cualquier tipo como parametro y hace todo lo posible para castearlo a un String. Su funcion es exactamente que la del operador de suma (+) aplicado a String y algun otro tipo. El unico tipo que es imposible representar como String son los Arrays.

- **int toInt (String value)**

  Debido a que la clase Integer no existe en Y_Compiler, para convertir un numero representado en un String a tipo int se puede emplear esta funcion. Note que el punto decimal es tomado como caracter

invalido, si el String no se puede convertir a int se lanza una exception en tiempo de ejecucion. Se recomienda ser cuidadoso con su uso.

- **double toDouble (String value)**

  Esta funcion devuelve un double extraido de su representacion en forma de String. Si la String no es valida por contener caracteres especiales (ademas del “-” o el “.”) Se lanza una exception en tiempo de ejecucion. Se recomienda ser cuidadoso con su uso.

- **char toChar (String c)**

  Esta funcion devuelve el primer caracter contenido en el String enviado. Es posible enviarle un String con length > 1, ya que unicamente el primer caracter sera regresado y el resto sera ignorado.

- **double pow (double base, double exponent)**

  Esta funcion eleva la base al exponente enviado. Ambos numeros pueden ser double o negativos de ser necesario. Si el exponente es double esto implica el calculo de raices de orden n. El compilador resuelve estas raices empleando el metodo de Newtown. La accuracy que el compilador utiliza es relativamente baja para mantener el tiempo de ejecucion optimo. Sin embargo si el usuario esta interesado en calcular raices mas exactas puede modificar la accuracy del compilador empleando directivas explicadas mas arriba en este documento.

- **? abs (? value)**

  Este metodo devuelve un int o un double segun se haya enviado como parametro y devuelve el valor absoluto de este.

Optimizacion de codigo intermedio

El codigo intermedio puede ser optimizado en base a ciertas reglas. La optimizacion de codigo que el Y_Compiler implementa esta basada en el metodo de mirilla. El metodo no es explicado en este documento. Para optimizar codigo sin importar si es el codigo generado por el compilador o codigo escrito por terceros, simplemente dirijase a la tab de Optimizacion. Notara que esta vacia. Seleccione el titulo “Reporte de Optimizacion” Al hacerlo dara inicio la optimizacion, tomando como entrada el codigo escrito en ese momento en la tab de ejecucion. Tras finalizar la optimizacion se mostrara una tabla con informacion detallada sobre las reducciones hechas y mas abajo se muestra la salida. La salida puede ser copiada y pegada de regreso a la tab de ejecucion o depuracion para ser ejecutada.

Sistema de archivos virtual

Varias funcionalidades utiles de un IDE o de un interprete estan fuertemente ligadas al manejo de archivos de la maquina cliente. Ya que el Y_Compiler es una aplicacion puramente web, no es posible acceder a disco directamente para lectura o escritura por motivos de seguridad. Sin embargo, para resolver esta restriccion el Y_Compiler implementa su propio sistema de archivos virtual puramente simbolico. Para poder importar o leer algun archivo este debe haber sido cargado al sistema virtual con anterioridad. Este sistema es puramente simbolico y simple. A continuacion se explica su uso:

En la tab principal, al momento de subir algun archivo o crear uno nuevo, este es agregado automaticamente al folder que se tenga seleccionado en ese momento. Por default, no existe ningun folder, y por lo tanto los archivos son agregados a la carpeta raiz del sistema. Si se desea agregarlos en carpetas diferentes, puede irse a la tab denominada “folders & classes’ Al top de esta tab se encuentra un text-field y a la derecha un boton nombrado “Add folder” Puede escribir cualquier nombre en el text-field y seleccionar “Add folder” para registrar la carpeta en el sistema en el folder que se tiene seleccionado en ese momento. Este hecho no selecciona la carpeta, simplemente la crea. Posteriormente puede seleccionar esta carpeta y se coloreara azul. Al tener seleccionada alguna carpeta y agregar un folder o algun archivo este automaticamente se agregara a la carpeta seleccionada. De esta forma puede crear carpetas anidadas. Seleccionando un folder recien creado antes de crear uno nuevo, agregara el nuevo folder a la carpeta seleccionada. Puede dejar seleccionada la carpeta y dirijirse a la tab principal para cargar los archivos a la carpeta seleccionada.

Opciones de visualizacion del AST

Por default, el AST no muestra todos los detalles asociados a sus tokens. Un token esta definido por un nombre, un texto asociado, un numero de linea, un numero de columna, una clase y un archivo. Todos estos detalles son escondidos en la grafica del AST para mantenerlo agradable a la vista. Unicamente se muestra el texto asociado a cada uno de sus tokens. Sin embargo, si el usuario esta interesado en ver estos detalles, puede seleccionar el titulo “AST” para hacer un toggle de opciones. Por default los detalles estan desactivados, sin embargo si selecciona el titulo, los detalles seran activados y la siguiente vez que compile la entrada, el AST mostrara todos los detalles asociados a sus tokens. Si desea esconderlos de nuevo, puede seleccionar el titulo nuevamente y re-compilar.