IceBox Compiler

- Darío Fragas González C411
- Abraham González Rivero C412

1. Detalles técnicos

A continuación se describe como compilar un código dado en COOL a MIPS. Ejecute:

```
cool.py [-h] [--output_file OUTPUT_FILE] [--lexer] [--parser] [-t] [-c]
               [--log-level LOG_LEVEL]
               input_file

IceBox Compiler

argumentos posicionales:
  input_file : el archivo que donde se encuentra el código fuente.

argumentos opcionales:
  -h, --help : muestra este mensaje y termina la ejecución.
  --output_file: el archivo de salida( default: el mismo que se dio como
entrada pero con .mips como extensión).
  --lexer: ejecuta solo etapa de tokenización del lexer.
  --parser: ejecuta hasta la etapa de parsing.
  -t: printea los tokens que dio el lexer.
  -c, --cil: da como salida un programa de CIL
  --log-level LOG_LEVEL : setea el modo del log(default:INFO).
```

Estructura y organización de los directorios del proyecto a partir de la carpeta *src*:

```
.
├── codegen
│       ├── __init__.py
│       ├── cil_ast.py
│       ├── cil_to_mips.py
│       ├── cocl_to_cil.py
│       ├── mips_ast.py
│       └──  mips_codegen.py
├── parsing
│       ├── __init__.py
│       ├── ast.py
│       ├── lex.py
│       └── parser.py
├── semantic
│       ├── __init__.py
│       ├── context.py
│       ├── scope.py
│       ├── type_builder.py
│       ├── type_checker.py
│       ├── type_collector.py
│       └── types.py
├─ test_cool
│       ├── dispatch.cl
│       └── formals.cl
├─ utils
│       ├── loggers.py
│       └── visitor.py
├── cool.py
├── coolc.sh
├── makefile
├── Readme.md
```

2. Arquitectura del Compilador

El desarrollo del proyecto se dividió en 3 módulos principales: codegen, parsing y semantic.

Análisis lexicográfico

Análisis léxico Este proceso se realiza en *lex.py*. Proceso 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 lexer se auxilia de la clase *CharacterStream* para manipular la cadena:

```
class CharacterStream:
    def __init__(self, source_code: str):(...)

    def next_char(self) -> Optional[str]:(...)

    def peek_char(self) -> Optional[str]:(...)

    def get_position(self)-> CharPosition:(...)

    def reset(self):(...)
```

Esta clase ofrece herramientas para manipular la cadena como: obtener el próximo carácter de la cadena, obtener la posición,línea y columna, en que se encuentra el carácter, étc.

En la clase *TokenType* se definen todos los tipos de token que pudieran encontrarse en la cadena:

```
class TokenType(Enum):
    OBJECTID = "OBJECTID"
    INT_CONST = "INT_CONST"
    STRING_CONST = "STRING_CONST"
    DOT = "DOT"
    COMMA = "COMMA"
    COLON = "COLON"
    SEMICOLON = "SEMICOLON"
    AT = "AT"
    (...)
    EOF = "EOF"
```

Para representar un token se define la clase *Token* que posee un valor, un tipo y la posición en que se encuentra el token.

```
 def __init__(self, type, value, position):
        self.type = type
        self.value = value
        self.position = position
        log.debug(
            f"Created Token:",
            extra={"type": type, "location":          position,"value": va
        )
```

La clase *Lexer* se encarga de la tokenización de la cadena. Se inicializa recibiendo una cadena, la cual convierte a un *CharacterStream*, explicado anteriormente. Posee además un conjunto de keywords que se mapean a tokens y una lista de errores. La tokenización se realiza en el método *lex* el cual va consumiendo los caracteres de la cadena llamando al método *fetch_token* quien va realizando la validación de los caracteres y armando los tokens. Si un error es detectado se reporta pero se continua la ejecución. El proceso de tokenización se realiza en $O(|w|)$ donde $|w|$ es la longitud de la cadena.

```
class Lexer:
   def __init__(self, stream):
     (...)

   def error(self, message, position: Optional[CharPosition] = None):
     (...)

   def fetch_token(self):
     (...)

   def lex(self):
     (...)
```

Análisis sintáctico

El proceso de parsing se realiza en *parser.py*. La clase Parser se encarga de realizar esta tarea. Nuestra implementación se inicializa al recibir una lista de tokens.

La definición de los nodos del *AST*(árbol de sintaxis abstracta) se encuentra implementada en *ast.py*. Todo nodo posee un valor, un tipo y una posición e implementan el patrón visitable, con el método *accept* que permite a un visitor visitar el nodo. Se muestran algunos nodos como ejemplo:

```
    class ProgramNode(Node):
    def __init__(self, classes, location):
        self.classes: List[ClassNode] = classes
        super().__init__(location)

    def __str__(self) -> str:
        return "\n".join(str(c) for c in self.classes)

    class FeatureNode(Node):
    def __init__(self, location):
        super().__init__(location)

    class AttributeNode(FeatureNode):
    def __init__(self, name, attr_type, init: "ExpressionNode", location):
        super().__init__(location)
        self.name = name
        self.attr_type = attr_type
        self.init = init
```

Para parsear se utiliza el método *parse*. En este se origina el árbol de sintaxis abstracta. El parser construye los distintos nodos en profundidad, o sea , primero los hijos y luego el padre. Por ejemplo, para construir el nodo *Program*, raíz del *AST* de *COOL*, ya se deben haber construido todos los nodos Class. Un programa de *COOL* consiste en una serie de definiciones de clases. Cada clase a su vez posee un conjunto de atributos y de funciones. Las expresiones que pueden formar parte de dichas funciones son el corazón del lenguaje. Cada nodo conoce sus posibles reglas de derivación y como evitar errores de desambiguación entre las producciones a la hora de determinar que producción aplicar. El método *eat* se encarga de consumir el token y validar que cumpla con los requisitos de la producción en cuestión. El *parsing* se realiza en tiempo lineal gracias a que se tiene en cuenta la precedencia de operadores, reportando todos los errores que encuentra. Quedaría mejorar descartar *Tokens*, cuando se encuentra un error, hasta un nuevo punto estable, pues al encontrar un error surgen múltiples errores condicionados por el primero.

Producciones

```
program := [[class]]+
class ::= class TYPE [inherits TYPE] { [[feature; ]]*}
feature := ID method
                | ID attribute

                | ID : TYPE [ <- expr]
method := ( [ formal [[, formal]]*] ) : TYPE { expr }
formals := formal [[, formal]]*
expr ::=ID <- expr
           | expr [@TYPE].ID( [ expr [[ , expr ]]*] )
           | ID( [ expr  [[ , expr ]] *] )
           | if expr then expr else expr if
           | while expr loop expr pool
           | { [[ expr ; ]]+}
           | let ID : TYPE [<-expr] [[,ID : TYPE [<- expr ]]]*in expr
           | case expr of [[ID : TYPE => expr; ]]+esac
           |new TYPE
           |isvoid expr
           |expr+expr
           |expr- expr
           |expr*expr
           |expr/expr
           |ẽxpr
           |expr<expr
           |expr<=expr
           |expr=expr
           |not expr
           |(expr)
           |ID
           |integer
           |string
           |true
           |false
```

Análisis semántico

El análisis semántico se realiza en el módulo *semantic*.En esta fase es donde se revisa que se cumplan todos los predicados semánticos que caracterizan al lenguaje *COOL* y por tanto se revisa la consistencia y uso correcto de los tipos declarados. Para el chequeo semántico son necesarios tres recorridos sobre el *AST* obtenido del proceso de *parsing* . La primera para recolectar los tipos para el contexto, la segunda para definir los atributos y métodos de cada tipo y la tercera para realizar el chequeo de tipos. Utilizando el patrón *Visitor* es como realizamos los pertinentes recorridos por el *AST*.

La clase TypeCollector definida en _type_collector.py_se verán solo las declaraciones de clases, para guardar todos los tipos definidos en el lenguaje.

Para ellos nos apoyaremos de un contexto, clase *Context* definida en *context.py* y que funciona parecido a un diccionario, en el cual se guardaran los tipos de las clases a medida que las vamos visitando.

```
    class TypeCollector(Visitor):
    def __init__(self) -> None:
        self.context = None
        self.errors = []

    def error(self, message, location, type, value):
      (...)


    def visit__ProgramNode(self, node: ProgramNode):
      (...)

    def visit__ClassNode(self, node: ClassNode):
      (...)
```

Luego de realizar el recorrido y haber encontrado todos los tipos, se procede a ejecutar una segunda pasada por el *AST*. De esta tarea, se encarga el *TypeBuilder* definido en *type_builder.py*. *TypeBuilder* pasará por cada tipo encontrado para construirlo junto con sus atributos y métodos. A este visitor se le pasa el contexto generado por el

TypeCollector.py . En este punto la tarea principal es la de visitar los atributos y métodos de los tipos y agregárselos, sin antes haber realizado algunas comprobaciones, como que no se definan atributos de tipos que no existen, o que se redefinan atributos de los padres o que existan múltiples atributos con el mismo nombre. Análisis semejantes son hechos con las funciones, impidiendo la creación de métodos ya existentes en una misma clase o funciones que son sobrescritas por herencia sigan teniendo la misma cantidad de argumentos y tipo de retorno. Para el caso de herencia se lanzan errores cuando se hereda de tipos que no existen. O de los tipos *Int*, *Bool*, *String* y *Obj* que no se permite.

```
class TypeBuilder(Visitor):
  def __init__(self, context: Context, errors=None):
    (...)


  def error(self, message, location, type, value):
    (...)

  def visit__ProgramNode(self, node: ProgramNode):
    (...)

  def visit__ClassNode(self, node: ClassNode):
    (...)

  def visit__AttributeNode(self, node: AttributeNode):
    (...)


  def visit__MethodNode(self, node: MethodNode):
    (...)
```

Finalmente en el último recorrido del *AST* ,*TypeChecker*, definido en *type_checker.py* ,se encarga de chequear todos los nodos del *AST* cerciorándose de que cumplan con las reglas definidas para ellos. Por ejemplo en esta fase es donde resolvemos conflictos de realizar operaciones binarias sobre tipos que no lo permiten (ejemplo sumar dos *strings*). Otro de los análisis es que para trabajar con una variable en una clase, esta ha de estar definida, ya sea como atributo o como una variable dentro de un let, etc. Es por ello que es de vital importancia en este recorrido el uso de un *scope*, definido en *scope.py* , este concepto permite conocer las variables que tenemos visibles en los diferentes niveles. Esto es de vital importancia pues cuando tratamos funciones que reciben argumentos con nombres iguales a atributos de clase, queremos tratar con el argumento, y no con los atributos de la clase. Es por ellos que cada clase define su propio y este es pasado a sus hijos, pero las expresiones como el case y let, que puede definir nuevas variables, reciben su propio *scope*, con lo que se desambigua entre todas las variables declaradas. El TypeChecker es la herramienta fundamental que permite llevar a cabo el polimorfismo en el lenguaje, pues junto con el se verifica que un tipo pueda ser sustituido por otro, además que es esencial para determinar atributos que se definen en clases padre y se utilizan en una clase hijo, etc. Además, la importancia del *TypeChecker* radica en que determina el tipo estático de cada expresión, lo que nos permite después en la generación de código hacer llamados polimórficos en $O(1)$.

```
class TypeChecker(Visitor):
    def __init__(self, context: Context, errors=None):


    def error(self, message, location, type, value):


    def visit__ProgramNode(self, node: ProgramNode, scope=None):


    def visit__ClassNode(self, node: ClassNode, scope: Scope):


    def visit__AttributeNode(self, node: AttributeNode, scope: Scope):



    def visit__MethodNode(self, node: MethodNode, scope: Scope):
```

```
    def visit__AssignNode(self, node: AssignNode, scope: Scope):

    def visit__DispatchNode(self, node: DispatchNode, scope: Scope):

    def visit__BinaryOperatorNode(self, node: BinaryOperatorNode, scope: S

    def visit__UnaryOperatorNode(self, node, scope):

    def visit__IfNode(self, node: IfNode, scope: Scope):

    def visit__BlockNode(self, node: BlockNode, scope: Scope):

    def visit__LetNode(self, node: LetNode, scope: Scope):

    def visit__CaseNode(self, node: CaseNode, scope: Scope):

    def visit__CaseOptionNode(self, node: CaseOptionNode, scope: Scope):

    def visit__NewNode(self, node: NewNode, scope: Scope):

    def visit__IsVoidNode(self, node: IsVoidNode, scope: Scope):

    def visit__NotNode(self, node: NotNode, scope: Scope):

    def visit__PrimeNode(self, node: PrimeNode, scope: Scope):

    def visit__IdentifierNode(self, node: IdentifierNode, scope: Scope):

    def visit__IntegerNode(self, node: IntegerNode, scope: Scope):
    def visit__StringNode(self, node: StringNode, scope: Scope):

    def visit__BooleanNode(self, node: BooleanNode, scope: Scope):

    def visit__MethodCallNode(self, node: MethodCallNode, scope: Scope):

    def visit__WhileNode(self, node: WhileNode, scope: Scope):
```

Generación de código

El paso de *COOL* a *Mips* es demasiado complejo, por ello se dividió el proceso de generación de código en dos etapas:

Paso de Cool a CIL:

Para la generación de código intermedio nos auxiliamos del lenguaje de máquina *CIL*, que cuenta con capacidades orientadas a objetos y nos va a permitir generar código *MIPS* de manera más sencilla. El *AST* de *CIL* se obtiene a partir de un recorrido por el *AST* de *COOL*, para el cual nos apoyamos nuevamente en el patrón visitor. El objetivo de este

recorrido es desenrollar cada expresión para garantizar que su traducción a *MIPS* genere una cantidad constante de código. *CIL* tiene tres secciones: * TYPES: contiene declaraciones de tipo. * DATA: contiene todas las cadenas de texto constantes que serán usadas durante el programa. * CODE: contiene todas las funciones que serán usadas durante el programa.

Al convertir de Cool a CIL se obtiene el mapeo correcto de atributos y métodos por la forma de recorrer el árbol en el análisis semántico.

```
for type in self.context.types.values():
            self.attrs[type.name] = {
                attr.name: (i, htype.name)
                for i, (attr, htype) in enumerate(type.all_attributes())
            }
            self.methods[type.name] = {
                method.name: (i, htype.name)
                if htype.name != "Object"
                or method.name not in ["abort", "type_name", "copy"]
                else (i, type.name)
                for i, (method, htype) in enumerate(type.all_methods())
            }
```

La primera sección que se construye es .TYPES con las declaraciones de los tipos que se van a usar en el programa. En CIL no existe el concepto de herencia, la forma de asegurar que un tipo pueda acceder a sus métodos y atributos heredados es declarándolos explícitamente en su definición. Además, es necesario garantizar que el orden en que se definen los mismos en el padre se conserve en los descendientes. Para ello a la hora de definir un tipo A se declaran en orden los atributos y métodos correspondientes comenzando por los de su ancestro más lejano hasta llegar a su padre y a los propios. Nótese que se hace necesario guardar el tipo al que pertenece el atributo o método originalmente, a continuación se explica por qué. Dado un tipo A que hereda de B ¿Qué pasa con los atributos heredados cuando vamos a crear una instancia de A? ¿Cómo accedemos a la expresión con que se inicializa cada atributo si se declaró en otro tipo? Después de un breve análisis, salta a la luz que es necesario que los atributos tengan constructores. Entonces, inicializar un atributo heredado se traduce a asignarle el valor que devuelve el constructor del mismo. Para hacer el llamado a dicho constructor es necesario saber el tipo donde fue declarado el atributo originalmente, por eso se guarda en el proceso de construcción del tipo antes descrito. Lo mismo sucede con los métodos. La sección .DATA se llena a medida que se visitan cadenas de texto literales, además se añaden algunas otras que nos serán útiles más adelante. Por ejemplo, se guardan los nombres de cada tipo declarado para poder acceder a ellos y devolverlos en la función type_name. Expliquemos entonces de qué va la sección .CODE, que no por última es menos importante. De manera general, está conformada por las funciones de COOL que se traducen a CIL. En el cuerpo de estas funciones se encuentra la traducción de las expresiones de COOL.Este proceso se hace más complejo para ciertos tipos de expresiones. Analicemos una de estas. Las expresiones *case* son de la siguiente forma:

```
case  expr0  of
 ID_1  :  TYPE_1  =>  expr1 ;
. . .
 ID_n > : TYPE_n  =>  exprn ;
esac
```

Esta expresión se utiliza para hacer pruebas sobre el tipo de los objetos en tiempo de ejecución. Con ese fin, se evalúa *expr0* y se guarda su tipo dinámico *C*. Luego se selecciona la rama cuyo tipo *TYPE_k* es el más cercano a *C* entre los tipos con que *C* se conforma y se devuelve el valor del *exprk* correspondiente. El tipo dinámico *C* no se conoce hasta el momento de ejecución, que es cuando se evalúa la expresión, por tanto, la decisión de por qué rama se debe decantar el case no se puede tomar desde *CIL*. La solución consiste entonces en indicarle a *MIPS* los pasos que debe tomar en esta situación. Para esto, se genera el código *CIL* para cualquiera de los posibles tipos dinámicos de *expr0*, que no son más que todos los tipos que heredan del tipo estático de *expr0*.

De *CIL* a *MIPS*:

Para la generación de código *MIPS* se definió un visitor sobre el *AST* de *CIL* generado en el paso anterior, este visitor generará a su vez un *AST* de *MIPS* que representa las secciones

.data y .text con sus instrucciones; donde cada nodo conoce su representación en *MIPS*. Posteriormente se visitará el nodo principal del *AST* de MIPS y se producirá el código que será ejecutado por el emulador de *SPIM*. Al visitar el cil.Program se visitarán los nodos de la sección dottype, para representar en .data la tabla de métodos virtuales, para cuando se produzcan llamadas a métodos no estáticos. Por cada tipo de nodo se registra en .data un label con el nombre del tipo, .word como tipo de almacenamiento, y una serie de labels, cada una correspondiente a un método del tipo.

```
Object : .word, Object_abort, Object_copy, Object_type_name
```

Para acceder a un método específico de un tipo se busca en la dirección de memoria dada por el label correspondiente a este, sumada con el índice correspodiente al método, multiplicado por 4 este índice está dado por el orden en que se declararon los métodos, aquí se hallará un puntero al método deseado. El siguiente paso es visitar la sección dotdata, para registrar los strings declarados en el código de *COOL*, de la siguiente forma:

```
string_1 : .asciiz, "Hello, World.\n"
```

Finalmente se visitarán los nodos de la sección dotcode, que corresponden a las instrucciones del programa. Cada uno de estos nodos es un FunctionNode, en cada uno se van generando nodos del *AST* siguiendo la siguiente línea: * Se reserva el espacio de las variables locales correspondientes a la función. * Se actualiza el frame pointer #$fp# con el stack pointer. * Se guarda la dirección de retorno $ra en la pila. * Se guarda el frame pointer anterior en la pila. * Se visitan las instrucciones de la función. * Se restaura el puntero al bloque remplazándolo con el que había sido almacenado. * Se restaura la dirección de retorno.

- Se libera el espacio que había sido reservado en la pila.

Siempre se conocerá el offset, con respecto a $fp correspondiete a las variables locales y parámetros que se utilizan en el cuerpo de una función.

Para realizar llamadas a funciones que reciban argumentos es obligatorio guardar los argumentos en la pila antes de llamar a la función.

El recorrido por las instrucciones no es presenta gran complejidad, es simplemente traducir sencillas expresiones de *CIL* a expresiones de MIPS, sin embargo hay algunos casos interesantes que vale la pena destacar. * La reserva dinámica de memoria para instanciar tipos se realiza mediante *Allocate*, el compilador reservará un espacio de tamaño (CantidadAtributos + 1) * 4. En los primeros 4 bytes ser guarda la dirección del tipo de la instancia, y en las siguientes palabras están reservadas para los atributos. La representación de las instancias de tipos en memoria se estructuró así:

Tipo de Atributo	**Atributo 1**	**Atributo 2**	**…**	**Atributo n**
Dirección	Dirección + 4	Dirección + 8		Dirección + 4*n

- Existen dos tipos de llamados a funciones, llamado estático y dinámico. El llamado estático es muy sencillo es simplementer saltar al label dado mediante la función de MIPS *jal* y al retornar, liberar el espacio en la pila correspondiente a los argumentos pasados a la función. Por otro lado el llamado dinámico es más complejo, pues dada la instancia y el índice del método, se busca en la pila la instancia, se toma la posición 0 que corresponde a la dirección(d) de su tipo, y a partir de esta se obtiene la función que está en d + 4 * i. Luego se salta al label de la función y por último se libera el espacio en la pila correspondiente a los argumentos pasados. Mucho nodos son importantes entre ellos los que corresponden a la entrada y salida, a las operaciones sobre cadenas, y operaciones lógicas y aritméticas, estos llevan más trabajo con la lógica de MIPS que no consideramos necesario de abordar.