Proyecto Final Complementos de Compilación COOL-Compiler.

Autores:

Daniel Reynel Dominguez Ceballos C-411

Abel Antonio Cruz Suárez C-411

Requisistos

- $python\ 3.7$ o superior
- Instalar PLY mediante pip.
- Si se quieren hacer uso de los test automáticos instalar PLY, $pytest$ y $pytest-ordering$, que se pueden. Todos los requerimientos pueden ser instalados ejecutando python -m pip install -r
  requeriments.txt desde la raíz del proyecto.

Uso del compilador

Para probar el compilador debe abrir una terminal y correr en la carpeta src:

```
python3 __main__.py "path del .cl a ejecutar" "archivo donde almacenar codegen"
```

Ejemplo:

```
python3 __main__.py "/media/abelo/Local Disk/4to/Mio/2do Semestre/CMP/cool-compiler-2021/tests/codegen/life.cl" "salida.s"
```

Y en el archivo $salida.s$ tenemos el código $MIPS$ generado, el cual puede ser probado en simuladores como $QtSPIM$ o el propio simulador $MIPS$. Este archivo debe ser una extension $.s$, $.mips$ o $.asm$ si se desea correr en un simulador. Dicha salida se encontrará en cool-compiler-2021/src .

Otra forma para ejecutar el compilador es hacer uso del ejecutable $coolc.sh$ contenido en la carpeta src de la siguientes forma: ./cool.sh "path".

Arquitectura del compilador

El desarrollo de proyecto se dividió en cinco módulos encargados cada uno de llevar a cabo las tareas del compilador. Es por ello que existe el módulo $lexer$, $parser$, $semantic$, $cil\_builder$ y $mips\_builder$. Los dos últimos encargados de la generación de código, dividida en dos fases. La primera donde se genera un lenguaje intermedio que permite realizar el paso del ast de Cool al ast de Cil y la segunda fase es donde se genera el código de $mips$ a partir del lenguaje generado anteriormente.

Para las dos primeras fases se hace uso de las herramientas de construcción de compiladores lex y $yacc$, existentes en el paquete PLY de $python$. Emplear PLY incluye admitir el análisis sintáctico $LALR(1)$, así como proporcionar una amplia validación de entrada, informes de errores y diagnósticos. Estas dos herramientas se complementan perfectamente pues, $lex.py$ proporciona una interfaz externa en forma de función $token()$ que devuelve el siguiente $token$

válido en el flujo de entrada. $yacc.py$ llama a esto repetidamente para recuperar $tokens$ e invocar reglas gramaticales.

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 la sintaxis generalmente se especifica en términos de una gramática BNF(notación de Backus-Naur).

Análisis lexicográfico

Para llevar a cabo esta fase se emplea la herramienta $lex.py$. en el módulo $lexer.py$. Con ella se toma un texto de entrada y se forman los $tokens$ mediante el uso de las expresiones regulares. Para ello se define para cada $token$ una expresión regular encargado de reconocerlo. Ejemplo:

```
def t_EQUAL(self,t):
    r'='
    self.get_column(t)
    return t

def t_PLUS(self,t):
    r'\+'
    self.get_column(t)
    return t

def t_MINUS(self,t):
    r'-'
    self.get_column(t)
    return t
```

```
def t_ID(self,t):
    r'[a-z][a-zA-Z_0-9]*'

    v = str.lower(t.value)
    if v in self.reserved:
        t.type = self.reserved[v]
        t.value = v
    else:
        t.type = 'ID'

    self.get_column(t)
    return t


def t_NUMBER(self,t):
    r'\d+(\.\d+)?'
    self.get_column(t)
    t.value = float(t.value)
    return t
```

Como se puede apreciar en las imágenes anteriores, extraídas del código del proyecto, a la izquierda se observan tres sencillas expresiones regulares para reconocer los operadores de igualdad, suma y resta; formadas únicamente por el símbolo del operador. En la imagen de la derecha vemos expresiones regulares más complejas para reconocer un identificador y un número.

Como se puede apreciar para definir las reglas de reconocimiento de $tokens$ se utilizan las funciones, pues es necesario definir acciones extra cuando se ha reconocido el $token$, como computar su columna, obtener el valor numérico si es un número, entre otras. La regla de expresión regular se especifica en la cadena de documentación de la función. Esta recibe un solo argumento que es una instancia de $LexToken$. Este objeto tiene atributos de $t.type$ que es el tipo de $token$ (como un $string$), $t.value$ que es el lexema (el texto real coincidente), $t.lineno$ que es el número de línea actual y $t.lexpos$ que es la posición del $token$ en relación con el comienzo del texto de entrada. De forma predeterminada, $t.type$ se establece en el nombre que sigue al prefijo $t\_$. Este nombre no es más que un literal con el que llamamos al $token$.

Análisis sintáctico

Para llevar a cabo el proceso de $parsing$ se crea el módulo $parser.py$ y este a su vez hace uso de $yacc.py$, encargado de implementar esta fase en PLY. $yacc.py$ utiliza la técnica de análisis sintáctico conocida como análisis LR o $shift-reduce$. A modo de resumen, el análisis LR es una técnica de $bottom-up$ que intenta reconocer el lado derecho de varias reglas gramaticales. Cada vez que se encuentra un lado derecho válido en la entrada, se activa el código de acción apropiado y los símbolos gramaticales se reemplazan por el símbolo gramatical del lado izquierdo.

El modo en que se emplea esta técnica resulta semejante al ya visto en clases prácticas, es por ello que en cada regla gramática se va construyendo el AST. Este enfoque consiste en crear un conjunto de estructuras de datos para diferentes tipos de nodos de árbol de sintaxis abstracta y asignar nodos a $p[0]$ en cada regla.

A continuación vemos un ejemplo de se usa $yacc.py$:

```
def p_arith(self,p):
    '''arith : arith PLUS term
             | arith MINUS term
             | term'''
    if len(p) ==4 and p[2] == '+':
        p[0] = PlusNode(p.slice[2], p[1], p[3], p.lineno(2), p.slice[2].column)
    elif len(p) ==4 and p[2] == '-':
        p[0] = MinusNode(p.slice[2], p[1], p[3], p.lineno(2), p.slice[2].column)
    else: p[0] = p[1]

def p_term(self,p):
    '''term : term STAR unary
            | term DIV unary
            | unary'''
    if len(p) == 4 and p[2] == '*':
        p[0] = StarNode(p.slice[2], p[1], p[3], p.lineno(2), p.slice[2].column)
    elif len(p) == 4 and p[2] == '/':
        p[0] = DivNode(p.slice[2], p[1], p[3], p.lineno(2), p.slice[2].column)
    else: p[0] = p[1]
```

En este ejemplo, cada regla gramatical está definida por una función donde el $string$ de documentación de esa función contiene la especificación adecuada de la gramática libre de contexto. Las declaraciones que componen el cuerpo de la función implementan las acciones semánticas de la regla. Cada función acepta un solo argumento p que es una secuencia que contiene los valores de cada símbolo gramatical en la regla correspondiente. Los valores de $p[i]$ se asignan a símbolos gramaticales. En el primer caso, estaríamos ante una regla para producir un nodo de suma o resta, en dependencia de el símbolo en $p[2]$ y el tamaño de la regla y en caso contrario se deriva en un término. Abajo se crea la función encargada de manejar la reglas de los términos. En este caso pudiéramos derivar en una multiplicación, división o en un elemento unario. Aclarar que en ambas reglas, para $arith$ y $term$ en caso de que se reconozca un símbolo($+, -, /, *$) estamos asignado un nodo binario de nuestro AST, pero a modo general, para los no terminales, el valor está determinado por lo que se coloca en $p[0]$ cuando se reducen las reglas.

Gramática

```
program : class_list

class_list : def_class SEMI class_list
           | def_class SEMI
```

```
def_class : class TYPE OCUR feature_list CCUR
          | class TYPE inherits TYPE OCUR feature_list CCUR

feature_list : def_attr SEMI feature_list
             | def_func SEMI feature_list
             | empty

def_attr : ID COLON TYPE
         | ID COLON TYPE LARROW expr

def_func : ID OPAR param_list_call CPAR COLON TYPE OCUR expr CCUR

param_list_call : param_list
                | param_list_empty

param_list : param
           | param COMMA param_list

param_list_empty : empty

param : ID COLON TYPE

expr_list : expr SEMI expr_list
          | expr SEMI

declar_list : declar
            | declar COMMA declar_list

declar : ID COLON TYPE
       | ID COLON TYPE LARROW expr

assign_list : case_assign
            | case_assign assign_list

case_assign : ID COLON TYPE RARROW expr SEMI

expr : boolean

boolean : not comparison
        | comparison

comparison : comparison EQUAL boolean
           | comparison LESS boolean
           | comparison LESSEQ boolean
           | arith

arith : arith PLUS term
      | arith MINUS term
      | term

term : term STAR unary
     | term DIV unary
     | unary

unary : factor

unary : NOX unary
```

```
unary : isvoid expr


factor : OPAR expr CPAR

factor : factor DOT ID OPAR arg_list_call CPAR
       | ID OPAR arg_list_call CPAR
       | factor ARROBA TYPE DOT ID OPAR arg_list_call CPAR

factor : atom


atom : let declar_list in expr

atom : while expr loop expr pool

atom : if expr then expr else expr fi

atom : case expr of assign_list esac

atom : OCUR expr_list CCUR

atom : ID LARROW expr

atom : NUMBER

atom : true
     | false

atom : ID

atom : new TYPE

atom : STRING


arg_list_call : arg_list
              | arg_list_empty

arg_list : expr
         | expr COMMA arg_list

arg_list_empty : empty
```

Análisis Semántico

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. Con $TypeCollector$ se verán solo las declaraciones de clases, para guardar todos los tipos definidos en el lenguaje.

Para ellos nos apoyaremos de un **contexto**, en el
cual se guardaran los tipos de las clases a medida que las vamos visitando, por tanto, el código del $Collector$ consiste en crear los tipos necesario según se va pasando por las clases. Una vez que se han recolectado todos los tipos se analiza que no exista herencia cíclica. El $TypeCollector$ resuelve también el problema de que en la declaración de una clase X se pueda tener un atributo de tipo Z, donde Z es una clase que se declare mucho después de X, pues el $TypeCollector$ asegura que tenemos los nombres de los tipos cuando vayamos a instaciar atributos y demás.

A continuación se observa como se define la clase $TypeCollector$ (omitiendo las funcionalidades):

```
class TypeCollector(object):
    def __init__(self, errors=[]):
        self.context = None
        self.errors = errors
        self.type_level = {}
        self.BUILT_IN_TYPES = ['Int','String','Bool','Object','SELF_TYPE']

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

    @visitor.when(ProgramNode)
    def visit(self, node):
        self.context = Context()
        self.context.create_type('SELF_TYPE')

        for def_class in node.declarations:
            self.visit(def_class)

        def get_type_level(typex):


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

La función get_type_level es la empleada para determinar si herencia cíclica.

Luego de realizar el recorrido y haber encontrado todos los tipos, se procede a ejecutar una segunda pasada por el AST. En este caso se pasará por cada tipo encontrado para construirlo junto con sus atributos y métodos. A este visitor se le pasa el contexto generado por el $TypeCollector$. 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.

Finalmente en el último recorrido del AST, el $TypeChecker$ 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 $scope$, 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 $scope$ 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 acabo el polimorfismo en el lenguaje, pues junto con el $contexto$ se verifica que un tipo pueda ser sustituido por otro, además que se es esencial para determinar atributos que se definen en clases padre y se utilizan en una clase hijo, etc.

Generación de código

Idea básica

Para poder realizar la generación de código primero se llevó el AST de $COOL$ a un AST de CIL. Para ello se visitó cada nodo de $COOL$ para rellenar tres listados principales del $ProgramCilNode$:

- dottype: Donde se registran los tipos en cil
- dotdata: Donde se genera todo el data necesario
- dotcode: Donde se general las instrucciones a seguir para la ejecución del programa

Luego, a partir del AST de CIL se generó el código de $MIPS$ mediante la visita a los nodos del AST.

Estrategia seguida para inicializar el código de MIPS

Al empezar la ejecución del programa se tiene un $label\ main$ para empezar a ejecutar a partir desde la primera línea de código de $MIPS$ y que sea compatible con varios emuladores (algunos hacían un $jump\ and\ link$ a $main$ mientras que otros no, por lo que queríamos algo compatible con varios emuladores),

Inmediatamente se realiza un $jump\ and\ link$ a $entry$: el cual se encarga de hacer un $Allocate$ para reservar espacio de memeoria para una instancia de $Main$ y luego a su función $INIT$ para inicializar todos sus atributos, pasándole el $self$ (El $self$ es la dirección de memoria de la instancia que se obtuvo al realizar el $Allocate$ de la instancia en la que se está actualmente) como primer parámetro.

Al terminar esa inicialización se llama a la $function\_main\_at\_Main$ para empezar a ejecutar el código correspondiente de $COOL$ (En este proceso hemos simulado un $(newMain).main()$ inicial, pasándole a la función $main$ el $self$ correspondiente como parámetro.

Manejo de memoria

Para cada tipo de $Cool$ tenemos en la sección de $.data$ un $label$ a partir del cual guardamos la siguiente información:

Estrategia usada para guardar la información de un tipo en $mips$:

```
data_0: .asciiz "Hello"
data_1: .asciiz "World"
data_2: .asciiz " "
data_3: .asciiz "Cool!"
```

Tenemos para cada tipo un $data\_adress$, que es donde se encuentra la información correspondiente a un tipo. Para acceder a esta dirección utilizamos el $label$ $TypeName\_methods$. En cada instancia de un tipo está guardada este $data\_adress$ y se utiliza para acceder a cualquier información que es común para el tipo. Como por ejemplo la dirección a la que debo saltar para ejecutar una de las funciones, su $type\_name$, el $data\_adress$ de su padre, esto sin tener que replicar esta información en cada instancia de la memoria del tipo correspondiente. La idea que seguimos es que en cada instancia de un tipo se le guarda la dirección de memoria donde se encuentra su $data\_adress$.

Si tenemos la dirección de memoria del tipo, podemos acceder a su información o sus funciones a partir de sumar múltiplos de 4 a esa dirección para acceder el atributo correspondiente ($offset$). A continuación se explicará que información representa cada para cada offset.

- En $data\_adress$ con offset 0 tendremos un número que representa la cantidad de espacio en memoria que se necesita para representar una instancia del tipo que representa (ese tamaño es la cantidad de atributos*4 para la dirección de memoria de cada uno de sus atributos $+4$ bytes para guardar la dirección de memoria donde se encuentra el $address$ correspondiente al tipo actual.
- En $data\_adress$ con offset 4 se guarda la dirección de memoria donde se encuentra el inicio del $string$ que devolvería la si hiciese un llamado a $type\_name$ en esa isntancia.
- En $data\_adress + 8$ se guarda el $data\_adress$ que contiene la información del tipo que es padre del tipo actual, esto nos permitirá más adelante en un $caseNode$ encontrar cual es el tipo más cercano al de la instancia actual entre todos los $branch$
- A partir del $data\_address + 12$ se encuentran labels a las funciones correspondientes al tipo que se representa. Estas funciones cumplen las siguientes propiedades:
    - Están ordenadas de la misma forma que en el padre, con las funciones del padre primero y las creadas nuevas por la clase actual después(ejemplo más adelante).
    - Si se redefine una función del padre, el nombre del label de la función correspondiente en el tipo actual será el nombre de la función redefinida.

Para entender bien el funcionamiento podemos ver el siguiente ejemplo en Cool:

Sean A y B clases tal que B hereda de A

```
class A
    {
        f():Int
        {
            1
```

```
        };
        g():Int
        {
            2
        }
    };
class B inherits A
    {
        h():Int
        {
            3
        };
        f():Int
        {
            4
        };
    };
```

A define las funciones:
f( ):Int (Nueva)

g( ):Int (Nueva)

B define las funciones:

h( ):Int (Nueva)

f( ):Int (Redefinida)

En data aparecerá de la siguiente forma:

```
#TYPES
type_Object: .asciiz "Object"
Object_methods:
.word 4
.word type_Object
.word 0
.word function_abort_at_Object
.word function_type_name_at_Object
.word function_copy_at_Object

type_IO: .asciiz "IO"
IO_methods:
.word 4
.word type_IO
.word Object_methods
.word function_abort_at_Object
.word function_type_name_at_Object
.word function_copy_at_Object
.word function_out_string_at_IO
.word function_out_int_at_IO
.word function_in_string_at_IO
.word function_in_int_at_IO

type_Int: .asciiz "Int"
Int_methods:
.word 8
.word type_Int
.word Object_methods
```

Nótese como en B se conservan las funciones de A en el mismo orden, y en caso de redefinir alguna función (En este caso la función f de A) se conserva en B en la misma posición que tenia en A, pero con el label dirigido a la implementation de esa función en B. Esta propiedad nos sera muy útil mas adelante.

Para guardar los $string$ que se declaran desde $COOL$ se guarda en data ese $string$ y cuando sean necesario se accede a ellos desde la sección correspondiente del código(se verá más adelante) .

```
class Main inherits IO
{
    main(): IO
    {
        {
            out_string("Hello");
            out_string("World");
            out_string(" ");
            out_string("Cool!");
        }
    };

};
```

Este ejemplo genera la data:

```
type_A: .asciiz "A"
A_methods:
.word 4
.word type_A
.word Object_methods
.word function_abort_at_Object
.word function_type_name_at_Object
.word function_copy_at_Object
.word function_f_at_A
.word function_g_at_A

type_B: .asciiz "B"
B_methods:
.word 4
.word type_B
.word A_methods
.word function_abort_at_Object
.word function_type_name_at_Object
.word function_copy_at_Object
.word function_f_at_B
.word function_g_at_A
.word function_h_at_B
```

Se tiene también una dirección para mensajes auxiliares que se desean mostrar, como los de errores, el mensaje que se muestra en el método $abort()$,el $string$ vacío que se inicializa por default, un espacio de memoria de 1028 bytes para guardar los posibles 1024 caracteres de entrada que puede tener un $string$ y para cada tipo que se declara, el $string$ que devuelve su función $type\_name()$.

En la sección de data se genera un $label\ void\_data$ (en esa dirección hay un valor de 0 que no será importante, ya que algo es $void$ si su valor es la dirección de $void\_data$) , esto se utiliza para interpretar que si alguien tiene tipo dinámico $void$, su valor será el la dirección de este label en memoria. (Se utiliza para lanzar $Runtime\ Error$ en caso de que se haga un llamado a partir de

una instancia de $void$, o que la expresión del $case$ de tipo dinámico $void$).

Representación de tipos en memoria: Instancias

Para representar tipos en memoria se utiliza la siguiente estrategia:

Sea n la cantidad de atributos de un tipo A
Para representar una instancia del tipo A en memoria se liberan en el heap $(n+1)*4$ bytes de espacio. (Para verificar si se acabó el espacio del $heap$ comparo la posición de este con la del stack pointer, si el heap superó al stack pointer lanzó $Runtime\ Error: Heap\ Overflow$)

A continuación según el tipo que se quiere instanciar se guarda en el fondo del espacio liberado en el $heap$ para representar al tipo actual (o sea en la posición que se devuelve al hacer $syscall$ en $MIPS$)
O sea guardo la dirección del $label$: $ClassName\_methods$

attr5
attr4
attr3
attr2
attr1
A_methods

En este caso a una instancia de la clase A se le guardó el $data\_adress$ a la información de su tipo y luego cada atributo un $offset$ distinto múltiplo de 4. (En memoria se le colocan todos sus atributos, eso incluye los de sus padres en el orden en que se declaran desde abajo hacia arriba, o sea primero los de su mayor ancestro (el primero de abajo hacia arriba es el 1ro de su mayor ancestro), después los de su 2do mayor ancestro y así sucesivamente, por ultimo los suyos)

8	g()_A
4	f()_A
0	A_methods

12	h()_B
8	g()_A
4	f()_B
0	B_methods

Estrategia de ordenamiento de funciones y atributos en tipos: Aplanamiento

Mientras se crea el AST de CIL se crean dos diccionarios, en uno a cada tipo de $COOL$ le asigno las funciones correspondientes en el orden que se planteó anteriormente.

Luego se crea otro diccionario donde a cada tipo de $COOL$ se le asignan todos los nodos de declaración de atributos que tiene el tipo actual adicionándole el correspondiente de sus padres.

Con estas estrategias se asegura como aplanar todos los tipos de forma tal que estos tengan acceso a todos sus atributos y las inicializaciones de estos, incluyendo los atributos de los padres. También coloco a casa tipo las funciones correspondientes.

Para realizar este procedimiento se utiliza siguiente idea:
Dígase que A es un tipo correctamente plano si A tiene todos los atributos de su padre en orden, seguido de los propios atributos que define A, con cada atributo teniendo la inicialización que se declara en el padre.
El objetivo del algoritmo es tener todos los tipos correctamente planos

Dado un tipo A en $COOL$ que hereda de un tipo B en $COOL$, si el tipo B es correctamente plano, adicionar los atributos de B con sus inicializaciones correspondientes seguido de los atributos de A, obtengo A', el cual es el tipo A correctamente plano

Para aplanar todos los atributos de forma correcta se parte del tipo $Object$, el cual por definición es correctamente plano, ya que no tiene padre. Luego visito todos los hijos de $Object$ aplicando el método de aplanar, que consiste en colocar los atributos de $Object$ en cada uno de sus hijos, lo cual hace que sus hijos sean correctamente planos, realizo el algoritmo recursivamente en cada uno de los hijos y de esta forma se tienen todos los tipos de $COOL$ correctamente planos

Para aplanar funciones se sigue una estrategia similar, solo que esta vez cuando tomo una función del padre en vez de colocar el como función en el diccionario primero reviso si mi tipo actual redefine esa función y en caso de ser positivo redefino la actual.

Con esto se tienen no solo los atributos con sus inicializaciones correspondientes, sino también un offset para las funciones al cual puedo acceder según el tipo estático de la expresión actual a la función dinámica correspondiente. De esta forma puedo acceder a las funciones correspondientes y los atributos del tipo dinámico a partir del estático(ambos tienen el mismo offset)

En este ejemplo tenemos la clase B que hereda de A con su representación del offset en su $data\_adress$. Como podemos observar si se tiene una expresión que tiene tipo estático A y llama a la función f() de offset 4, yo puedo decirle a un tipo dinámico B que llame a su función f() que esta en el offset 4 sin saber cual es su tipo dinámico, pues siempre se cumple que tipo dinámico $\leq$ Tipo estático y por tanto puedo saber el offset de su función durante la generación del código y el $address$ desde el cual accedo a su offset esta en su instancia en memoria, por tanto puedo acceder a ella fácilmente durante la ejecución.

0	A_methods
4	f()_A
8	g()_A

0	B_methods
8	f()_B
4	g()_A
12	h()_B

Nótese que esta información se encuentra en . $data$, lo que se coloca en cada instancia es la dirección de $A\_methods$ y $B\_methods$

Liberar espacio en memoria para cada tipo: Allocate

El nodo $Allocate$ de CIL genera un código en $mips$ correspondiente a liberar un espacio en memoria en el $heap$ en base a un tipo estático determinado (esta operación se realiza durante una instanciación o cuando genero tipos básicos como Int,$String$ o $Bool$) y guarda en una variable local interna esa dirección para utilizarla posteriormente. Luego guarda la dirección con la información del tipo en la instancia.

En caso de llamados a funciones a partir de una expresión, esa dirección a la que se le hace $Allocate$ es lo que recibe la función como $self$.

Funciones INIT para la inicialización de tipos en memoria

Para cada tipo de $COOL$ existe una función $INIT\_TypeName$ a la cual se le hace un llamado cada vez que se desea crear una nueva instancia de ese tipo. Esta función se encarga de inicializar cada uno de los atributos del tipo empezando por los del padre. Recibe como parámetro un $self$ que sería la dirección de memoria de la instancia que se esta inicializando, esta dirección de memoria se obtiene antes gracias al $Allocate$ (Por este motivo siempre que se hace un $INIT$ se realiza antes un $Allocate$ y se le pasa el resultado como parámetro $self$ al $INIT$)

Para realizar los llamados a funciones se utiliza la siguiente estrategia:

Se tiene un nodo $CallNode$ que genera el código de MIPS que $pushea$ los parámetros a la pila y darles sus valores correspondientes calculados, luego se tiene un $FunctionNode$, que es el cuerpo de la función, pero también realiza la tarea de colocar sus variables locales internas en la pila y colocar en el tope el $return\ adress$

Cada función tiene un diccionario de variables que dado un parámetro o una local interna devuelve el offset de dicha variable, de esa forma se puede saber que local interna o parámetro se usa en cada momento, la forma en que se representa en la pila es la siguiente:

Sea la función $f(self, x : Int, y : Bool)$ que utiliza 3 $local\ internal$ de nombres: L1,L2 y L3. (El $self/expresion$ correspondiente se le pasa al nodo que se encarga de generar el código de llamar a la función)

Al realizar un llamado a una función la pila crece y decrece como se verá a continuación:

Estado Inicial:

SP	Pila
100	
96	
92	
88	
84	
80	
76	
72	

Al hacer el $CallNode$

SP	Pila
100	y
96	x
92	self
88	
84	
80	
76	
72	

Al hacer el $FunctionNode$

SP	Pila
100	y
96	x
92	self
88	L3
84	L2
80	L1
76	$ra
72	

Al terminar $FunctionNode$

SP	Pila
100	y
96	x
92	self
88	
84	
80	
76	
72	

Al terminar $CallNode$

SP	Pila
100	
96	
92	
88	
84	
80	
76	
72	

Al finalizar las operaciones que realiza la función se guarda el resultado esperado en $\$s0$, se carga el $return\ adress$ que se guardo en el tope de la pila y se devuelve la pila al estado en que estaba antes de entrar al cuerpo de la función, luego se hace $jr\ \$ra$ para regresar a $CallNode$

En el cuerpo del $CallNode$ ahora se atrapa el valor que se encuentra en el registro $s0 donde se guardó el resultado de la función y se retorna la pila a su estado original antes de hacer el llamado

Nótese que el estado de la pila se mantiene, ya que todo el espacio que se libera al hacer el $CallNode$ y resolver el cuerpo de la $FunctionNode$ se recupera al finalizar cada uno y el resultado que estaba en $s0 ahora esta en la pila en la posición representada por la variable local correspondiente que se le paso al nodo para guardar el valor.

Explicación de ideas de algunos de los nodos que consideramos especial mención

CaseNode:

Para calcular el $branch$ que se debe ejecutar se busca el tipo estatico que tiene menor distancia del tipo dinamico que de

LetNode:

Para las variables que se declaran en el let tenemos un $scope$ especial que agrega dichas variables, pero que a diferencia del de $COOL$ no contiene los parámetros de funciones o los atributos de los tipos, esto es útil porque si una variable no se encuentra en el $scope$ especial del let entonces es el parámetro de una función o un atributo del tipo actual.

GettAtributeNode y SetAttributeNode:

Este nodo dado un tipo estático de una clase, su instancia y el nombre del atributo que estoy buscando devuelve en una variable local interna que se le pasa el valor del atributo de dicha clase, puedo utilizar el tipo estático para buscar el offset de un atributo por el hecho de que los atributos están aplanados correctamente.

Tipos Base y su representación en memoria:

- Int: Se representa en memoria como cualquier tipo, se le asigna un atributo $value$, que guardará el verdadero valor del entero. Para comparar con otro Int basta con buscar en su instancia con offset 4 para atrapar el valor.
- $Bool$: Funciona igual que el entero, solo que en su instancia de memoria se guarda la dirección correspondiente a las propiedades y funciones del t ipo$Bool$

value	4
type_addr	Int_methods

value	8
type_addr	Int_methods

value	1
type_addr	Bool_methods

- $String$: Se representa con 2 atributos en memoria: el value con $offset = 4$ como la dirección real donde se encuentran las letras del $String$ en memoria y un atributo Len con el $Length$ del $String$

Len	12
Value	str_location
Type_addr	String_methods

$str\_location$ es una dirección en memoria que puede provenir de $data$ o de algún lugar del $heap$ y que contiene la siguiente información:

H
E
L
L
O
“ ”
W
O
R
L
D
¡
ZERO

$str\_location$ apunta a la dirección de memoria con el valor numérico que representa la H. (ZERO es una forma visual que se utilizó para representar a 0 como valor)