Compilador de COOL

Amanda Marrero Santos - C411
Manuel S. Fernández Arias - C411
Loraine Monteagudo García - C411

1 USO DEL COMPILADOR

Para el uso del compilador es necesario tener Python 3.7 o superior. Se hace uso de los paquetes `ply` para la generación del lexer y el parser y de `pytest` y `pytest-ordering` para la ejecución de los tests automáticos. Todos los paquetes mencionados pueden ser instalados usando pip ejecutando `pip install -r requirements.txt` en la raíz del proyecto. El archivo principal es `main.py`, ubicado en la carpeta `src`, y recibe como argumento 2 ficheros: uno de entrada que debe tener la extensión `.cl` con el código en COOL y otro de salida que será en donde se guardará el código MIPS generado. También se puede ejecutar el compilador haciendo uso del archivo `coolc.sh`, ubicado también en `src`, que espera como parámetro el programa en COOL. El fichero de MIPS generado por el compilador tendrá el mismo nombre que el fichero de entrada y será ubicado en la misma carpeta, pero se le añadirá la extensión `.mips`. Este fichero puede ser ejecutado por el simulador de MIPS `spim`.

2 ARQUITECTURA DEL COMPILADOR

Para el desarrollo del compilador se usó PLY, que es una implementación de Python pura del constructor de compilación lex/yacc. Incluye soporte al parser LALR(1) así como herramientas para el análisis léxico de validación de entrada y para el reporte de errores.
El proceso de compilación se desarrolla en 3 fases:

1. Análisis sintáctico: se trata de la comprobación del programa fuente hasta su representación en un árbol de derivación. Incluye desde la definición de la gramática hasta la construcción del lexer y el parser.
2. Análisis semántico: consiste en la revisión de las sentencias aceptadas en la fase anterior mediante la validación de que los predicados semánticos se cumplan.
3. Generación de código: después de la validación del programa fuente se genera el código intermedio para la posterior creación del código de máquina.

Cada una de estas fases se dividen en módulos independientes ubicados en `src`, que se integran en el archivo principal `main.py`

2.1 Análisis sintáctico

El análisis sintáctico se divide en 2 fases: en una se realiza el análisis léxico, con la construcción de un lexer, y en la otra se realiza el proceso de parsing, definiendo la gramática e implementado un parser para la construcción del Árbol de Sintaxis Abstracta (AST).

2.1.1 Análisis léxico

En esta fase se procesa el programa fuente de izquierda a derecha y se agrupan en componentes léxicos (*tokens*) que son secuencias de caracteres que tienen un significado. Todos los espacios en blanco, comentarios y demás información innecesaria se elimina del programa fuente. El lexer, por lo tanto, convierte una secuencia de caracteres (strings) en una secuencia de tokens. Después de determinar los caracteres especiales de COOL, especificados en su documentación, se deciden las propiedades necesarias para representar un token. Un token tiene un lexema, que no es más que el string que se hace corresponder al token y un tipo para agrupar los que tienen una característica similar. Este último puede ser igual o diferente al lexema: en las palabras reservadas sucede que su lexema es igual a su tipo, mientras que en los strings y en los números, estos son diferentes. Mientras que un token puede tener infinitos lexemas, los distintos tipos son predeterminados. Además, para el reporte de errores se guarda la fila y la columna de cada token.
La construcción del lexer se realiza mediante las herramientas de PLY. Para su construcción es necesario la definición de una variable `tokens` que es una lista con los distintos tipos de tokens. Después se especifican cuáles secuencias de caracteres le corresponderán a cada tipo de token, esto se especifica mediante expresiones regulares. Para cada tipo de token se definió una función mediante la convención de PLY de nombrar `t_tipo`, donde `tipo` es el tipo de token, y en el docstring la expresión regular que lo describe.

2.1.2 Parsing

El proceso de parsing consiste en analizar una secuencia de tokens y producir un árbol de derivación. Por lo tanto, se comprueba si lo obtenido en la fase anterior es sintácticamente correcto según la gramática del lenguaje.
El parser también se implementó mediante PLY, especificando la gramática y las acciones para cada producción. Para cada regla gramatical hay una función cuyo nombre empieza con `p_`. El docstring de la función contiene la forma de la producción, escrita en **EBNF** (Extended Backus-Naur Form). PLY usa los dos puntos (:) para separar la parte izquierda y la derecha de la producción gramatical. El símbolo del lado izquierdo de la primera función es considerado el símbolo inicial. El cuerpo de esa función contiene código que realiza la acción de esa producción.
En cada producción se construye un nodo del árbol de sintaxis abstracta, como se hace en Listing 2. El parámetro `p` de la función contiene los resultados de las acciones que se realizaron para parsear el lado derecho de la producción. Se puede indexar en `p` para acceder a estos resultados, empezando con `p[1]` para el primer símbolo de la parte derecha. Para especificar el resultado de la acción actual se accede a `p[0]`. Así, por ejemplo, en la producción `program : class_list` construimos el nodo `ProgramNode` conteniendo la lista de clases obtenida en `p[1]` y asignamos `ProgramNode` a `p[0]`

Listing 1: Función de una regla gramatical en PLY

```
def p_program(self, p):
    'program : class_list'
    p[0] = ProgramNode(p[1])
```

El procesador de parser de PLY procesa la gramática y genera un parser que usa el algoritmo de shift-reduce LALR(1), que es uno de los más usados en la actualidad. Aunque LALR(1) no puede manejar todas las gramáticas libres de contexto, la gramática de COOL usada fue refactorizada para ser procesada por LALR(1) sin errores.
La gramática especificada en el manual de COOL fue reconstruida para eliminar cualquier ambigüedad y teniendo en cuenta la precedencia de los operadores presentes.

Para la construcción del árbol de derivación se definieron cada uno de los nodos. El objetivo de dicho árbol es describir la forma en que la cadena es generada por la gramática. Intuitivamente, esta información nos permite "entender", sin ambigüedad, el significado de la cadena.
Además, se hizo uso de las herramientas de PLY para la recuperación de errores.

2.2 Análisis semántico

El objetivo del análisis semántico es validar que los predicados semánticos se cumplan. Para esto construye estructuras que permitan reunir y validar información sobre los tipos para la posterior fase de generación de código.
El árbol de derivación es una estructura conveniente para ser explorada. Por lo tanto, el procedimiento para validar los predicados semánticos es recorrer cada nodo. La mayoría de las reglas semánticas nos hablan sobre las definiciones y uso de las variables y funciones, por lo que se hace necesario acceder a un "scope", donde están definidas las funciones y variables que se usan en dicho nodo. Además se usa un "contexto" para definir los distintos tipos que se construyen a lo largo del programa. En el mismo recorrido en que se valida las reglas semánticas se construyen estas estructuras auxiliares.
Para realizar los recorridos en el árbol de derivación se hace uso del patrón visitor. Este patrón nos permite abstraer el concepto de procesamiento de un nodo. Se crean distintas clases implementando este patrón, haciendo cada una de ellas una pasada en el árbol para implementar varias funciones.
En la primera pasada solamente recolectamos todos los tipos definidos. A este visitor, `TypeCollector`, solamente le interesan los nodos `ProgramNode` y `ClassNode`. Su tarea consiste en crear el contexto y definir en este contexto todos los tipos que se encuentre.
Luego en `TypeBuilder` se construyen todo el contexto de métodos y atributos. En este caso nos interesan además los nodos `FuncDeclarationNode` y `AttrDeclarationNode`.
En el tercer recorrido se define el visitor `VarCollector` en el que se procede a la construcción de los scopes recolectando las variables definidas en el programa, teniendo en cuenta la visibilidad de cada uno de ellos. Requieren especial atención los atributos, cuya inicialización no puede depender un atributo que esté definido posteriormente a él; esto se logra llevando un índice de los atributos que han sido definidos hasta el momento y buscando en el scope los atributos a partir de dicho índice. Un problema similar ocurre con las expresiones `let` y `case`, para las cuales un nuevo scope es creado para que las variables definidas dentro de dichas expresiones sean solamente visibles dentro de dicho scope.
Por último, en `TypeChecker` se verifica la consistencia de tipos en todos los nodos del AST. Con el objetivo de detectar la mayor cantidad de errores, en cada corrida se toman acciones como definir `ErrorType` para expresar un tipo que presentó algún error semántico.

2.3 Generación de código

Con el objetivo de generar código de COOL de forma más sencilla se genera además un código intermedio, CIL, ya que el salto directamente desde COOL a MIPS es demasiado complejo.

2.3.1 Código Intermedio

Para la generación de código intermedio de COOL a MIPS se usó el lenguaje CIL, bastante similar al impartido en clases de Compilación. Solamente se añadieron algunas instrucciones, la mayoría de ellas con el objetivo de representar de forma más eficiente los métodos de bajo nivel que existían en los objetos built-in de COOL.
Para la generación de CIL se usa el patrón visitor, generando para cada nodo en COOL su conjunto de instrucciones equivalentes en el lenguaje intermedio. Además, para facilitar la gen-

eración de código de máquinas nos aseguramos de que todas las variables locales y las funciones tengan un nombre único a lo largo de todo el programa.
Una instrucción que requirió especial interés fue `case` que tiene la forma:

Listing 2: Forma de la expresión case

```
case <expr0> of
    <id1> : <type1> => <expr1>;
    . . .
    <idn> : <typen> => <exprn>;
esac
```

Donde `expr0` es evaluado y si su tipo dinámico es `C` entonces es seleccionada la rama `typek` con el tipo más especializado tal que `C` se conforme a `type`. Para facilitar esta selección, cada una de las ramas *i* de `case` fue ordenada de acuerdo a la profundidad que tiene `typei` con respecto al árbol de herencia que tiene como raíz a `Object`. Así, los nodos con una mayor profundidad son los más especializados. Luego, se tiene una instrucción especial en CIL para representar *conforms*, que determina si la clase `C` se conforma con `typek`. La primera rama *i* que sea seleccionada será la que cumplirá con los requerimientos, entonces se asignará `<expr0> <- <typei>`, esta se evalúa y retorna `<expri>`
La mayoría de los errores en runtime fueron comprobados generando el código CIL correspondiente, para esto se crea un nodo especial `ErrorNode`, que recibe el mensaje que se generará si se llega a dicho nodo. Por ejemplo, al crear el nodo `DivNode`, se chequea si el divisor es igual a cero, si lo es se crea un `ErrorNode` y en otro caso se ejecuta la división. El único error que no es analizado en la generación de código intermedio fue el de *Substring out of range*, causado al llamarse la función `substr` de `String`; este error es comprobado directamente cuando se genera MIPS.
Otra de las particularidades que se tuvo en cuenta para la generación de CIL fue la inicialización de los atributos, tanto los heredados como los propios de la clase. Cuando se crea una instancia de una clase se deben inicializar todos sus atributos con su expresión inicial, si tienen alguna; en otro caso se usa su expresión por defecto. Con el objetivo de lograr esto se creó para cada tipo un constructor, cuyo cuerpo consiste en un bloque, dándole un valor a cada uno de sus atributos. Este es llamado cada vez que se instancia un tipo.
Los tipos built-in definidos en COOL, (Object, IO, Int, String, Bool) son definidos desde CIL. Para crear el cuerpo de cada una de sus funciones se hizo necesario la creación de nodos nuevos en CIL. `ExitNode` y `CopyNode` para la implementación de `abort` y `copy` de `Object`. Las instrucciones `Read` y `Print` fueron sustituidas por `ReadInt`, `ReadString`, `PrintInt` y `PrintString` con el objetivo de implementar de forma más sencilla las funciones `in_int`, `in_string`, `out_int` y `out_string` de `IO` respectivamente, y debido a que en MIPS se requiere un trato diferenciado al tipo `int` y `string` cuando se hacen llamados al sistema. Para los métodos de `String`: `length`, `concat` y `substr` se crearon los nodos `LengthNode`, `ConcatNode` y `SubstringNode`. De esta forma, gran parte de las clases built-in se crean directamente en la generación de MIPS para lograr mayor eficiencia y debido a las particularidades de cada una de las funciones que requieren de un procesamiento especial.

2.3.2 Código de Máquina

Para generar el código de máquinas se usa una vez más el patrón visitor, recorriendo todos los nodos de CIL que fueron creados.
Entre las instrucciones se trata de manera especial la igualdad entre strings, que es realizada comparando carácter por carácter.

Register Allocation:
Uno de los principales problemas en la generación de código es decidir qué valores guardar en cada registro. Con el objetivo de hacer más eficiente dicha asignación se particionaron las instrucciones de cada una de las funciones del código intermedio en *Bloques básicos*, que son una secuencia maximal de instrucciones que cumplen que:

- El control de flujo solo puede entrar al bloque a través de la primera instrucción del bloque. Es decir, no hay saltos al medio del bloque.
- Se dejará el bloque sin saltos ni llamados de función, estas solo estarán posiblemente en la última instrucción del bloque.

En nuestro caso, en algunos bloques existirán saltos ya que algunas instrucciones de CIL, como `Concat`, `Length`, etc usan saltos cuando estos se generan en MIPS; pero como el código de estos nodos es creado directamente, el uso eficiente de los registros y de las instrucciones de MIPS fue garantizado dentro de dichos nodos.
Para formar dichos bloques primero se determina, dada una lista de instrucciones, aquellas que son *líderes*, que serán las primeras instrucciones de algún bloque básico. Las reglas para encontrar los líderes son:

1. La primera instrucción en el código intermedio es un líder.
2. Cualquier instrucción que siga inmediatamente a algún salto (incondicional o condicional) o alguna llamada a una función es un líder.
3. Cualquier instrucción que es el objetivo de algún salto incondicional o condicional es el líder.

Entonces, para cada líder, su bloque básico consiste en él mismo y en todas las instrucciones hasta, pero sin incluir, el siguiente líder o la última instrucción del código intermedio.
Para la asignación de las variables en los registros se usan 2 estructuras auxiliares:

Register descriptor: Rastrea el actual "contenido" de los registros. Es consultado cuando algún registro es necesitado. A la entrada de un bloque, se asume que todos los registros están sin usar, por lo que a la salida cada uno de estos se limpian, guardando su contenido en la memoria.

Address descriptor: Rastrea las localizaciones donde una variable local puede ser encontrada. Posibles localizaciones: registros y pila. Todas las variables locales son guardadas en la pila ya que en los registros solo estarán de forma temporal.

Además se calcula para cada instrucción una variable booleana `is_live`, que determina si en una instrucción es usada el valor de dicha variable (notar que de cambiarle el valor a la variable esta no se considera usada, su valor es redefinido). Se computa también su `next_use`, que contiene el índice de la instrución en donde fue usada después de esa instrucción. Para calcular dichas propiedades se recorre el bloque empezando por la última posición, en un principio todas las variables están "muertas" y su `next_use` no está definido (es null). Para la realización de este algoritmo usamos una estructura auxiliar, `SymbolTable`, que lleva cual es el último valor `next_use` y `is_live` de la variable. Para cada instrucción `x = y op z` que tiene como índice *i* se hace lo siguiente:

1. Se le asigna a la instrucción *i* la información encontrada actualmente en la `SymbolTable` de `x`, `y` y `z`.
2. En la `SymbolTable`, le damos a `x` el valor de "no vivo" y "no next use".
3. En la `SymbolTable`, le damos a `y` y a `z` el valor de "vivo" y al `next_use` de `y` y `z` el valor *i*

Cada instrucción tiene: `out`, `in1`, `in2` para determinar cuáles son los valores que necesitan registros. Así, si alguno de estos campos no es nulo y es ocupado por una variable se procede antes de procesamiento de dicha instrucción a asignarle registros a cada variable. Para la asignación de un registro se siguen las siguientes reglas:

1. Si la variable está en un registro ya, entonces se retorna ese registro.
2. Si hay un registro sin usar, retorna este.
3. Se elige entre los registros llenos aquel que contenga la variable que sea menos usada en las instrucciones restantes del bloque, salvándose primero en el stack el contenido del registro y se retorna dicho registro.

Si se tiene una expresión del estilo `out = in1 op in2`, entonces notar que el registro de `out` puede usar uno de los registros de `in1` y `in2`, si ninguno de estos operandos son usados en las instrucciones siguientes. Se usa el `next_use` de estas variables, comprobando que esta no haya sido usada en alguna instrucción siguiente, y si estas están "vivas", es decir, su valor actual es necesitado.
Se tienen disponibles para usarse casi todos los registros de MIPS, excepto: `$v0` que es usado como valor de retorno en todas las funciones y para realizar llamadas al sistema, `$ra` que es contiene la dirección de retorno de las funciones, `$a0` que también es usado para llamadas al sistema y `$t8` y `$t9` que son usados como registros auxiliares para la generación de código.

Representación de los Objetos:
Otra de las problemáticas en la generación de código fue la representación de los objetos en memoria. Se necesita guardar información referente al tipo de objeto, los métodos del objeto, sus atributos y alguna manera de representar su padre. Esto último es necesario para la implementación de `ConformsNode`.
A continuación se muestra un diagrama de como se representaron dichos objetos en memoria:

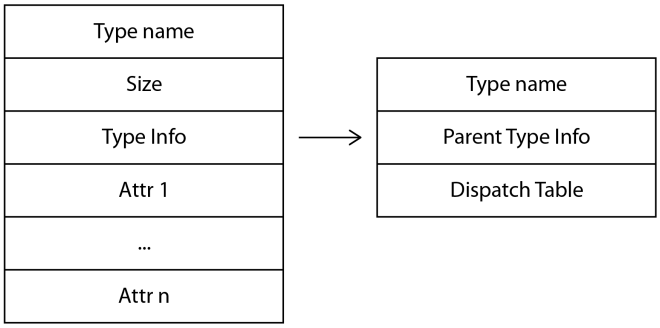

Figure 2.1: Representación de los objetos en memoria

Cuando se crean los tipos, para cada tipo se crea una tabla de `Type Info`, que guarda la información de los tipos que se presenta en la parte derecha de la Figura 2.1. `Type Name` es una referencia al nombre del tipo, y sirve como identificador del mismo, `Parent Type Info` guarda una referencia a la tabla de tipos del padre, mientras que `Dispatch Table` tiene una referencia a una tabla en la que están ubicadas los métodos del objeto tal y como se guardan en CIL: primero los métodos de los padres y luego los de ellos. Los métodos sobreescritos son guardados en la misma posición que ocuparían en el padre.
Luego, cuando se crea un objeto, se calcula para cada uno de ellos el tamaño que ocupará: esto será igual al total de atributos más los 3 campos adicionales usados para guardar información del tipo. En el primer campo se pone el nombre del tipo, luego el tamaño que este ocupa,

y después se busca la dirección del `Type Info` correspondiente al nuevo objeto que se crea. Después son ubicados los atributos primero los del padre y después los definidos por el objeto. Después, para buscar un atributo se calcula el offset de este atributo en su tipo estático, y a este se le sumará 3 porque ahí es en donde se empiezan a acceder a los atributos.
Un algoritmo similar es el que se usa para acceder a los métodos: se busca en la posición 3 de la tabla del objeto, donde se obtiene `Type Info` y desde allí en la posición 3 se accede a la dirección del `Dispatch Table`, a partir de la cual se busca el índice de la función llamada según la información que se tiene del tipo estático del objeto, luego, esta función es llamada. Notar que en caso de que su tipo estático no coincida con su tipo dinámico no importa, porque en la dirección buscada estará la verdadera dirección del método. Este puede ser diferente si, por ejemplo, es sobreescrito. A este proceso se le llama `Dynamic Dispatch`, representado como `DynamicCallNode` en contraposición con `StaticCallNode`. En este último se especifica la función a llamar, es decir, no es necesario buscar dinámicamente en la información del objeto, se accede al método de un tipo especificado.
Por otra parte, también requirió especial atención el nodo `ConformsNode`, usado para determinar si el tipo de una expresión se conforma con un `typen`. Para calcular esto se accede al `Type Info` del objeto y a partir de ahí, se realiza un ciclo por los `Parent Type Info` hasta encontrar alguno que sea igual a `typen`. La igualdad de tipos se comprueba accediendo a `Type Name`, como el mismo string es usado para representar este nombre, en el caso de que dicho string coincida con el nombre de `typen`, entonces se puede decir que los dos tipos son iguales.
Existen otros nodos que hacen uso de esta representación de objetos. `TypeOfNode`, por ejemplo, devuelve el primer campo del objeto correspondiente al nombre del tipo.

3 Problemas técnicos y teóricos

3.1 Comentarios y strings en el lexer

En el proceso de tokenización de una cadena dos de los elementos más complicados resultan ser los comentarios y los strings. Para su procesamiento se definieron 2 nuevos estados en Lexer: `comments:exclusive` y `strings:exclusive`.
Por definición la libreria PLY contiene el estado `initial` , donde se agrupan todas las reglas y tokens definidas para el procesamiento normal. En la definición de un nuevo estado, se definen nuevas reglas para el procesamiento especializado, es decir ,que ciertos tokens dentro de estos estados pueden ser analizados de manera diferente a que si estuvieran en el estado `initial`. De esta manera cuando se entra en en uno de los estados, todas las reglas que no correspondan al mismo son ignoradas.
Comentarios:
Tipo 1 (`--`): como es toda la línea no necesario entrar en un estado para procesarlo, con una expresión regular pudimos darle solución.
Tipo 2 (`(*...*)`): con el primer `(*` entramos en el estado de `commets`, así asumimos que a partir de ese momento todo es un comentario hasta que aparezca el `*)` que lo cierre. No necesariamente regresamos al estado inicial donde las reglas qué no son de comentarios no son ignoradas, sino que se tiene un control estricto de que comentario está cerrando dicho símbolo. Para esto se mantienen unos paréntesis balanceados con los inicios y finales de los comentarios y solo se regresa al estado inicial si el balance es igual a 0, lo que implica q ese `*)` cierra verdaderamente el inicio del comentario.
Strings:
En cuanto se analiza el carácter `"` desde el estado inicial del parser, se pasa al estado `strings`, donde definimos un conjunto de reglas especializadas para tokenizar de manera correcta el string. Entre dichas reglas se pueden destacar las siguientes:

- `T_strings_newline`: encargada de analizar `'\n'`, carácter muy importante en los strings que representa el cambio de línea.
- `T_strings_end`: Esta regla procesa el " dentro del estado `string`. Por defición si aparece dicho carácter en este estado significaría el fin del string en cuestión, a no ser esté antecedido por el carácter '\', en cuyo caso, se considera como parte del string.

3.2 Tipos por valor y por referencia

Una de las problemáticas en el desarrollo del compilador fue la representación de los objetos built-in, en especial: `Int`, `Bool` y `String` que en realidad no son representados como objetos; los dos primeros son tratados por valor mientras el último aunque es por referencia no tiene campos y es guardado en el la sección .data de MIPS. Sin embargo, todos son descendientes de `Object` que es la raíz de todas las clases en COOL y por lo tanto, poseen los métodos que están definidos en ese tipo. Con el objetivo de optimizar la memoria ocupada en el heap estos objetos no son guardados allí y no poseen la estructura de los objetos explicada anteriormente. Todas estas clases tienen en común que no es posible heredar de ellas, por lo tanto, una variable de tipo `Int`, `Bool` y `String` no es posible que sea otra cosa. Esto permite que no haya problemas a la hora de hacer dispatch: siempre es posible determinar la función a la que se va a llamar, por lo que las funciones de cada uno de estos tipos siempre se llaman estáticamente, determinando el nombre de la función. Los métodos que heredan de Object son redefinidos, ya que estos están implementados teniendo en cuenta la representación de objetos expuesta anteriormente que estos tipos no tienen.
A su vez, se requiere un trato especial en nodos como `TypeOfNode` porque estas variables no tienen en su primer campo el nombre de su tipo. Para llevar a cabo este análisis se guarda, para cada variable, el tipo de dirección a la que hace referencia. Se tienen como tipo de dirección: String, Bool, Int y por Referencia, este último para designar a los objetos representados mediante el modelo de objetos.
Por otro lado, la única manera de tratar algún objeto tipo `String`, `Bool` o `Int` es haciendo un casteo de este objeto de manera indirecta: retornando uno de estos valores cuando el tipo de retorno del método es `Object` o pasándolos como parámetro cuando el tipo de argumento es `Object` o de manera general asignándolos a una variable tipo `Object`. En estos casos se espera que el valor del objeto sea por referencia, así que se hace *Boxing*, llevándolos a una representación de objetos, almacenándolos en el heap. Para esto se crea un nodo especial en CIL `BoxingNode` a partir del cual se llevará a cabo la conversión en MIPS. Notar que el proceso contrario *Unboxing*, en el que se convierte un objeto por referencia a uno por valor, no es necesario implementarlo en COOL: el casteo directo no existe en este lenguaje y asignar un objeto de tipo `Object` a uno `Bool`, `Int` o `String` consiste un error de tipos.

3.3 Valor Void en MIPS

Otro de los problemas que se tuvo en cuenta fue la representación del valor nulo en MIPS. Para modelar este valor se creó un nuevo tipo, denominado `Void`, que lo único que contiene es como primer campo el nombre del tipo `Void`. Para comprobar que un valor es void, por lo tanto, simplemente se comprueba que el nombre del tipo dinámico de una variable sea igual a `Void`. Los errores de runtime ocasionados por intentar llamar estática o dinámicamente a una variable en void son controlados, así como la evaluación de una expresión en case, que de ser void ocasionaría un error al intentar buscar la información del tipo. Por lo que lo único necesario para la representación de void fue ese campo de nombre del tipo. Comprobar si una expresión es void consiste, por lo tanto, solamente en determinar si su tipo es `Void`.