COOL COMPILER

- Raul Beltran Gomez
- Victor Amador Sosa

1. Detalles tecnicos

A continuacion se especifica como compilar algunos apartados del proyecto”

- Compilar un caso dado, un .cl. Esto compila un codigo escrito en COOL y lo escribe en MIPS:

```
$ pip install -r requirements.txt
$ cd src
$ python3 [cool.py](http://cool.py) INPUT_FILE, OUTPUT_FILE
```

- Ejecutar los test de lexer, parser y semantica. Ejecuta todos los testers offline sin uso de pytest:

```
$ pip install -r requirements.txt
$ cd src
$ python3 my_tests.py
```

- Compilar todos los casos de prueba con pytest:

```
$ pip install -r requirements.txt
$ src cd
$ make test
```

2. Arquitectura del compilador

Cool Compiler esta disennado para traducir codigo escrito en COOL(Classroom Object-Oriented Languaje) a codigo en MIPS(Microprocessor without Interlocked Pipeline Stages). El proyecto esta estructurado por 4 etapas: Analisis Lexicografico, Analisis Sintactico, Analisis Semantico y Generacion de Codigo.

Estructura

El codigo esta estructurado en 5 modulos.

- Error: Modulo encargado de guardar errores
- Lexer: Modulo encargado de tokenizar el codigo escrito en cool
- Parser: Modulo encargado de parsear los tokens de salida del lexer y generar el AST
- Semantic: Modulo encargado de detectar errores semanticos usando el AST como entrada
- Codegen; Modulo encargado de convertir el codigo escrito en COOL a codigo en CIL, y luego convertir el codigo en CIL a codigo en MIPS

2.1 Errores

La clase `Error` se encuentra implementada en el directorio `src/compiler/error`. Esta clase es la encargada de recibir los errores de la fase lexica, sintactica y semantica. La clase recibe un error y lo guarda con su posicion, descripcion y tipo.

```
class CoolError (Token):
    ERRORS = []
    def __init__(self, by, token = None, pos = None, lineno = None , value = None, index = None, end = None) -> None:
       ...

    def __str__(self):
        return f"ERROR({self.lineno},{self.pos}): ...)"
```

2.2 Analisis Lexicografico

El analisis lexico esta implemetado en el directorio `src/compiler/lexer` consta de un analizador lexico general, un analizador de strings y un analizador de comentarios, los dos ultimos se especializan en una funcion especifica, estas son analizar los strings de cool y los comentarios de cool respectivamente.

El analizador lexico general se centra en tokenizar una cadena de texto escrita en lenguaje COOL, siguiendo las reglas de este. Para realizar este alanilisis se usa la biblioteca `sly`, especificamente el modulo `Lexer`. Sigue los siguientes pasos para lograr la tokenizacion.

Tokens

```
tokens = {
        CLASS, INHERITS,
        IF, THEN, ELSE, FI,
        LET, IN,
        WHILE, LOOP, POOL,
        CASE, OF, ESAC,
        ISVOID, NEW, NOT,
        TRUE, FALSE,
        ID, INT_CONST, TYPE,
        ASSIGN, DARROW, LE,
        STRING
    }
```

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

```
keyword = [ 'class', 'inherits',
            'if', 'then', 'else', 'fi',
            'let', 'in',
            'while', 'loop', 'pool',
            'case', 'of' , 'esac',
            'isvoid', 'new', 'not',
            'true','false'
    ]
```

Tokenize

- Ignora los saltos de línea, las tabulaciones y los espacios en blanco.
- Si el token se encuentra en las palabras clave (`keyword`) lo trata como un token específico en la lista de `tokens`.
- Si el token comienza con mayúsculas lo guarda como un token con `token.type = "TYPE"`.
- Si el token es una cadena de números lo guarda como un token con `token.type = "INT"`.
- Si el token comienza con una letra minúscula o con `_` lo guarda como un token con `token.type = "ID"`.
- Si el token comienza con `"` lo pasa al analizador de strings para procesarlo y guardarlo como string.
- Si el token comienza con `-` trata el resto de la línea como un comentario, hasta el próximo salto de línea.
- Si el token comienza con `(*` lo pasa al analizador de comentarios para ser tratado como inicio de un comentario.
- Si el token es un literal lo guarda con el tipo del literal.
- Por último, los tokens específicos `<-`, `<=` y `=>` son tratados independientemente dado que no son literales ni cumplen con las condiciones anteriores para analizarlos.

Gestion de errores

La gestion de errores en el lexer esta dividida en:

- `Invalid Character`: este error se detecta cuando hay un caracter invalido mientras se tokeniza el codigo, el modulo `Lexer` de sly posee esta deteccion de errores, se logra sobreescribiendo el metodo `error` de la clase `Lexer`, desde aqui se le pasa el error al modulo `Error` implementado.
- `Null Character in String`: este error se detecta el la fase de analisis de strings, si el caracter nulo esta en el string se le pasa este error al modulo `Error`.

- `EOF in string`: este error tambien se detecta en la fase de anailis de string, si el string nunca cierra entonces se pasa este error al modulo `Error`.
- `EOF in comment`: este error se detecta en la fase de analisis de de comentarios, actua igual que el EOF en strings.

Uso

Dada una cadena de texto se le pasa esta al lexer y se le pide la tokenizacion, esto devuelve un iterable de tokens.

```
class CoolLexer(Lexer):
...
code:str = "codigo escrito en COOL"
lexer = CoolLexer()
lexer.tokenize(code)
```

2.3 Analisis Sintactico

El analisis sintactico esta implementado en el directorio `src/compiler/parser`, en el archivo `cool_parser.py`. El analisis sintactico se encarga de generar el AST y de detectar errores sintacticos. Para la implementacion de esta etapa se usa el modulo `Parser` de la biblioteca `sly`. El parser implementado en este modulo corresponde a un parser ascendente LALR(1). El uso de decoradores en este moduo permite parsear los tokens siguiendo una gramatica especifica, dada. Por ejemplo si queremos una `expr` se hace lo siguiente:

```
@_('ID ASSIGN expr')
    def expr(self, p):
        #expr::= ID <- expr
        ...
        return Assign('<-',...)
```

Esto nos dice que una expresion puede ser de la forma que indica el decorador (`expr::= ID <- expr`). Dentro de este metodo se define su comportamiento, en este caso se devuelve un node de tipo `Assing` de nuestro AST.

El parser es el encargado crear el AST asi como asignar posicion a cada uno de los nodos de este AST, para en la fase de analisis semantico tener acceso a estas posiciones.

Al modulo `Parser` se le asigna una precedencia para que no exista conflictos con el `shit/reduce`. Esta precedencia esta dada en el manual de COOL.

```
precedence = (
        ('right', 'ASSIGN'),        #lv1
        ('left','NOT'),             #lv2
        ('nonassoc', '=','<','LE'), #lv3
        ('left', '+', '-'),         #lv4
        ('left', '*', '/'),         #lv5
        ('left', "ISVOID"),         #lv6
        ('left', '~'),              #lv7
        ('left', '@'),              #lv8
        ('left', '.'),              #lv9
    )
```

AST

El AST esta dado por una gerarquia de clases donde el nodo principal es `PlotNode`, esto permite que cada nodo del AST pueda ser ploteado. Es decir para pintar el AST.

Cada uno de los nodos del AST posee el tipo del token, un contexto, una posicion en el codigo, referencia al nodo padre y un conjunto de nodos hijos. La asociasion padre-hijo esta dada por los scope, por ejemplo, un `Feature` es hijo del nodo `CoolClass` donde esta implementado, que a su vez es hijo del nodo `Program`. Un nodo `Int` puede ser hijo de un nodo `ArithmeticOP` que a su vez puede ser hijo de un nodo `Assing`, …, siguiendo esta linea se llega siempre al nodo `Program`.

El Analisis Sintactico esta disennado de tal forma que se genere un Arbol de Sintacsis Abstracta.

Gestion de errores

Al igual que el modulo `Lexer`, el modulo `Parser` de la biblioteca `sly` posee un metodo para la deteccion de errores gramaticales. La deteccion de errores es entonces dada por la sobreescritura de este metodo y, dentro de esta implementacion se le pasa el error al modulo `Errors`.

Uso

Dado un conjunto de tokens resultado de una tokenizacion en la etapa lexicografica, se le pide el metodo `parse` del modulo `Parser` a la intancia correspodiente, y este se encarga de parsear estos tokens siguiendo el orden correspondiente de estos.

```
lexer = CoolLexer()
parser = CoolParser(lexer=lexer)
parser.parse(lexer.tokenize(code))
```

A la clase `CoolParser` se le pasa la instancia del lexer, ya que estainstancia es la encargada de asignar posicion a los tokens.

2.4 Analisis Semantico

Esta implementado en el directorio `src/compiler/semantic`.Para el analisis semantico se hace uso del AST generado por la fase sintactica. Esta fase posee un modulo `context`, este es el encargado de asignar variables y funciones a un contexto dado. Por ejemplo el contexto de una clase tendra difinidos los atributos en la parte de las variables y los metodos en la parte de las funciones, cada una de las funciones pertenecientes a esta clase tendra su propio contexto que es hijo del contexto de la clase en la cual se encuentra, asi, haciendo uso de referencias al contexto padre se puede saber si estan definidos ciertos atributos y funciones. Cada nodo del AST tendra su propio contexto, en casos especiales el contexto sera un contexto interno hijo del contexto padre, como es el caso de los `let`, en los otros casos, donde una expresion no necesite un contexto interno, el contexto de la misma sera el contexto del padre, de esta manera una variable de un `let` tendra como contexto el contexto del `let` padre, y tendra contextos padres hasta la clase, de esta forma puede usar y acceder a variables y funciones declaradas en los contextos padres, como pueden ser los atributos de la clase, las funciones, los parametros de la funcion donde se encuentra o bien otro `let` padre de este. Otros nodos con contextos internos son los `case` y los `def`.

Contextos

En esta fase, los contextos son los encargados de definir variables y funciones, asi como validar que estas puedan ser creadas y/o utiizadas en un contexto dado. Si se intenta usar una variable que no esta definida en un contexto entonces se lanza un error en tiempo de compilacion, una variable o funcion esta definida en un contexto si lo esta definida en el mismo o en un contexto padre de este. Los contextos tambien son los encargados de sobreescribir las variables. Por ejemplo, si se tiene una clase con el atributo `x`, dentro de esta clase se tiene una funcion con parametro `x` y dentro de esta funcion un `let` con una variable de nombre `x`, cada una de estas `x` sera una variable distinta, sobreescrita en cada uno de los contextos hijos, si se pide usar la `x` dentro del `let` entonces se accede a la `x` de este contexto interno, si se pide usar la `x` dentro de la funcion pero no dentro del `let`, entonces se usa la `x` de los parametros de la funcion, si se pide la `x` dentro de la clase o en otra funcion de la clase que no la sobreescriba, entoces se usa la `x` de los atributos. Cada una de estas variables puede ser de un tipo distino, y de cada una de estas se sabra su informacion en el contexto correspondiente.

```
class VariableContext():
    def __init__(self, father) -> None:
        self.types:dict[str:Context] = {}
        self.functions:dict[str:Feature.CoolDef] = {}
        self.variables:dict[str:CoolVar] = {}
        self.father:Context = father
        self.type = env.object_type_name
        self.cclass:CoolClass = None
```

Un contexto entonces tendra los tipos, las funciones y las variables. En el ejemplo anterior cuando se pide `x` se busca por cada una de las variables del contexto, si no esta busca en el padre y asi sucesivamete. Una vez la encuentra en el diccionario, el `value()` de este tiene la informacion de la variable (nodo de AST), informacion donde se ecnuentra su tipo.

Type Checking

Cada nodo del AST tiene informacion de su tipo estatico asi como una funcion `get_type()` para obtener su tipo dado el contexto. La mayoria de los nodos, su tipo es exactamente el tipo estatico, pero en nodos como el `Dispatch` se tiene que `get_type()` es el tipo del `callable` que llama, no tiene un tipo especifico que se le pase, sino que busca en el contexto el tipo que devuelve la funcion que se llama, tambien tiene un tipo estatico, que seria el tipo que se espera de la parte izquierda, de la expresion que llama el `callable`, este tipo se le pasa en el codigo con `@TYPE`. Los `ID` son otro caso que no tienen un tipo definido, sino que buscan su tipo en el contexto con `get_type()`, de esta forma busca en las variables del contexto, y si esta definido entonces su tipo sera el de la variable asociada a este `ID`.

Los casos especiales `case` y `if` tiene mas de un tipo, luego su `type` sera una lista de tipos.

Ejemplos de type-checking son:

- En una operacion aritmetica, se pide `get_type()` de la parte izquierda y de la parte derecha, y se valida que ambos sean de tipo `INT.`
- En una operacion `assign` se pide `get_type()` izquierdo y `get_type()` derecho, y se valida si ambos son del mismo tipo o el tipo de la parte derecha herede del tipo de la parte izquierda.
- Si en la parte derecha de un `assign`, por ejemplo, hay un `case`, se comprueba que cada uno de los tipos posibles del `case` coincidan con el tipo de la part izquierda, o que su tipo herede del tipo de la parte izquierda.

Para saber si un tipo hereda de otro s usa la funcion `inherit_from_type(type)`:

```
def inherit_from_type(self,type):
    '''
    Se llama desde la estructura `E` (de tipo `C`), y se le pasa el tipo `P`,
     esto evalua y devuelve si `C <= P`
    - nota: `C<=P`, en Cool significa que el tipo `C` es `P` o hereda de `P`
    '''
    cclass = self.get_type_as_class()
    if cclass.type == type:return True #En este caso es para cuando entra al contexto de SELF_TYPE. SELF_TYPE --->> Class Type
    if cclass is None:
        return False
    return cclass.inherit_from_type(type)

def get_type_as_class(self):
    '''
    Busca en el contexto el contexto que representa a este tipo, el
    diccionario context.types se enceuntra una asignacion str:Context que
    dado un tipo en string devuelve el contexto de este, luego de este contexto
    obtiene la clase, ya que es un contexto de un tipo, tendra asosiada una
    clase(un tipo).
    '''
    if self.context == None:
        self.get_contex_from_father()

    temp_cotext = self.context.get_context_from_type(self.get_type())
    if temp_cotext == False:
        return None
    else:
        return temp_cotext.cclass
```

Esta funcion busca el contexto en cuestion, la clase a la que pertenece un `ID` dado y busca recursivamente si hereda del tipo que se le pasa. Esta funcion se llama desde un nodo del AST.

Validacion

La validacion se hace a nivel de contexto, para validar un nodo del AST se le pide a su contexto que lo valide pasandole el `self`, de esta forma el contexto recibe a quien debe validar. El proceso de validacion se ejecuta desde los hijos a los padres, primero se compreba que los hijos sean validos(recursivamente) y luego se valida el nodo en cuestion. De esta forma si un hijo no es valido tampoco lo sera el nodo padre de este. La validacion esta dada por las reglas semanticas de COOL, por ejemplo una operacion aritmetica es valida si sus dos hijos son validos y ademas el `get_type()` de cada hijo es `INT`, a su vez un `if` con una operacion aritmetica en su scope de else o then, por ejemplo, es valido si ambos scope son validos, que a su vez es valido si esa operacion aritmetica lo es. La validacion de los `callable` sigue las reglas de COOL, es decir si los argumentos de llamado son del mismo tipo, mismo orden e igual cantidad que los parametros que tiene la definicion de la funcion. Un case es valido si no repite el mismo tipo en la parte izquierda de sus variales, asi como que el tipo que se le asigne exista en el contexto del programa.

Definicion de variables y metodos

Siguiendo las reglas de COOL, para definir una variable primero debe ser valida, luego, en caso de ser un atributo, se debe buscar en el contexto de la clase o en los contextos padres que esta no este definida anteriormente, ademas el tipo estatico que se le asignaa la variable debe estar definido en los tipos del contexto del programa, si no cumple con estas condciones, no es valido definir la variable y lanza error. En caso de los metodos se busca en el propio contexto que no esten ya definidos, si no lo estan se busca en los contextos padres para ver si hay sobreescritura, en caso que no haya sobreescritura y no este definido en el mismo contexto, entonces es valido definir el metodo; en caso de sobreescritura hay que comprobar que el tipo de salida del metodo sea el mismo, asi como la cantidad de parametros y tipo de estos sean los mismos que el metodo que se quiere sobreescribir. La definicion de variables en los let esta dada por que sea valida la parte derecha y el tipo asiganado a la variable existe.

Gestion de errores

En la semantica existen varios tipos de errores, `TypeError`, `NameError` y `SemanticError`:

- **TypeError:** este esta dado por el check type, si los tipos no corresponden con los esperados, un ejemplo de ello es en la operacion aritmetica, que espera operandos de tipo `INT`. `TypeError: non-Int arguments: Int + String`
- **NameError:** este se lanza cuando no esta definido un nombre, un `ID`. Por ejemplo si intentaos usar un nombre no definido. `NameError: Undeclared identifier d.`
- **SemanticError:** este se lanza cuando se rompe una regla semantica de COOL, por ejemplo redefinir una clase basica que no esta permitida o redefinir atributos, son reglas sematicas que COOL no permite. `SemanticError: Attribute x is multiply defined in class.`

Uso:

Una vez se tiene el AST que devuelve la etapa sintactica, este se le pasa a un metodo que se encarga de llamar al programa y validar este, luego cada validacion se hace desde los hijos hacia los padres.

```
lexer = CoolLexer()
parser = CoolParser(lexer = lexer)
program = parser.parse(tokens) #Esto devuelve el AST del programa
validate_program(program) #Esto valida al programa, e internamente el programa valida desde sus hijos a los padres recursivamente
```

2.5 Generacion de codigo

Para la generacion de codigo se valida se le pasa el AST del codigo escrito en COOL. El programa de CIL se encarga de generar codigo en CIL modificado. La generacion de codigo en CIL se hace linea por linea, y luego para convertir de CIL a MIPS tambien se genera linea por linea. Todo el control del codigo se hace a nivel de CIL. El proceso de pasar desde CIL para MIPS solo procesa linea por linea por linea el codigo en CIL y escribe el correspondiente en MIPS.

Los procesos mas importantes de la genracion de codigo podrian dividirse en las siguientes implementaciones:

- Control de la pila para el paso de parametros, guardar temporales, guardar variables de los scopes del let y otros procesos que necesitaron de esto.
- Reservacion de memoria para la creacion de instancias de clases.
- Parte estatica de las clases en la . `data`.
- Guardar datos en posiciones de memoria para los atributos de las clases.
- Acceder a los metodos de las clases de forma dinamica, usando la parte estatica de cada uno de estos.
- Simulacion de herencia para el caso de los case.

Control de la pila

La pila es utlizada siempre que se abre un nuevo scope, en le caso del let se guarda en la pila cada una de las variables que se van declarando, para saber las posiciones en pila de cada variable se usan diccionarios en pyton. El diccionario llamado `scope:dict[str:int]` controla las posiciones en la pila, si queremos acceder a una variable local entonces pedimos su posicion en pila usando como `key` el id de la variable que se quiere acceder. Cada vez que se reserva espacio en la pila `ReserveStack` se corren las posiciones en el scope sengun el espacio reservado, manteniendo el control de las posiciones en todo momento. Para pasar argumentos a funciones se usa tambien la pila, se guardan todos los argumentos en pila en el orden que esperan los parametros la funcion, siempre en la posicion cero de los parametros se pasa `self`. Antes de llamar a una funcion se guardan los temporales en uso en la pila, dado que es responsbilidad del invocador guardar las variables temporales. En la pila tambien se guardan las direcciones de retorno al que debe regresar un metodo despues de ser llamado, asi en caso de varios llamados dentro del otro se tiene siempre guardada la posicion a la que se debe retornar y en caso d ser sobreescrita en otro llamado, se recupera de la pila cuando sale de aqui. La pila se reserva y se libera, en cada una de estas operaciones se controlan las posiciones en el scope de las variables.

Reservacion de memoria

Se reserva memoria `ReserveHeap` cada vez que se necesita crear una nueva instancia de una clase o cuando se va a guardar un string. Al contrario de la pila, la memoria no se libera, una vez utilizada no se vuelve a liberar ese espacio de memoria. Las instancias de clases poseen de forma contigua en memoria:

- En la posicion 0, una referencia al `typy_name` de la clase.

- En la posicion 4 posee una referencia a la parte estatica de la clase
- A partir de la posicion 8 se encuentran los atributos de la clase de forma continua. Para acceder a los atributos de la clase se usa un diccionario que usa como `key` el id del atributo y tiene como `value` la posicion de este en momoria a partir de una direccion (puntero).

Parte estatica de los tipos

La parte estatica de las clases se encuentra en la seccion `.data` y consta de 3 partes:

- En la posicion 0 se encuentra la lista de herencias de la clase.
- En la posicion 4 se encuentra el espacio que consume una instancia de esta clase en memoria.
- A partir de la posicion 8 se encuentran las referencias a las etiquetas de los metodos de la clase.

```
StaticA: .word A_inherits, 8, A_type_name, A_abort, A_copy
```

La ejecucion de metodos se hace de forma dinamica en caso que no se use un tipo estatico con `@TYPE`. Si se quiere llamr al metodo `type_name` de una instancia de `A`. Se hace lo siguiente:

```
move $t0, instance #donde instance es el registro con la direccion de la instancia
lw $t0, 4($t0) #se carga la parte estatica de la clase que esta e la posicin 4 de la istancia
lw $t0, 8($t0) #8 es la posicion donde se encuentra la referencia a typename
...#paso de parametros
jalr $t0 # se salta a la direccion guardada en el registro $t0
```

Los metodos estan todos igualmente alineados en memoria, es decir, su referencia en la data esta alineada en memoria. Si un metodo es heredado o sobreescrito de otra clase, estos tendran igual posicion relativa en la parte estatica de su tipo.

Guardar y acceder a atributos

Para guardar o acceder a atributos se guarda en un temporal el valor de la instancia, luego se carga o se guarda en las posiciones asociadas a los atributos de cada clase, la posicion de los atributos esta dada por un diccionario `{id:pos}`. De esta forma, similar a como se hace con los metodos se accede a las posiciones de memoria relativas de cada instancia.

```
#se asume que la instancia a la cual se le quieren acceder/modificar atributos esta en la direcion de memoria que apunta $t0

lw $t0, n($t0) #donde n es la posicion relativa en memoria del atributo que se quiere leer
sw $t0, k($t0) #donde k es la posicion relativa en memoria del atributo que se quiere escribir
```

Las posiciones de los atributos heredados siempre es la misma en todos los tipos, cada tipo tiene su propio espacio para cada atributo, y el control de las posiciones es importante en la herencia.

Simulacion de Herencia

Para simular los niveles de herencia se guarda en data una parte asociada a ello, donde en cada posicion se guarda un numero de nivel de herencia relativo. La referencia a esta parte de la memoria que guarda los niveles de herencia relativos para cada tipo se guardan en la posicion 0 de la parte estatica de cada tipo. Asi se accede de forma parecida a los metodos.

```
.data
StaticG: .word G_inherits, 8, G_type_name, G_abort, G_copy
G_inherits: .word -1, -1, -1, 6, -1, 5, 4, 3, -1, -1, 2, 1, -1

.text
...
move $t0, instance #donde instance es el registro con la direccion de la instancia
lw $t0, 4($t0) #se carga la parte estatica de la clase que esta e la posicin 4 de la istancia
lw $t0, 0($t0) #0 es la posicion donde se encuentra la referencia a los niveles de herencia
lw $t0, p($t0) #p es la posicion relativa del tipo con el cual se desea saber que nivel de herencia relativa hay con este y la instanc
...
```

El caso anterior, buscando los niveles de herenca de la clase `G`, esta dado para el siguiente codigo en COOL:

```
class A {};
class B inherits A {};
class C inherits B {};
class D inherits A {};
class E inherits B {};
class F inherits C {};
class G inherits F {};
```

Int	Str	Bool	Obj	IO	A	B	C
-1	-1	-1	6	-1	5	4	3

De esta forma queda organizado el nivel de herencia, -1 indica que no hereda nunk de este tipo, y los numeros positivos indican que tan cerca esta su nivel, donde 1 representa el mismo tipo, 2 representa la erencia directa, 3 representa la herencia de la herencia y asi sucesivamente.

Al igual que los atributos y metodos, las posiciones de los niveles de herencia en memoria son la misma para cada tipo, y se guarda en un diccionario `type:lv`.