# Reporte sobre el Compilador de Cool

## Uso del compilador

El `Compilador de Cool` implementado es un programa en lenguaje **Python**; programa que tiene como función principal dada la dirección de un archivo tipo `"file.cl"`, que contenga un programa de **Cool**, analizar dicho fichero reportar errores si tiene, en caso contrario generar un nuevo fichero en la misma dirección en esta ocasión terminado en `".mips"`. El programa se puede ejecutar con el comando:

```
python -m cool_compiler <file_dir>
```

En implementación del compilador se utilizo la librería **sly**, en especifico la version 0.4, la misma debe ser instalada previo a la ejecución del programa. Para instalar dicha librería se debe correr el comando:

```
pip install sly==0.4
```

## Arquitectura del compilador

El compilador es un único gran módulo `cool_compiler`, que contiene otros 6 módulos más:

### 1 - Módulo Cmp

Módulo aportado por los profesores de la asignatura, cuyas implementaciones ha servido de apoyo para las distintas clases practicas y proyectos a lo largo del curso. De este módulo se utilizan principalmente las clase `Type` y `Scope`, asi como los decoradores necesarios para implementar el patrón **visitor**. A dichas implementaciones a lo largo del desarrollo se le realizaron pequeñas modificaciones para ajustarlas, más aun, a las necesidades del desarrollo:

- __@visitor.result__: Nuevo decorador que se agregó a los decoradores del patrón visitor. Este decorador recibe la clase que sera el resultado de la función a la cual decora, una vez dicha función concluye sus valores de retorno son utilizados para instancias la clase en cuestión. Además pasa del nodo de entrada hacia el nuevo nodo resultante la información sobre a que linea y columna pertenecen dichos nodos en el código original

### 2 - Módulo Lexer

Módulo que comienza con el análisis de código de cool que se desea compilar, su función principal es convertir la cadena de caracteres en una cadena de token. Los token son la primera abstracción que se le aplica al código, estos son subcadenas que son lexicográficamente significativas y según este significado se les asigna un tipo (literal, keyword, type, identificador, number, string, etc). El desarrollo de este módulo se apoyo en la libreria **sly**; específicamente la clase **Lexer**, la cual brinda la facilidad de definir un

autómata que reconozca las subcadenas significativas para un lenguaje dad. Heredando de la clase **Lexer** y definiendo las propiedades *literals* y *tokens* se puede expresar la lista de expresiones regulares que el autómata debe reconocer, en este caso en particular se definieron de la siguiente manera:

```
...
tokens = {
    CLASS, INHERITS,
    IF, THEN, ELSE, FI,
    LET, IN,
    WHILE, LOOP, POOL,
    CASE, OF, ESAC,
    ISVOID, NEW, NOT,
    TRUE, FALSE,
    ID, NUMBER, TYPE,
    ARROW, LOGICAR, LESS_OR,
    STRING
}

literals = {
    '{','}', '@', '.', ',', ';',
    '=', '<', '~', '+', '-',
    '*', '/', '(', ')', ':',
}
...
```

La propiedad *tokens* de la clase que hereda de **Lexer**, por defecto interpreta que los tipos de token definidos en la lista se le deben asignar a toda ocurrencia literal de dicho tipo, obviando si los caracteres se encuentran en mayúscula o no. En caso en que el comportamiento por defecto no se ajuste a las necesidades del lenguaje, **sly** permite que se redefina la expresión regular asignada a los tipos de token definidos (por ejemplo en el caso de cool los token ID y STRING). La sintaxis para refefinir la expresión regular, es definiendo a la clase que hereda de **Lexer** una propiedad igual al tipo de token que quiere redefinir:

```
...
ID      = r'[a-zA-Z][a-zA-Z0-9_]*'
STRING  = r'\"'
...
```

Otra de las grandes ayudas que aporta **sly** que la facilidad de definir un método que se llamará al momento una ocurrencia de un tipo de token determinado. De esta manera se pueden definir nuevos comportamiento y personalizar el análisis del autómata. Para explotar esta característica de la librería basta con definir un método dentro de la clase con nombre igual a al tipo de token al que desea reaccionar

```
...
def STRING(self, token):
    lexer = CoolString(self)
...
```

Esta facilidad resultó muy útil para manejar el reconocimiento de los string y los comentarios de **Cool**. Para estos dos casos se definieron otras dos clase que hereda de **Lexer** para darle un análisis diferenciado. Estos fragmentos de código tiene un comportamientos similares al lenguaje `(ab)*` el cual no es regular, por lo cual no basta con un autómata finito determinista para reconocer todo el lenguaje. Para completar el reconocimiento de estos fragmentos, apoyados en **sly**, se implementaron autómatas con manipulación de memoria y asi poder contar la cantidad de ocurrencias de los distintos delimitadores

### 3 - Módulo Parser

El módulo Parser se encarga de chequear la consistencia sintáctica del código en cuestión. Dicho módulo también fue implementado con la ayuda de **sly**, específicamente con su clase **Parser**. Esta clase facilita la definición de gramáticas atributadas de manera extremadamente cómoda. Heredando de **Parser** se pueden definir métodos como atributo de cada producción de la gramática y mediante el decorador **@_** se especifica la producción a la que se le debe asignar dicho atributo. Ejemplo:

```
# cclass: no terminal y epsilon terminal
# en prod.cclass se encuentra el resultado del no terminal cclass
# en prod.epsilon se encuentra el terminal epsilon
@_("cclass epsilon")
def class_list(self, prod):
    return [prod.cclass]
```

Además **Parser** ofrece las herramientas para desambiguar en casos en que exista colisiones entre las producciones, como por ejemplo las producciones que tiene como cabecera el no terminal `expression`. Para que el parser sepa decidir cual de las producciones se debe seleccionar en cada escenario se necesita definir una prioridad entre las producciones. Para definir las precedencias de los distintos operadores, la clase **Parser** tiene la propiedad **precedence**, tupla de tuplas ordenadas de menor a mayor precedencia. Dicha propiedad inicialmente se encuentra vacia, y de ser necesario se pueden predefinir, en el caso particular del compilador de cool se predefinió de la manera siguiente:

```
...
precedence = (
    ('right', 'ARROW'),
    ('left','NOT'),
    ('nonassoc', '=','<','LESS_OR'),
    ('left', '+', '-'),
    ('left', '*', '/'),
    ('left', "ISVOID"),
    ('left', '~'),
    ('left', '@'),
    ('right', 'IN'),
    ('left', '.'),
)
...
```

Aprovechando la característica de que la gramática se encuentra recogida en una clase, se desarrollaron algunas herramientas para realizar la inversión de la dependencia entre la gramática y el ast, mediante el patrón **Factory**. Desde el módulo parser se definió un enum con los nombres de los nodos que la clase parser le pasará a la factoría de nodos, además de un decorador que enlace un método con el nombre del nodo que el mismo creará

### 4 - Módulo Type

Este módulo es un apoyo para el resto de los módulos principales, en el cual se definirán los tipos build-in, y se inyectarán mediante el decorador `build_in_type` al diccionario de la clase **CoolTypeBuildInManager** la cual cuenta con una implementación según el *patrón singleton*. Al momento en que cada uno de los tipos build-in son agregados al diccionario interno de manager de los mismo se crea una instancia y se ejecutan todas sus funciones mediante las herramientas de la librería de Python `inspect`. El resultado final de este módulo es una clase que contiene toda la información de los tipos build-in accesible a partir de los nombres de los mismos y una implementación bastante extensible de la definición de los mismo. Ejemplo:

```
@build_in_type
class String(Type):
    def __init__(self):
        super().__init__("String")  # Define el nombre del tipo
        type_body_def(self)

    @property
    def is_shield(self): # sobre escribe alguna de las
        return True      # propiedades de la clase Type

    def value(self):    # define un atributo
        self.define_attribute("value", self)

    def length(self):   # define una función
        self.define_method("length", [], Int())
    ....
```

### 5 - Módulo Semantic

La fase del análisis semántico del código en cuestión se encuentra conformado por dos recorridos sobre el **AST** resultante de la fase **Parser** además de la implementación de la factoría de *nodos* antes comentada que unido al proceso anterior origina el **árbol de sintaxis abstracta** correspondiente al código de entrada. Como el parser construye los distintos nodos, según la factoría, en profundidad, osea primero los hijos y luego el padre, entonces se decidió unir la factoría con el primer recorrido que se estudio en las clases de chequeo semántico, **Recolección de Tipos**, pues antes de construir el nodo **Program**, raíz del **AST de Cool**, ya se construyeron todos los nodos **Class** y por tanto ya se pueden tener localizados todos los nombres de los nuevos tipos definidos en el programa que se intenta compilar. Para la implementación de dicha factoría, se definieron algunos decoradores con los que se logran mapear cada método con la palabra clave del nodo que dicho método

construirá, logrando así en la clase **CoolFactory** un aspecto similar al *patrón visitor*, patrón de diseño de todas las implementaciones siguiente en el proceso de compilación.

```
@factory
class CoolFactory:
    @decored(Nodes.Program)
    @compose
    def program(self, class_list):
        return Program(class_list, self.global_names)

    @decored(Nodes.Class)
    @compose
    def cool_class(self, name, parent, features):
        self.global_names.append((name, self.lineno, self.index))
        return CoolClass(name,parent,features)
```

Siguiendo el paradigma **FP** cada uno de los procesos y análisis que se realizan a partir de esta etapa tendrán como resultado un nuevo **AST** cada cual con la información agregada del recorrido que lo origino, logrando asi procesos puros libres de efectos segundarías y estados internos, además de objetos inmutables. Esta metodología, analizar la salida de un procesos anterior y definir un nuevo **AST** com respuesta del análisis actual, lo comienza la factoría de nodos definiendo el primer **AST** sobre el que se trabajará

Siguiendo las pautas marcadas en las clases de **Compilación** una vez realizada la recolección de tipos, se hace necesario construir los mismo y así contar con una documentación de los tipos definidos en el programa para futuros procesamientos. Este recorrido es bastante simple ya que no es mas que la acumulación de información sobre los distintas funciones y atributos de cada uno de los tipos definidos en el nuevo programa. Además de chequear que todos los tipos a los que se les hace referencia de manera explicita; ya sea el tipo de los atributos, de los parámetros de las funciones, del retorno de las funciones, asignaciones `let`, o ramas del los `case`, estén definido y reconocidos por procesos anteriores. Genera como resultado un nuevo **AST**, donde en todo nodo en el que se hace referencias explicitas al los distintos tipos se guarda la toda la documentación recogida en dicha pasada sobre el tipo en cuestión

Se finaliza el chequo semántico con un último recorrido donde se validan todas las reglas y predicados con las que debe cumplir un codigo de **Cool** según la documentación. Entre los problemas más interesantes que se presentan en esta fase pueden ser el control y definición de los distintos `scope` y de los tipos estáticos de cada una de las expresiones, pues el resto no es más que la comprobación de una serie de predicados, según el nodo en cuestión y el resultado de los nodos de los que dicho nodo depende. Para la resolución de los distintos `scope`, se reutilizo la clase **Scope** de **cmp**, aportado por los profesores en el momento en que se impartieron los contenidos relacionados a este procesamiento. El tipo estático de cada expresión se guarda en los respectivos nodos de respuesta y su definición depende de la expresión en cuestión, pues nodos como las operaciones aritméticas o de comparación tiene un tipo estático previamente definidos por la documentación, Int y Bool respectivamente, mientras que otras como los `if` y los `case` se debe realizar un búsqueda entre los ancestros de los tipos estáticos de cada una de las ramas para encontrar el primer

ancestro común entre los tipos de cada rama. Una vez finalizadas todas las validaciones de los distintos predicados por los distintos nodos se genera un nuevo **AST** el cual sumando a la información anterior, los nodos de tipo **Expression** contiene además el tipo estático de la expresión

## 6 - Módulo Codegen

La fase final de la compilación, sabiendo que el código en cuestión es correcto tanto sintáctica como semánticamente, es la generación de un nuevo programa que ejecute las ordenes trasmitida en el programa de entrada, en el sistema subyacente. En este caso en particular el lenguaje seleccionado para dicho programa final fue **MIPS** y el proceso de su generación se encuentra dividido en dos por un lenguaje intermedio (CIL) para facilitar la traducción de **Cool** a **Mips**

### *Cool a CIL*

Este paso intermedio en la generación del programa final, es muy flexible y abierto, del lenguaje intermedio solo se generá un **AST** del mismo, con lo cual dicho lenguaje no debe cumplir con otras reglas sintácticas y semánticas, salvos las que entienda el receptor del **AST** resultante. Con lo cual en todo momento se le puede realizar todo tipo de cambios al lenguaje para facilitar la traducción ente **Cool** y **Mips**. El **CIL** resultante al termino de este trabajo, se basa en las ideas expuestas por los profesores de la asignatura, un lenguaje donde toda instrucción tiene máximo 3 operadores y en la mayoría de los casos uno de ellos es la variable a la que se le asignara el resultado de la operación. Siguiendo las ideas de nuestros profesores, el nodos **Program** del **AST** de **CIL** se encuentra dividido en tres grandes componentes:

- **Data**: Diccionario donde se almacena una serie de datos estáticos importantes para la ejecución del programa, principalmente los string literales definidos en el programa y otros predefinidos como `_____null_____`, `_____error_____` o `_____void_str_____`. Además en esta sección se almacena una información importante sobre los tipos del programa como una lista de enteros que representa la rama paterna del mismo (bajo el label `(type name)\_parents`), el nombre literal del mismo (bajo el label `(type name)\_name`) y la tabla virtual de funciones, diccionario donde se almacena el nombre de cada uno de los métodos del tipo y el nombre del padre más cercano que lo define
- **Type**: Lista de las distintas documentaciones de los tipos definidos en el programa, con su nombre, atributos y métodos, estos dos últimos ordenados en función de la profundidad en al que fueron definidos, primeros los atributos y funciones definidos por los padres y luego el el tipo en cuestión
- **Functions**: Lista de los nodos **Function** del **AST de CIL**, dichos nodos cuenta con el nombre de la función, lista de parámetros, lista de variables locales y lista de expresiones. Dicha lista contiene todas las funciones definidas a lo largo del programa, una funcione `new` por cada tipo definido (que se encarga de asignar los valores a los distintos atributos de la clase en cuestión) más una función **main** encargada de crear la nueva instancia de la clase **Main** y de llamar a la función **main de Main**

El recorrido que genera **CIL** en rasgos generales, se concentra en construir los nodos **Function**, almacenado la lista de comandos resultante de recorre la expresión de la función en cuestión, recorrido en el que se definirían las variables locales necesarias. Los nodos **Expression** de **CIL** se pueden dividir en *comandos* y *operaciones*, los *comandos* no tiene resultado, mientras que las *operaciones* si lo tiene. El resultado de evaluar la función `visit` en nodos de tipo **Expression** de **Cool**, es una lista de nodos **Expression** de **CIL**. Para el caso de la *operación* se define una palabra clave ‘’, palabra que se utiliza para que el padre del nodo en cuestión detecte que tiene que cambiarla el nombre de la variable local en la que se guardara el resultado final de la operación; además de funcionar como un pequeño auto test, pues si en el recorrido del padre de alguna expresión espera que la lista resultante termine con una operación con esta palabra clave y no es así, al internar remplazar dicho valor se genera una excepción especial para rápidamente detectar dicho error. A lo largo del recorrido por la expresión de cada función en particular se definen varias variables locales, algunas naturales y marcadas por el programa, como son las definiciones `let` o `case`, y otras artificiales debido a las necesidades de almacenar el resultado de operaciones intermedias, como por ejemplo las operaciones aritméticas que generan una variable local por cada operando. Otra de las palabras clave (_) que se definieron de la fase desarrollo de este recorrido tiene que ver con minimizar la cantidad de variables locales artificiales a generar, pues si el resultado de una operación se va a utilizar inmediatamente que se resuelve dicha operación no será necesario almacenar este resultado, ejemplo:

```
    @visitor.when(AST.IfThenElse)
    def visit(self, node: AST.IfThenElse, scope: Scope):
        expr_list = [ASTR.Comment(f'Evalua la condicion de un If')]
        # evaluá la condición del if
        expr_list += self.visit(node.condition, scope)
        # Cambia la palabra clave <value> por la palabra clave _
        expr_list[-1].set_value('_')
        expr_list.append(ASTR.Comment(f'Fin de la evaluacion de la
condicion de un IF'))

        #Genera label con nombre únicos en el nuevo programa CIL
        label_then = self.new_name(f'then_{self.currentFunc.name}',
self.label_list)
        label_fin = self.new_name(f'fin_{self.currentFunc.name}',
self.label_list)
        # Utiliza el resultado de la expresión antes resuelta
        expr_list.append(ASTR.IfGoTo('_', label_then))
        ...
```

En la implementación de este recorrido se reutilizó la clase **Scope** de **cmp** para manejar los distintos scope de las distintas variables locales. El scope no hace distinción entre parámetro de la función y variables locale, ya sean “naturales” o “artificiales”, dejando fuera los atributos de la clase a la que pertenece la función en cuestión. De esta manera se distinguen entre la asignación a una variable local y la asignación a atributos de la clase. En la resolución de los nodos **Assing** de **Cool** es un buen ejemplo de todo lo sobre esta recorrido:

```
    @visitor.when(AST.Assing)
    def visit(self, node: AST.Assing, scope: Scope):
        exp_list = [ASTR.Comment(f'Asignando un nuevo valor a la
variable {node.id}')]
        # evaluá la expresion
        exp_list += self.visit(node.expr, scope)
        # Cambia la palabra clave <value> por la palabra clave _
        exp_list[-1].set_value('_')
        # Busca si la variable es local o es un atributo de self
        var = scope.find_variable(node.id)
        if not var is None:
            # Caso en que la variable es local y se le asigna el valor
de la expresion anterior
            exp_list.append(ASTR.Assign(var.type, '_'))
            exp_list.append(ASTR.Assign(super_value, var.type))
        else:
            # Caso en que la variable es un atributo de self y se le
asigna el valor de la expresion anterior
            exp_list.append(ASTR.SetAttr('self',
self.currentType.attr[node.id], '_'))
            exp_list.append(ASTR.GetAttr(super_value , 'self',
self.currentType.attr[node.id]))

        return exp_list # retorna una lista de Expression de CIL
```

### *CIL a MIPS*

La genreación código **Mips** parte de una librería escrita en el propio lenguaje, la cual contendrá gran parte de las herramientas en las que se apoyará el código final, pues es una buena idea orientar hacia el paradigma funcional las implementaciones de un lenguaje de bajo nivel que brinda pocas abstraciones y contar con una lista de funcionalidades básicas para la ejecución de todo programa de **Cool** que ya se encuentren debidamente testeado y aporte gran seguridad al momento de encontrar bugs. La librería consta de tres archivos (*int.mips*, *str.mips*, *bool.mips*), en los cuales se impelmentan funciones básicas como las operaciones aritmeticas y de comparaciones entre los enteros, la comunicación con la entrado y la salida estandar tanto para enteros como para strings, las funciones de *lenght*, *concat*, *substr* y *eq* de strings, además de las funciones que crean los distintos objetos **Int**, **Bool** y **String**. A lo largo de las distintas implementaciones en **Mips** se adoptó una pequeña leyenda de registro.

- *$s6* : Dirección de memoria donde se encuentra la instancia de **self**
- *$s4* : Registro para guardar las informacionde que el recorrido **Cool a Cil** marca como temporales con la palabra clave _
- *todos los registros $a*: Son los registros en los que la libreria implementada espera los distintos parametros

Partiendo de esta base, la implementación del recorrido para generar el código **Mips** como punto final del proceso de compilación queda encargado de traducir cada nodo de **CIL** a la lista de instrucciones de **Mips**, donde la mayoría realizarán llamados la las funciones

previamente definidas, y resolver otros problemas importantes que no se puden incluir en una libreria previamente definida, como el manejo de la memoria tanto en el *heap* como el *stack*, la localización de varibles locales, parametros de funciones y atributos de instancias.

En el código **Mips** final se necesita definir de alguna manera los tipos, conocer su lista de funciones y guardar los valores de los atributos de las distintas instancias del mismo. En este caso en la sección **.data** de el código que se genera finalmenalmente tendremos varias informaciones relacionadas con los tipos y cada instancia hará referecincia a la dirección de memoria en donde se encuentra toda esta información. Como en **Cool** todos los tipos heredan de **Object** e incluso tipos como **Int** y **Bool** que no tiene métodos ni atributos propiamente definidos, igual necesita de una estructura de objetos para acceder a los métodos de **Object**. Por tanto en el código de **Mips** resultante todos los valores reales, osea los numeros literales del código, los valores que se van computando, los booleano que se cambian por ceros o unos, y las direcciones de memoria donde se encuetran los strings, se encuentran en el *heap* referenciados por una direccion de memoria que apunta al principio de la instancia, dicha dirección se encontrará ya sea en el *stack* como varaible local o en *$s4_ como variable temporal. Luego todas las variables locales y temporales, del _stack_ y _$s4* son direcciones de memoria del *heap* donde se encuentran todas las intancias, las cuales hacen referencias a todos sus atributos definidos en el programa más uno adiciona **type_name** que es la direccion de memoria donde podemos ver toda la información del tipo en cuestión y una propiedad extra más **value** para los tipos básicos **Int**, valor nūmerico, **Bool**, cero o uno, **String**, dirección de memoria de string. Ejemplo:

```
.data
Main_parents: .word 1, 6, 14, 15, 0,    # 1-Object, 6-IO, 14-Parse,
15-Main, 0-fin
Main_Name: .asciiz "Main"
Main: .word Main_Name, 4, Main_parents, Object_abort, ...,
IO_in_int, ..., Parse_read_input, ..., Main_main,

.text
#Allocate a una class Main
#atributo type_name en puntero + 0
#atributo boolop en puntero + 4
#atributo rest en puntero + 8
#atributo g en puntero + 12
li $a0, 16           # Reservan memoria de (cantidad de attr + 1) * 4
li $v0, 9
syscall
sw $v0, 4($sp)      #Guardando en la variable local self puntero de la
instancia de la clase Main
#Assignacion de la insformacion de tipo a Main
la $t0, Main
move $s4, $t0       # Guarda la información del tipo en la variable
temporal _
lw $t0, 4($sp)      # Leer la instancia de la clase de la pila
move $t1, $s4
sw $t1, 0($t0)      # Setea la propiedad type_name con el valor de _
lw $t0, 4($sp)      # Lee el valor de la var self
```

```
addi $sp, $sp, -4
sw $t0, 0($sp)      # Push a la pila con self
jal new_ctr_Main    # LLama a la funcion que resuelve el valor inicial
de los atributos
```

Otro punto importante al momento generar el código final, es el debido mapeo de los parametros y veriables locales con su dirección en el *stack*. Para llevar este conteo, a meddida que se va generando el código, se implemento una clase **Stack** con los métodos:

- *push*: Recibe el nombre de una variable local, la agrega a su cola interna y devuelve la lista de comando de **Mips** para pushear en la pila
- *read*: Recibe un registro y el nombre de una variable local, devuelve la lista de comando para copiar el valor de la variable de la pila al registro
- *write*: Recibe un registro y el nombre de una variable local, devuelve la lista de comando para copiar del registro a la direccion de la pila que refiere a la variable
- *close*: No recibe parametros, y devuelve la lista de instrucciones para limpiar la pila entre llamados de funciones

Apoyado en los conceptos anteriores el recorrido **visitor CIL a Mips** se mueve según los nombres que van apareciendo en una lista global del recorrido, dicha lista comienza con la funcion `main` y va creciendo con cada nodo `SimpleCall`, `Call` o `VCall` que agregan a la lista el nombre de la función a la que se va a llamar. De esta manera solo se genera las funciones que tiene alguna posibilidad de ser ejecutadas

### *Generación de la Expresion Case - CheckType*

Para la resolució*n de las expresiones `case` en el módulo de generación de código se creó un pipeline exprecífico que cuenta con un nuevo nodo en el **AST** de **CIL**, **CheckType**, y de una función en la libreria de funciones basicas de **Mips**. El nodo **CheckType** se crea con tres parametros al momento de su creación, el nombre de la variable local donde se guardara el valor booleano, la variable que tiene la información de tipo de la expresión del `case` y el valor numerico que se le asigan al tipo de la rama en cuestión. Por tanto la expresión `case` se traduce en un nodo **CheckType** por cada una de sus ramas, los cuales evaluan los distintos predicados para decidir cual de ellas se debe ejecutar. Previo a la generación del **AST** de **CIL** de esta expresión se ordena todas las ramas en el orden topológico de la relación hereditaria de los tipos. Luego en la libreria de funciones basicas de **Mips** se implemento la funcion `__check_type__` que recive la dirección de memoria de una lista de enterios, osea el label (type_name)_parents, y un entero, devuelve un booleano que es true y el entero se encuenta en la lista, false en caso contrario. De esta manera la expresión `case` se reduce al chequeo secuencial de el predicado `CheckType` y la ejecución de la rama donde el predicado es afirmativo.