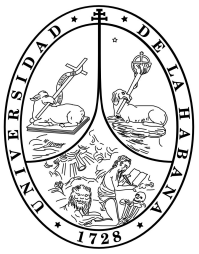


Informe Escrito de Complementos de Compilación

Juan José López Martínez j.lopez2@estudiantes.matcom.uh.cu
Grupo C411

Juan Carlos Esquivel Lamis j.esquivel@estudiantes.matcom.uh.cu
Grupo C411

Ariel Plasencia Díaz a.plasencia@estudiantes.matcom.uh.cu
Grupo C412

Índice

1 Orientación

La evaluación de la asignatura Complementos de Compilación, inscrita en el programa del 4to año de la Licenciatura en Ciencia de la Computación de la Facultad de Matemática y Computación de la Universidad de La Habana, consiste este curso en la implementación de un compilador completamente funcional para el lenguaje COOL.

COOL (Classroom Object-Oriented Language) es un pequeño lenguaje que puede ser implementado con un esfuerzo razonable en un semestre del curso. Aun así, COOL mantiene muchas de las características de los lenguajes de programación modernos, incluyendo orientación a objetos, tipado estático y manejo automático de memoria.

2 Plan de Trabajo

No.	Etapas	Fecha de Entrega
1	*Análisis Lexicográfico* (*lexing*)	23 *de febrero del* 2021
2	*Análisis Sintáctico* (*parsing*)	20 *de marzo del* 2021
3	*Análisis Semántico* (*semantics*)	7 *de mayo del* 2021
4	*Generación de Código* (*code_generation*)	25 *de febrero del* 2022
5	*Interfaz Visual* (*pyqt*)	1 *de agosto del* 2021

3 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, de *pytest* y *pytest-ordering* para la ejecución de los tests automáticos y de pyqt5 para el diseño de la interfaz gráfica. Todos los paquetes mencionados con anterioridad 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 tanto argumentos posicionales como opcionales:

1. *input_file* : Argumento posicional que indica la ruta del fichero de entrada con el código COOL. Dicho archivo debe tener la extensión *.cl*.
2. *output_file* : Argumento posicional que indica la ruta del fichero de salida en donde se guardará el código MIPS generado. Dicho archivo debe tener la extension *.mips*.
3. *help* : Muestra una pequeña e importante documentación sobre estos comandos.
4. *debug* : Argumento opcional booleano que especifica si queremos guardar los archivos intermedios de las fases transitadas por el compilador.

También se puede ejecutar el compilador haciendo uso del archivo *coolc.sh*, ubicado también en *src*, que espera los dos primeros parámetros anteriormente mencionados. 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*. Esperamos que este fichero pueda ser ejecutado por spim, el simulador de MIPS.

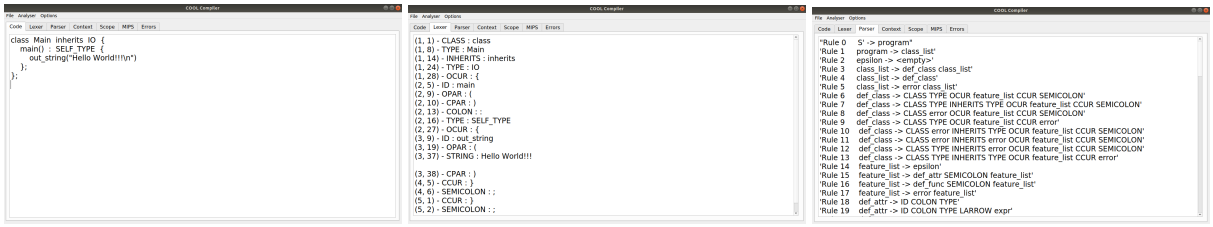
Además, añadimos un *makefile*, con las siguientes etiquetas y funcionalidades:

- *main* : Compila el proyecto.
- *visual* : Ejecuta la aplicación de escritorio COOL Compiler.
- *info* : Muestra una pequeña descripción del proyecto.
- *version* : Imprime la versión actual del proyecto.
- *install* : Instala todas las dependencias necesarias para ejecutar el proyecto.
- *clean* : Elimina temporalmente algunos archivos.
- *test* : Corre los casos de prueba de cada etapa.
- *help* : Muestra esta ayuda.

4 Interfaz Visual

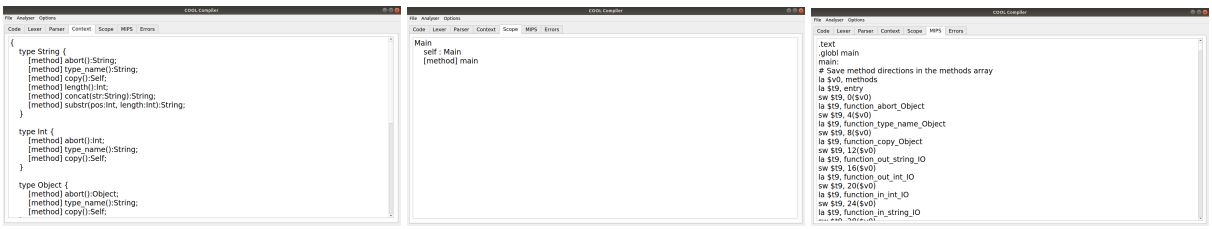
La interfaz visual fue llevada a cabo en pyqt5. Esta librería es un binding de la biblioteca gráfica Qt para el lenguaje de programación python. Está disponible para *Windows*, *GNU/Linux* y *Mac OS X* bajo diferentes licencias, por lo que nuestro proyecto con las dependencias requeridas puede ser ejecutado en los distintos sistemas operativos mencionados.

4.1 Capturas de Pantalla



(a) Código COOL (b) Lexer (c) Parser

Figura 1: Capturas de pantalla



(a) Contexto (b) Scope (c) Mips

Figura 2: Capturas de pantalla

4.2 Ayuda de la Aplicación

1. Escriba un programa válido en COOL. Por ejemplo:

```
class Main inherits IO {
    main() : SELF_TYPE {
        out_string("Hello, World!!!\n")
    };
};
```

2. Para comenzar el proceso de compilación, continúe con "Analyze/Run (F5)".
   - Podrá ver los resultados en las respectivas pestañas.
3. Se pueden cargar (Ctrl+L) los programas y guardar (Ctrl+S) los resultados alcanzados en diferentes formatos.
4. Podemos encontrar las opciones "Help (F1)", "Orientation (F7)", "Report (F8)" y "About authors (F11)".
   - Orientation (F7): Enseña la orientación del proyecto.
   - Report (F8): Muestra el informe escrito propuesto por los desarrolladores.
   - About authors (F11): Describe una pequeña información acerca de los desarrolladores.
   - Help (F1): Muestra esta ayuda.

5 Gramática

La siguiente gramática está escrita y especificada en términos de notación de Backus-Naur para el lenguaje de programación COOL. Dicha notación es utilizada para expresar gramáticas libres del contexto. Para mayor información podemos consultar el manual de referencia de COOL.

```
<program> ::= <classes>
<classes> ::= <classes> <class> ;
            | <class> ;
<class> ::= class TYPE <inheritance> { <features_list_opt> } ;
<inheritance> ::= inherits TYPE
                | <empty>
<features_list_opt> ::= <features_list>
                      | <empty>
<features_list> ::= <features_list> <feature> ;
                  | <feature> ;
<feature> ::= ID ( <formal_params_list> ) : TYPE { <expression> }
            | <attribute_init>
<formal_params_list_opt> ::= <formal_params_list>
                           | <empty>
<formal_params_list> ::= <formal_params_list> , <formal_param>
                       | <formal_param>
<formal_param> ::= ID : TYPE
<attribute_init> ::= ID : TYPE <- <expression>
                   | <attribute_def>
<attribute_def> ::= ID : TYPE
<expression> ::= ID <- <expr>
               | <expression>.ID( <arguments_list_opt> )
               | <expression><at-type>.ID( <arguments_list_opt> )
               | <case>
               | <if_then_else>
```

```
                | <while>
                | <block_expression>
                | <let_expression>
                | new TYPE
                | isvoid <expr>
                | <expression> + <expression>
                | <expression> - <expression>
                | <expression> * <expression>
                | <expression> / <expression>
                | ~ <expression>
                | <expression> < <expression>
                | <expression> <= <expression>
                | <expression> = <expression>
                | not <expression>
                | ( <expression> )
                | SELF
                | ID
                | INTEGER
                | STRING
                | TRUE
                | FALSE
<arguments_list_opt> ::= <arguments_list>
                       | <empty>
<arguments_list> ::= <arguments_list_opt> , <expression>
                   | <expression>
<case> ::= case <expression> of <actions> esac
<action> ::= ID : TYPE => <expr>
<actions> ::= <action>
            | <action> <actions>
<if_then_else> ::= if <expression> then <expression> else <expression> fi
<while> ::= while <expression> loop <expression> pool
<block_expression> ::= { <block_list> }
<block_list> ::= <block_list> <expression> ;
               | <expression> ;
<let_expression> ::= let <nested_vars> in <expression>
<nested_vars> ::= <let_var_init>
                | <nested_vars> , <let_var_def>
<let_var_init> ::= ID : TYPE <- <expression>
                 | <let_var_def>
<let_var_def> ::= ID : TYPE
<empty> ::=
```

6 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 tres 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.

6.1 Análisis Léxico y Sintáctico

En estas dos fases se emplearon las herramientas de construcción de compiladores *lex* y *yacc* a través del paquete de python *ply*. El mismo incluye la compatibilidad con el análisis sintáctico LALR(1), así como la validación de entrada, informes de errores y resulta ser bastante exigente con la especificación de reglas gramaticales y de tokens. PLY consta de dos módulos separados: *lex.py* y *yacc.py*. El primero se utiliza para dividir el texto de entrada en una colección de tokens especificados por una colección de reglas en forma de expresiones regulares (tokenización), mientras que el segundo se utiliza para reconocer la sintaxis del lenguaje que se ha especificado en forma de gramática libre del contexto. Las dos herramientas están diseñadas para trabajar juntas.

El módulo *yacc.py* implementa la componente de parsing de PLY teniendo como salida un árbol de sintaxis abstracta representativo del programa de entrada. *Yacc* utiliza la técnica de análisis sintáctico LR o shift-reduce. Tanto *lex* como *yacc* proveen formas de manejar los errores lexicográficos y sintácticos, que se determinan cuando fallan las reglas que les fueron definidas. Es importante señalar que tiene como requisito la especificación de la sintaxis en términos de una gramática BNF (notación de Backus-Naur). Esta notación es utilizada para expresar gramáticas libres del contexto. En la sección anterior se encuentra especificada la gramática utilizada en el proyecto.

El lexer y el parser del proyecto se encuentran implementados en los módulos *lexing/lexer.py* y *parsing/parser.py* respectivamente, además *utils/ast.py* ofrece la jerarquía del AST de COOL propuesta.

6.1.1 Palabras Claves

Las palabras claves no pueden ser usada como nombre de variables, pues su uso es exclusivo en el lenguaje para definir la estructura del mismo.

- case
- class
- else
- esac
- false
- fi
- if
- in
- inherits
- isvoid
- let
- loop
- new
- not
- of
- pool
- then
- true
- while

6.1.2 Otros Símbolos

Todos los símbolos presentados con posterioridad son aceptados por el compilador, cualquier otro símbolo fuera de los listados solo son permitidos cuando se encuentran dentro de estructuras como los comentarios y las cadenas de texto.

Símbolo	Descripción	Símbolo	Descripción
SEMICOLON	;	EQUAL	=
COLON	:	PLUS	+
COMMA	,	MINUS	-
DOT	.	STAR	*
OPAR	(	DIV	/
CPAR	)	LESS	<
OCUR	{	LESSEQ	<=
CCUR	}	ID	
LARROW	<-	TYPE	
ARROBA	@	NUM	
RARROW	=>	STRING	
NOX	~		

6.2 Análisis Sintáctico

Durante esta fase se analiza el cumplimiento de todos los predicados semánticos del lenguaje y, por tanto, el uso correcto de los tipos declarados. A continuación centraremos la atención en el problema de la verificación de dichos tipos.

Primeramente tenemos que hacer un recorrido por todo el AST para encontrar las definiciones de tipos, las cuales serán almacenadas en el concepto *Context*. Esto lo haremos utilizando el patrón *visitor*, el mismo sería utilizado en las siguientes pasadas al AST. Es importante conocer los nombres de las clases definidas de antemano, ya que podemos tener una declaración de un tipo *A* con un atributo de tipo *B*, donde la declaración del tipo *B* aparece luego de la de *A*. Con la clase *TypeCollector* se logra crear un contexto inicial que solo contendrá los nombres de los tipos, por eso es que solo visita los nodos del AST de tipo *ProgramNode* y *ClassDeclarationNode*.

```
class TypeCollector(object):
    def __init__(self, errors=[]):
        self.context:Context = None
        self.errors:list = errors

    @visitor.on('node')
    def visit(self, node):
        pass

    @visitor.when(ProgramNode)
    def visit(self, node:ProgramNode):
        pass

    @visitor.when(ClassDeclarationNode)
    def visit(self, node:ClassDeclarationNode):
        pass
```

Importante mencionar que la información referente a los tipos se almacena en el contexto a través de la clase *Type*, y que la misma incluye la función *conforms_to* con el objetivo de establecer la relación de conformidad entre tipos y garantizar el principio de sustitución. Posteriormente se pasa a construir los tipos como tal, con sus definiciones de atributos y métodos, por tal motivo se visitarán además los nodos que definen a los mismos en el AST. Durante esta pasada de la clase *TypeBuilder* también se chequea que la jerarquía de tipos conformada esté correcta y sea un árbol con raíz en el tipo *Object*. Esto último se resolvió comprobando si el grafo de tipos representa un orden topológico.

Por último tenemos un *TypeChecker* que verificaría la consistencia de tipos en todos los nodos del AST. El mismo recibe el contexto construido anteriormente y procesa por completo el AST. A lo largo de este recorrido fue esencial el uso del concepto *Scope*, el cual permite gestionar las variables definidas en

los distintos niveles de visibilidad, así como saber con qué tipo se definieron. Cada clase tiene su propio ámbito o *Scope* y a su vez cada método definido en esta. No obstante, la importancia de este concepto también se demuestra en el chequeo de las expresiones *Let* y *Case*, pues ambas poseen un ámbito interno para nuevas variables locales que podrían definir, por tanto, el *Scope* hijo que se le pasa a estos nodos permite desambiguar entre todas las variables declaradas.

Para manejar los errores de forma consistente, cada una de las clases anteriores posee como atributo una lista de errores de tipo *SemanticError*. De modo que ante cualquier error de chequeo de tipos, simplemente se crea una instancia de esta clase y se a nade a la lista. Aclarar que a cada error se le pasa la línea y columna del nodo del AST correspondiente.

6.3 Generación de Código

Esta fase comprende dos etapas esenciales. Primeramente es necesario traducir el código de COOL a un lenguaje que nos permita generar código de forma más sencilla. Este lenguaje se denomina CIL y todo programa en él tiene 3 secciones: *.TYPES*, *.DATA* y *.CODE*.

Durante esta etapa se realiza un recorrido del AST de COOL y se obtiene un AST de CIL, representando toda la información y semántica necesaria del programa de COOL. Con ello logramos un mayor nivel de abstracción al disponer de instrucciones en 3-direcciones y de cualquier cantidad de registros. Durante la segunda etapa se realiza un recorrido sobre el AST de CIL conformado para generar el código de MIPS finalmente. En ambas etapas, se emplea el patrón visitor.

6.3.1 Código Intermedio (CIL)

El módulo correspondiente a la generación del código intermedio es *code_generation/code_generation.py*. Aquí se encuentra el recorrido realizado sobre el AST de COOL por medio de la clase *Visitor*. Dicha clase contiene una serie de atributos claves y métodos auxiliares para facilitar la generación:

1. La variable *cil_program_node_info* es de tipo diccionario y tiene tres elementos con llaves *CODE*, *DATA* y *TYPES* donde almacenan los nodos correspondientes a las secciones *.CODE*, *.DATA* y *.TYPES* respectivamente de un programa en CIL.

2. La variable *current* es de tipo diccionario y posee dos elementos con llaves *TYPE* y *FUNCTION* donde almacenan instancias de *Type* y *Method* respectivamente.

3. Para definir variables locales e instrucciones dentro de *current* se usan las funciones auxiliares *generateLocalNode* y *saveCILInstruction*.

6.3.2 Código Ensamblador (MIPS)

En el módulo *code_generation/code_generation.py* se encapsula el proceso de generar código MIPS. Se tiene la clase *CILToMIPS* que se encarga de recorrer el AST de CIL generado anteriormente.

6.3.3 Objetos

A la hora de generar código MIPS es necesario establecer ciertos convenios, y uno de los más importantes es la forma en que se van a organizar los objetos en memoria. Para mostrar el diseño utilizado se va a hacer uso de las clases declaradas en la figura 3.

```
class A {
    a : Int <- 0;
    f(): Int { a };
};
```

```
class B inherits A {
    b : Int <- 1;
    f(): Int { b };
    g(): Int { a <- a + b };
};
```

```
class C inherits A {
    c : Int <- 2;
    h(): Int { a <- a - c };
};
```

Figura 3: Declaración de clases de ejemplo en COOL

En la figura 4 a la izquierda se muestra como quedarían dispuestos en memoria los objetos de tipo A, B y C definidos en la figura 3. Para cada objeto se va a tener que, a partir de su dirección de memoria, en el offset 0 se va a encontrar un tag que va a ser un número único para cada clase. Este número se va a corresponder con el orden en que se visita cada clase en un recorrido de tipo *DFS* sobre el árbol que representa la jerarquía de clases comenzando por la clase *Object* que siempre tendría tag 0. A continuación en el offset 4 se tiene un puntero que apunta a una dirección del segmento de datos donde se encuentra alamacenado el nombre de la clase. En el offset 8 va a estar contenido un número que representa el tamaño de la clase, lo cual se podría ver como cuántos cuadros ocupa. Le sigue en el offset 12, un puntero que apunta a una dirección donde van a estar dispuestas las funciones declaradas por la clase o heredadas, lo cual se puede apreciar en el cuadro que se muestra a la derecha y se explicará con más detalle luego. Por último a partir del offset 16 se van a encontrar el valor de los atributos declarados por la clase o heredados. Estos atributos van a aparecer en el orden en que fueron declarados, empezando por los pertenecientes al ancestro más lejano hasta llegar a los declarados por la propia clase.

Offset / Class	0	4	8	12	16	20
A	Atag	“A”	5	*A	a	
B	Btag	“B”	6	*B	a	b
C	Ctag	“C”	6	*C	a	c

Offset / Address	0	4
*A	fA	
*B	fB	g
*C	fA	h

Figura 4: Forma de colocar los objetos en memoria

En el cuadro que se encuentra en la parte derecha de la figura 4 se tiene cómo se van a organizar las funciones pertenecientes a una clase. Como se puede observar se va a seguir el mismo criterio de orden que en los atributos. La diferencia es que una clase puede redefinir las funciones heredadas y esto se va a reflejar como una sustitución en el offet que le corresponde a la función heredada, como se muestra en el ejemplo, que B redefie la función f y por tanto en ese offset va a apuntar a su propia definición, al contrario de C que va apuntar a la definida por A.

Este diseño está pensado para lograr el polimorfismo en los objetos pues, tanto para atributos como funciones, siempre los heredados van a estar en el mismo offset que en los ancestros, lo que beneficia que en los casos que un clase quiera sustituir a un ancestro sus atributos y funciones van a tener la misma forma de buscarse. En el caso de las clases built-in también se va a hacer uso del diseño explicado y los valores de estas van a estar almacenados como un atributo. El valor void va a estar representado como una dirección estática en memoria.

6.3.4 Funciones

La forma en que se resuelven los llamados a funciones incluyen otros de los convenios utilizados. Uno de los principales problemas a resolver fue dónde se iban a almacenar los parámetros que recibe una función. Para esto se decidió hacer uso de la pila, de forma que cuando se ejecuta la primera instrucción de una función los parámetros van a estar colocados en la pila en orden contrario al que fueron declarados y se podrá acceder a ellos usando como referencia el registro $\$sp$.

Otra situación a resolver es que para llevar a cabo cualquier operación es necesario almacenar valores temporales, que denominaremos *locals* de una función. Estos valores también van a estar almacenados en la pila. Desde CIL se tiene conocimiento de cuántos *locals* se van a necesitar y lo primero que va a hacer una función es "salvar" espacio en la pila para guardarlos y se va a tener conocimiento del offset que va a corresponder a cada *local* que también van a ser referenciados usando $\$sp$. Al final de la ejecución de una función el valor que esta retorna se va a almacenar siempre en el registro $\$a1$. Por último la función se encarga de sacar de la pila tanto los *locals* como los parámetros.

7 Problemas Técnicos y Teóricos

7.1 Comentarios

Durante la fase lexicográfica el mayor reto enfrentado fue la tokenización de strings y comentarios de múltiples líneas. Esto fue resuelto mediante el uso del concepto de estado que provee la herramienta *ply*. Cuando se detecta " o * el lexer entra en un nuevo estado en el que solo se regiría por las reglas definidas para dicho estado. Este comportamiento se mantendría hasta que se detecte " o *\ y se regresa al procesamiento normal. En el caso de los comentarios se mantiene un contador de los * encontrados para manejar los comentarios anidados.

7.2 Expresiones Case

Uno de los problemas más interesantes que se presentó fue la implementación de la funcionalidad que provee la expresión *case*. Es necesario dado un tipo T determinar, en tiempo de ejecución, entre un conjunto de tipos T_i tal que $1 < i \leq n$ cuál es el menor tipo con el que T se conforma. Para resolver esto se le asignó a cada tipo un *tag*, el cual se va a corresponder con el orden en que se visita cada clase en un recorrido de tipo DFS sobre el árbol que representa la jerarquía de clases comenzando por la clase *Object* que siempre tendría *tag* 0. Además cada tipo T va a saber cuál es el tipo con mayor *tag* $max\_tag(T)$ al que se puede llegar en un recorrido DFS desde él. Por la forma en que está definido el *tag* se puede decir que un tipo T se conforma con un tipo A si $tag_A \leq tag_T \leq max\_tag(A)$. Por tanto si se chequean los tipos T_i ordenados de mayor a menor según el *tag* el primero que cumpla dicha condición va a ser el tipo buscado.

7.3 Jerarquía de Errores

Para imprimir los errores nos inventamos una jerarquía de clases. Partimos de la clase base *CoolError*, la cual posee los atributos línea y columna. Luego, hacemos por cada tipo de error una clase que hereda de *CoolError* especificando el tipo de error y el texto del error. A continuación, una síntesis de la jerarquía mencionada, aunque puede consultarla en su totalidad en *utils/errors.py*.

```
class CoolError(Exception):
    def __init__(self, line, column, text):
        super().__init__(text)
        self.line = line
        self.column = column
    @property
    def error_type(self):
        return 'CoolError'
    @property
    def text(self):
        return self.args[0]
    def __str__(self):
        return f'({self.line}, {self.column}) - {self.error_type}: {self.text}'
    def __repr__(self):
        return str(self)

class CompilerError(CoolError):
    @property
    def error_type(self):
        return 'CompilerError'

class LexicographicError(CoolError):
    @property
    def error_type(self):
        return 'LexicographicError'
```

```
class SyntacticError(CoolError):
    @property
    def error_type(self):
        return 'SyntacticError'

class SemanticError(CoolError):
    @property
    def error_type(self):
        return 'SemanticError'

class NamesError(SemanticError):
    @property
    def error_type(self):
        return 'NameError'

class TypesError(SemanticError):
    @property
    def error_type(self):
        return 'TypeError'

class AttributesError(SemanticError):
    @property
    def error_type(self):
        return 'AttributeError'
```

8 Bibliografía

1. Manual de COOL.
2. Manual de SPIM.
3. Introducción a la Construcción de Compiladores.