COOL-COMPILER-2021

Reinaldo Barrera Travieso - C411

1 Orientación

1.1 COOL: Proyecto de Compilación

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.

1.2 Sobre el Lenguaje COOL

Ud. podrá encontrar la especificación formal del lenguaje COOL en el documento "COOL Language Reference Manual", que se distribuye junto con el presente texto.

2 Uso del compilador

El presente compilador fue desarrollado en el sistema operativo Ubuntu 20.4, donde se utilizo Python 3.8.10 como lenguaje de programación.

El proyecto en general tienen los siguientes requerimientos iniciales para su ejecución:

- ply
- pytest
- pytest-ordering

Para instalar todas las dependencias basta con ejecutar el siguiente comando en el directoria de raíz:

```
$ pip install -r requirements.txt
```

En particular como todo el proyecto fue realizado en python3, también cabe destacar que para los test de prueba se debe emplear la herramienta pytest3. Por otra parte, para instalar las dependencias también se debe usar el comando pip3.

NOTA: El compilador no fue probado en versiones anteriores de python por lo que no se tiene conocimientos de su correcto funcionamiento.

2.1 Estructura del proyecto

```
.
├── CIL
│   ├── ast.py
│   └── cil.py
├── coolc.sh
├── main.py
├── makefile
├── MIPS
│   └── mips.py
├── Parser
│   ├── ast.py
│   ├── lexer.py
│   ├── parser.out
│   ├── parser.py
│   └── parsetab.py
├── Semantic
│   ├── builder.py
│   ├── checker.py
│   └── collector.py
└── Tools
    ├── context.py
    ├── errors.py
    ├── messages.py
    ├── scope.py
    ├── tokens.py
    ├── utils.py
    └── visitor.py
```

2.2 Compilando su proyecto

Una vez dentro de la carpeta cool-compiler-2021 se abre una terminal y se ejecutan los siguientes comandos :

```
$ cd src
$ make clean
$ make
```

El primer comando es el encargado de cambiar el directorio actual al directorio source, en donde se encuentra el fichero principar para ejecutar el compilador. Este comando puede ser ignorado si la terminal se abre desde el directorio source.

El segundo comando el el encargado de limpiar toda las basura y residuos que pueden quedar dentro de la estructura de ficheros de las ejecuciones anteriores.

Por último el tercer comando es el encargado de compilar el resultado final, para este caso este comando es innecesario pues la ejecución del compilador no depende de una compilación previa, pues este se ejecuta directamente desde consola con python. Por este motivo este comando solo mostrara un mensaje simbólico al usuario.

2.3 Ejecutando su proyecto

En el proyecto se incluye el archivo */src/coolc.sh* el cual es el encargado de ejecutar el compilador. Este muestra inicialmente un mensaje con el nombre del compilador y los nombres de los autores, seguido de esto se pasa a ejecutar las instrucciones pertinente. Al ejecutar este archivo se debe pasar un argumento que seria el nombre del fichero que se quiere compilar. Para un mejor visualización en la terminal se debe ejecutar el siguiente comando:

```
$ ./coolc.sh <input_file.cl>
```

El argumento *<input_file.cl>* especifica el nombre del fichero, y como se trata de un compilador del lenguaje COOL la extensión de este siempre debe ser un .cl. Una vez terminado el proceso de compilación la salida de este es otro fichero de igual nombre pero ahora con la extensión .mips.

Otro forma de ejecutar el compilador es mediante el siguiente comando:

```
python3 main.py <input_file.cl> <output_file.mips>
```

Si se fijan en este caso, se debe pasar tanto el fichero de entrada como el fichero de salida, lo cual puede traer algunas consecuencia a la hora de pasar incorrectamente los argumentos. Para el caso del fichero de entrada puede suceder que no se escriba bien el nombre del mismo o que se le de una extensión no permitida, lo cual puede terminal en errores del programa a la hora de la ejecución.

2.4 Comprobando resultado

Una vez finalizado el proceso de compilación obtenemos un fichero de salida done se encuentra todo el código en ensamblador. Para poder comprobar el corrector funcionamiento de este se puede ejecutar algún simulador de código MIPS. Como sugerencia de dicho simulador esta spim, el cual trabaja en una arquitectura de 32bit para Ubuntu. Para ejecutar dicha simulación basa con ejecutar los siguientes comando en una terminal:

```
# Comando para instalar el simulador
$ sudo apt install spim

# Comando para ejecutar el simulador
$ spim -file <output_file.mips>
```

Para el caso que se quiera ver de una forma visual la simulación se debe ejecutar el siguiente comando:

```
$ xspim -file <output_file.mips>
```

2.5 Tests

El proyecto consta con un conjunto de test que realizan una serie de comprobaciones para cada una de las fases del compilador. Para los test de prueba se puede ejecutar el siguiente comando:

```
$ make test
```

Para este caso se utilizo *pytest3,* en el caso de que se quiera ejecutar con *pytest2* solo basta con ejecutar el comando siguiente:

```
$ make test-2
```

3 Arquitectura del compilador

El compilador esta formado por tres fase principales, las cuales a su vez están divididas en subfases las cuales permiten una estructuración del trabajo final. La construcción del compilador es de forma escalonada ya que cada fase necesita recursos de las fases anteriores. Por tanto, el compilador queda estructurado de la siguiente forma:

- Análisis sintáctico
  - Análisis léxico
  - Parsing
- Análisis semántico
  - Colector de tipos
  - Constructor de tipos
  - Colector de variables
  - Chequeo semántico
- Generación de código
  - Generación de código intermedio(CIL)
  - Generación de código ensamblador(MIPS)

Cada una de estas fases se dividen en módulos independientes ubicados en *./src*, que se integran en el archivo principal *./src/main.py*

3.1 Análisis sintáctico

El 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. Para ambos casos se empleó la biblioteca .PLY de python.

PLY consta de dos módulos separados; *lex.py* y *yacc.py*, ambos de los cuales se encuentran en un paquete de Python llamada capa. El módulo *lex.py* se utiliza para romper texto de entrada en una colección de fichas especificadas por una colección de reglas de expresiones regulares. *yacc.py*

se utiliza para reconocer la sintaxis del lenguaje que se ha especificado en el formulario de una gramática libre de contexto. yacc.py utiliza análisis sintáctico LR y genera sus tablas de análisis sintáctico utilizando el LALR(1) (por defecto) o algoritmos de generación de tabla SLR .

Las dos herramientas tienen el propósito de trabajar juntos. Específicamente, *lex.py* ofrece una interfaz externa en forma de una función de señal que devuelve el siguiente token válido en el flujo de entrada. *yacc.py* llama a esto varias veces para recuperar tokens e invocar las reglas gramaticales. La salida de *yacc.py* es a menudo un árbol de sintaxis abstracta (AST). Sin embargo, esto es totalmente en manos del usuario. Si se desea, *yacc.py* también se puede utilizar para implementar simples compiladores de una pasada.

Fuente: https://escompiladores.wordpress.com/2014/03/11/ply-una-implementacion-de-lex-y-yacc-en-python/

3.1.1 Análisis léxico

En esta fase se procesa el programa fuente de izquierda a derecha y se agrupan en componentes léxicos (tokens) que son secuencias de caracteres que tienen un significado. Todos los espacios en blanco, comentarios y demás información innecesaria se elimina del programa fuente. El lexer, por lo tanto, convierte una secuencia de caracteres (strings) en una secuencia de tokens. Después de determinar los caracteres especiales de COOL, especificados en su documentación, se deciden las propiedades necesarias para representar un token. Un token tiene un lexema, que no es más que el string que se hace corresponder al token y un tipo para agrupar los que tienen una característica similar. Este último puede ser igual o diferente al lexema: en las palabras reservadas sucede que su lexema es igual a su tipo, mientras que en los strings y en los números, estos son diferentes. Mientras que un token puede tener infinitos lexemas, los distintos tipos son predeterminados. Además, para el reporte de errores se guarda la fila y la columna de cada token.

La construcción del lexer se realiza mediante las herramientas de PLY. Para su construcción es necesario la definición de una variable tokens que es una lista con los distintos tipos de tokens. Después se especifican cuáles secuencias de caracteres le corresponderán a cada tipo de token, esto se especifica mediante expresiones regulares. Para cada tipo de token se definió una función mediante la convención de PLY de nombrar t_tipo , donde tipo es el tipo de token, y en el docstring la expresión regular que lo describe.

Símbolos terminales o tokens (Palabras claves)

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

Nota: 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.

Otros símbolos terminales o tokens

```
[;] - semi [:] - colon  [,] - comma [.] - dot [(] - opar
[)] - cpar [{] - ocur [}] - ccur [<-] - larrow [@] - arroba
[=>] - rarrow nox [=] - equal [+] - plus [-] - minus [*] - star
[/] - div [<] - less [<=] - lesseq [<id>] - id [<Type>] - type
[<123..>] - num ["abc..."] - string'
```

Nota: Todos los símbolos presentados anteriormente 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.

3.1.2 Parsing

El proceso de parsing consiste en analizar una secuencia de tokens y producir un árbol de derivación. Por lo tanto, se comprueba si lo obtenido en la fase anterior es sintácticamente correcto según la gramática del lenguaje.

El parser también se implementó mediante PLY, especificando la gramática y las acciones para cada producción. Para cada regla gramatical hay una función cuyo nombre empieza con p_ . El docstring de la función contiene la forma de la producción. PLY usa los dos puntos (:) para separar la parte izquierda y la derecha de la producción gramatical. Los símbolos del lado izquierdo de cada función son llamados no-terminales y el primero es considerado el símbolo inicial. El cuerpo de esa función contiene código que realiza la acción de esa producción. Al lado derecho pueden aparecer terminales(tokens), no-terminales, una combinación de ambos o epsilon. En el caso especial del símbolo epsilon se refiere a una producción en blanco. En la practica estas producción en blanco se emplea para los casos donde la aparición de token puede ser opcional.

En cada producción se construye un nodo del árbol de sintaxis abstracta. El parámetro p de la función contiene los resultados de las acciones que se realizaron para parsear el lado derecho de la producción. Se puede indexar en p para acceder a estos resultados, empezando con p[1] para el primer símbolo de la parte derecha. Para especificar el resultado de la acción actual se accede a p[0] . Así, por ejemplo, en la producción program : class_list construimos el nodo ProgramNode conteniendo la lista de clases obtenida en p[1] y asignamos ProgramNode a p[0]

Función de una regla gramatical en PLY

```
def p_program (self, p):
   'program : class_list'
    p[0] = ProgramNode (p[1])
```

El procesador de parser de PLY procesa la gramática y genera un parser que usa el algoritmo de shift-reduce LALR(1), que es uno de los más usados en la actualidad. Aunque LALR(1) no puede manejar todas las gramáticas libres de contexto, la gramática de COOL usada fue refactorizada para ser procesada por LALR(1) sin errores.

La gramática especificada en el manual de COOL fue reconstruida para eliminar cualquier ambigüedad y teniendo en cuenta la precedencia de los operadores presentes. A continuación se muestra la gramática que consiste la base de este compilador.

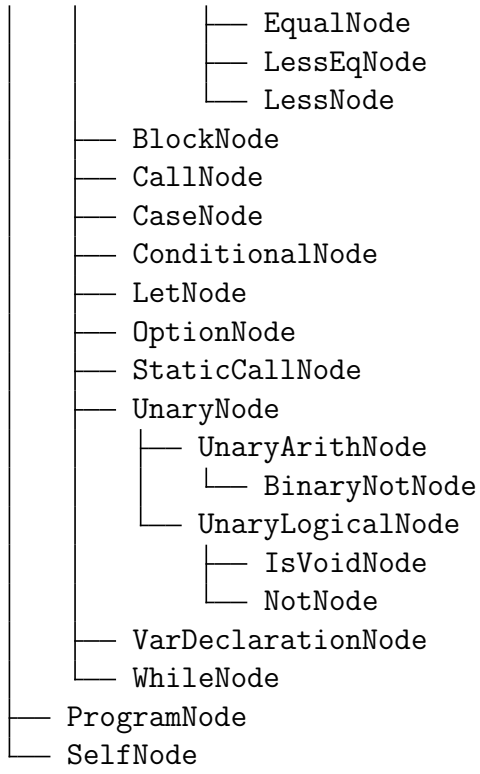
```
program          --> class_list
class_list       --> def_class class_list
                   | def_class
def_class        --> class type ocur feature_list ccur semi
                   | class type inherits type ocur feature_list ccur semi
feature_list     --> def_attr semi feature_list
                   | def_func semi feature_list
def_attr         --> id colon type
                   | id colon type larrow expr
def_func         --> id opar formals cpar colon type ocur expr ccur
formals          --> param_list
                   | epsilon
param_list       --> param
```

```
                  | param comma param_list
param             --> id colon type
let_list          --> let_assign
                  | let_assign comma let_list
let_assign        --> param larrow expr
                  | param
cases_list        --> casep semi
                  | casep semi cases_list
casep             --> id colon type rarrow expr
expr              --> id larrow expr
                  | comp
comp              --> comp less op
                  | comp lesseq op
                  | comp equal op
                  | op
op                --> op plus term
                  | op minus term
                  | term
term              --> term star base_call
                  | term div base_call
                  | base_call
base_call         --> factor arroba type dot func_call
                  | factor
factor            --> atom
                  | opar expr cpar
                  | factor dot func_call
                  | not expr
                  | func_call
                  | isvoid base_call
                  | nox base_call
                  | let let_list in expr'
                  | case expr of cases_list esac
                  | if expr then expr else expr fi
                  | while expr loop expr pool
atom              --> num
                  | id
                  | new type
                  | ocur block ccur
```

```
                  | true
                  | false
                  | string'
block            --> expr semi
                  | expr semi block
func_call        --> id opar args cpar
args             --> arg_list
                  | empsilon
arg_list         --> expr
                  | expr comma arg_list
```

Para el (AST) se definió una serie de objetos que forma una jerarquía, donde se consta de un nodo raíz a partir del cual se van construyendo los siguientes nodos. Cada nodo posee características e información diferente.

```
Node
├── DeclarationNode
│   ├── AttrDeclarationNode
│   ├── FuncDeclarationNode
│   └── mkdirDeclarationNode
├── ExpressionNode
│   ├── AssignNode
│   ├── AtomicNode
│   │   ├── ConstantBoolNode
│   │   ├── ConstantNumNode
│   │   ├── ConstantStrNode
│   │   ├── ConstantVoidNode
│   │   ├── InstantiateNode
│   │   ├── TypeNode
│   │   └── VariableNode
│   ├── BaseCallNode
│   ├── BinaryNode
│   │   ├── BinaryArithNode
│   │   │   ├── DivNode
│   │   │   ├── MinusNode
│   │   │   ├── PlusNode
│   │   │   └── StarNode
│   │   └── BinaryLogicalNode
```



3.2 Análisis semántico

El objetivo del análisis semántico es validar que los predicados semánticos se cumplan. Para esto construye estructuras que permitan reunir y validar información sobre los tipos para la posterior fase de generación de código. Todo esto se puede lograr haciendo varios recorridos por cada uno de los nodos de AST obtenido en la fase anterior. No siempre estos recorrido van a en todos los nodos, pues en algunos casos vasta con hacer un recorrido superficial para adquirir la información necesaria y teniendo en cuenta que siempre se van a ir recolectando los errores que se van detectando.

Para la fase semántica se crearon varias estructuras con ciertas funcionalidades que nos permite almacenar y obtener información durante todo este proceso. Estas estructura no solo tiene uso en esta fase si no que más adelante, en la siguiente fase seguiremos obtenido información de ellas.

- **Attribute:** Esta clase no va ha permitir almacenar toda la información

acerca de un atributo. *./src/Tools/context.py*

- **Method:** Esta clase no va ha permitir almacenar toda la información acerca de un método. *./src/Tools/context.py*
- **Type:** Esta clase no va ha permitir almacenar toda la información acerca de un tipo. También existen definidos los tipos básico del lenguaje que heredan directamente de la clase Type. *./src/Tools/context.py*
- **Context:** Esta clase no va ha permitir almacenar todas los tipos que existen en nuestro programa gracias a un diccionario el cual usa los nombre de las clases como llave. *./src/Tools/context.py*
- **Scope:** Esta clase nos va ha permitir almacenar todas las variables que existen en nuestro programa y conocer la extensión de estas en una cierta región del código. *./src/Tools/scope.py*

3.2.1 Colector de tipos

El colector de tipo es el primero y el mas superficial de los recorrido, pues este solo va a profundizar hasta el nodo donde se declaran las clases (ClassDeclarationNode). El objetivo de este es es recolectar los nuevos tipos que fueron definidos en el programa con la ayuda de la clase Context.

Inicialmente se crean todos los tipos por defecto que trae el lenguaje (Object, IO, Int, String, Bool) y después se pasa a visitar cada uno de las clases para construir el tipo especificado.

En este recorrido se van a detectar errores relacionados con la definición o redefinición de tipos.

3.2.2 Constructor de tipos

Para la construcción de tipos se realiza otra pasada por el AST pero ahora hasta los nodos que contienen la información de los atributos y los métodos (AttrDeclarationNode, FuncDeclarationNode). Una vez que se llega a un nodo que define una clase guardamos el tipo en el que nos encontramos y pasamos a visitar los atributos y los métodos, para así poder completar el tipo.

En este recorrido se van a detectar errores relacionado con la definición o redefinición de los atributos y los métodos. También es en este recorrido cuando sabemos si nuestros tipos caen en ciclos a la hora de heredar de otro.

3.2.3 Colector de Variable

Ya a partir de este, los recorridos son profundos, pues en este caso se desea pasar por todo los nodos que corresponden a instrucciones que pueden almacenar variables. Cada vez que encontramos una variable y con la ayuda de la clase Scope, la almacenamos.

Alguno de los errores que se pueden detectar en este recorrido es cuando se llama a una variable que no esta definida o se trata de modificar la variable self.

3.2.4 Chequeo semántico

Este es el último recorrido de esta fase, y por tanto el recorrido más profundo. En el se van a detectar la mayoría y restantes errores. A diferencia a las anteriores pasadas por el AST en esta no se va a almacenar información, pues solo se completara el análisis semántico comprobando que todo el programa esta correcto.

Gracias a que los anteriores recorridos por el AST se completo toda la información sobre los tipos y las variables, en este se pueden detectar errores como la compatibilidad de los tipos. Un ejemplo de esto es saber si la expresión de retorno concuerda con el tipo de retorno del método o si la expresión asignada a un atributo es del mismo tipo con el que fue definido.

3.3 Generación de código

Con el objetivo de generar código de COOL de forma más sencilla se genera además un código intermedio, CIL, ya que el salto directamente desde COOL a MIPS es demasiado complejo.

3.3.1 Código Intermedio

Para la generación de código intermedio de COOL a MIPS se usó el lenguaje CIL, bastante similar al impartido en clases de Compilación. Solamente se añadieron algunas instrucciones, la mayoría de ellas con el objetivo de representar de forma más eficiente los métodos de bajo nivel que existían en los objetos built-in de COOL. Para la generación de CIL se usa el patrón visitor, generando para cada nodo en COOL su conjunto de instrucciones equivalentes en el lenguaje intermedio. Además, para facilitar la generación de código de máquinas nos aseguramos de que todas las variables locales y

las funciones tengan un nombre único a lo largo de todo el programa. Una instrucción que requirió especial interés fue case pues expr0 es evaluado y si su tipo dinámico es C entonces es seleccionada la rama typek con el tipo más especializado tal que C se conforme a type. Para facilitar esta selección, cada una de las ramas i de case fue ordenada de acuerdo a la profundidad que tiene typei con respecto al árbol de herencia que tiene como raíz a Object. Así, los nodos con una mayor profundidad son los más especializados. Luego, se tiene una instrucción especial en CIL para representar conforms, que determina si la clase C se conforma con typek . La primera rama i que sea seleccionada será la que cumplirá con los requerimientos, entonces se asignará $< expr0 > < - < typei >$, esta se evalúa y retorna $< expri >$ La mayoría de los errores en runtime fueron comprobados generando el código CIL correspon diente, para esto se crea un nodo especial ErrorNode, que recibe el mensaje que se generará si se llega a dicho nodo. Por ejemplo, al crear el nodo DivNode , se chequea si el divisor es igual a cero, si lo es se crea un ErrorNode y en otro caso se ejecuta la división. El único error que no es analizado en la generación de código intermedio fue el de Substring out of range, causado al llamarse la función substr de String; este error es comprobado directamente cuando se genera MIPS.

Otra de las particularidades que se tuvo en cuenta para la generación de CIL fue la inicialización de los atributos, tanto los heredados como los propios de la clase. Cuando se crea una instancia de una clase se deben inicializar todos sus atributos con su expresión inicial, si tienen alguna; en otro caso se usa su expresión por defecto. Con el objetivo de lograr esto se creó para cada tipo un constructor, cuyo cuerpo consiste en un bloque, dándole un valor a cada uno de sus atributos. Este es llamado cada vez que se instancia un tipo. Los tipos built-in definidos en COOL, (Object, IO, Int, String, Bool) son definidos desde CIL. Para crear el cuerpo de cada una de sus funciones se hizo necesario la creación de nodos nuevos en CIL. ExitNode y CopyNode para la implementación de abort y copy de Object. Las instrucciones Read y Print fueron sustituidas por ReadInt, ReadString, PrintInt y PrintString con el objetivo de implementar de forma más sencilla las funciones *in_int*, *in_string* , *out_int* y *out_string* de IO respectivamente, y debido a que en MIPS se requiere un trato diferenciado al tipo int y string cuando se hacen llamados al sistema. Para los métodos de String : length, concat y substr se crearon los nodos LengthNode, ConcatNode y SubstringNode. De esta forma, gran parte de las clases built-in se crean directamente en la generación de MIPS para lograr mayor eficiencia y debido a las particularidades de cada

una de las funciones que requieren de un procesamiento especial.

3.3.2 Código de Máquina

Para generar el código de máquinas se usa una vez más el patrón visitor, recorriendo todos los nodos de CIL que fueron creados.

Entre las instrucciones se trata de manera especial la igualdad entre strings, que es realizada comparando carácter por carácter.

Register Allocation:

Uno de los principales problemas en la generación de código es decidir qué valores guardar en cada registro. Con el objetivo de hacer más eficiente dicha asignación se particionaron las instrucciones de cada una de las funciones del código intermedio en Bloques básicos, que son una secuencia maximal de instrucciones que cumplen que:

- El control de flujo solo puede entrar al bloque a través de la primera instrucción del bloque. Es decir, no hay saltos al medio del bloque.
- Se dejará el bloque sin saltos ni llamados de función, estas solo estarán posiblemente en la última instrucción del bloque.

En nuestro caso, en algunos bloques existirán saltos ya que algunas instrucciones de CIL, como Concat , Length , etc usan saltos cuando estos se generan en MIPS; pero como el código de estos nodos es creado directamente, el uso eficiente de los registros y de las instrucciones de MIPS fue garantizado dentro de dichos nodos.

Para formar dichos bloques primero se determina, dada una lista de instrucciones, aquellas que son líderes, que serán las primeras instrucciones de algún bloque básico. Las reglas para encontrar los líderes son:

1. La primera instrucción en el código intermedio es un líder.
2. Cualquier instrucción que siga inmediatamente a algún salto (incondicional o condicional) o alguna llamada a una función es un líder.
3. Cualquier instrucción que es el objetivo de algún salto incondicional o condicional es el líder.

Entonces, para cada líder, su bloque básico consiste en él mismo y en todas las instrucciones hasta, pero sin incluir, el siguiente líder o la última instrucción del código intermedio. Para la asignación de las variables en los registros se usan 2 estructuras auxiliares:

Register descriptor: Rastrea el actual "contenido" de los registros. Es consultado cuando algún registro es necesitado. A la entrada de un bloque, se asume que todos los registros están sin usar, por lo que a la salida cada uno de estos se limpian, guardando su contenido en la memoria.

Address descriptor: Rastrea las localizaciones donde una variable local puede ser encontrada.

Posibles localizaciones: registros y pila. Todas las variables locales son guardadas en la pila ya que en los registros solo estarán de forma temporal. Además se calcula para cada instrucción una variable booleana is_live , que determina si en una instrucción es usada el valor de dicha variable (notar que de cambiarle el valor a la variable esta no se considera usada, su valor es redefinido). Se computa también su next_use, que contiene el índice de la instrución en donde fue usada después de esa instrucción. Para calcular dichas propiedades se recorre el bloque empezando por la última posición, en un principio todas las variables están "muertas" y su next_use no está definido (es null). Para la realización de este algoritmo usamos una estructura auxiliar, SymbolTable, que lleva cual es el último valor next_use y is_live de la variable. Para cada instrucción $x = y$ op z que tiene como índice i se hace lo siguiente:

1. Se le asigna a la instrucción i la información encontrada actualmente en la SymbolTable de x, y y z .
2. En la SymbolTable, le damos a x el valor de "no vivo" y "no next use".
3. En la SymbolTable, le damos a y y a z el valor de "vivo" y al next_use de y y z el valor i

Cada instrucción tiene: out , in1 , in2 para determinar cuáles son los valores que necesitan registros. Así, si alguno de estos campos no es nulo y es ocupado por una variable se procede antes de procesamiento de dicha

instrucción a asignarle registros a cada variable. Para la asignación de un registro se siguen las siguientes reglas:

1. Si la variable está en un registro ya, entonces se retorna ese registro.
2. Si hay un registro sin usar, retorna este.
3. Se elige entre los registros llenos aquel que contenga la variable que sea menos usada en las instrucciones restantes del bloque, salvándose primero en el stack el contenido del registro y se retorna dicho registro.

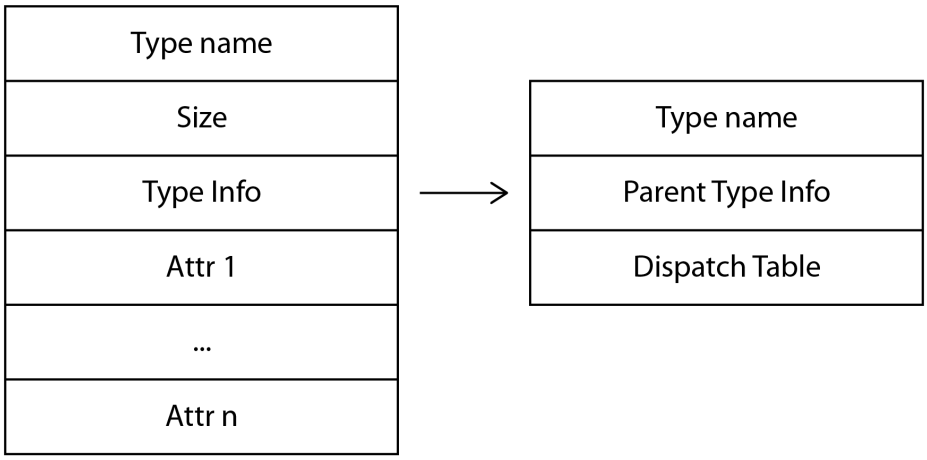
Si se tiene una expresión del estilo out = in1 op in2 , entonces notar que el registro de out puede usar uno de los registros de in1 y in2 , si ninguno de estos operandos son usados en las instrucciones siguientes. Se usa el next_use de estas variables, comprobando que esta no haya sido usada en alguna instrucción siguiente, y si estas están "vivas", es decir, su valor actual es necesitado.

Se tienen disponibles para usarse casi todos los registros de MIPS, excepto: $v0 que es usado como valor de retorno en todas las funciones y para realizar llamadas al sistema, $ra que es contiene la dirección de retorno de las funciones, $a0 que también es usado para llamadas al sistema y $t8 y $t9 que son usados como registros auxiliares para la generación de código.

Representación de los Objetos:

Otra de las problemáticas en la generación de código fue la representación de los objetos en memoria. Se necesita guardar información referente al tipo de objeto, los métodos del objeto, sus atributos y alguna manera de representar su padre. Esto último es necesario para la implementación de ConformsNode.

A continuación se muestra un diagrama de como se representaron dichos objetos en memoria:



Cuando se crean los tipos, para cada tipo se crea una tabla de Type Info , que guarda la información de los tipos que se presenta en la parte derecha de la Figura 2.1. Type Name es una referencia al nombre del tipo, y sirve como identificador del mismo, Parent Type Info guarda una referencia a la tabla de tipos del padre, mientras que Dispatch Table tiene una referencia a una tabla en la que están ubicadas los métodos del objeto tal y como se guardan en CIL: primero los métodos de los padres y luego los de ellos. Los métodos sobreescritos son guardados en la misma posición que ocuparían en el padre.

Luego, cuando se crea un objeto, se calcula para cada uno de ellos el tamaño que ocupará: esto será igual al total de atributos más los 3 campos adicionales usados para guardar información del tipo. En el primer campo se pone el nombre del tipo, luego el tamaño que este ocupa, y después se busca la dirección del Type Info correspondiente al nuevo objeto que se crea. Después son ubicados los atributos primero los del padre y después los definidos por el objeto. Después, para buscar un atributo se calcula el offset de este atributo en su tipo estático, y a este se le sumará 3 porque ahí es en donde se empiezan a acceder a los atributos. Un algoritmo similar es el que se usa para acceder a los métodos: se busca en la posición 3 de la tabla del objeto, donde se obtiene Type Info y desde allí en la posición 3 se accede a la dirección del Dispatch Table , a partir de la cual se busca el índice de la función llamada según la información que se tiene del tipo estático del objeto, luego, esta función es llamada. Notar que en caso de que su tipo estático no coincida con su tipo dinámico no importa, porque en la dirección buscada estará la verdadera dirección del método. Este puede ser diferente

si, por ejemplo, es sobreescrito. A este proceso se le llama Dynamic Dispatch, representado como DynamicCallNode en contraposición con StaticCallNode. En este último se especifica la función a llamar, es decir, no es necesario buscar dinámicamente en la información del objeto, se accede al método de un tipo especificado.

Por otra parte, también requirió especial atención el nodo ConformsNode , usado para determinar si el tipo de una expresión se conforma con un typen. Para calcular esto se accede al Type Info del objeto y a partir de ahí, se realiza un ciclo por los Parent Type Info hasta encontrar alguno que sea igual a typen. La igualdad de tipos se comprueba accediendo a Type Name, como el mismo string es usado para representar este nombre, en el caso de que dicho string coincida con el nombre de typen , entonces se puede decir que los dos tipos son iguales. Existen otros nodos que hacen uso de esta representación de objetos. TypeOfNode, por ejemplo, devuelve el primer campo del objeto correspondiente al nombre del tipo.

4 Problemas técnicos

4.1 Comentarios y strings en el lexer

En el proceso de tokenización de una cadena dos de los elementos más complicados resultan ser los comentarios y los strings. Para su procesamiento se definieron 2 nuevos estados en Lexer: comments:exclusive y strings:exclusive. Por definición la libreria PLY contiene el estado initial, donde se agrupan todas las reglas y tokens definidas para el procesamiento normal. En la definición de un nuevo estado, se definen nuevas reglas para el procesamiento especializado, es decir ,que ciertos tokens dentro de estos estados pueden ser analizados de manera diferente a que si estuvieran en el estado initial. De esta manera cuando se entra en en uno de los estados, todas las reglas que no correspondan al mismo son ignoradas.

Comentarios:

Tipo 1 (–): como es toda la línea no necesario entrar en un estado para procesarlo, con una expresión regular pudimos darle solución.

Tipo 2 ((*...*)): con el primer (* entramos en el estado de commets, así asumimos que a partir de ese momento todo es un comentario hasta que

aparezca el *) que lo cierre. No necesariamente regresamos al estado inicial donde las reglas qué no son de comentarios no son ignoradas, sino que se tiene un control estricto de que comentario está cerrando dicho símbolo. Para esto se mantienen unos paréntesis balanceados con los inicios y finales de los comentarios y solo se regresa al estado inicial si el balance es igual a 0, lo que implica q ese *) cierra verdaderamente el inicio del comentario.

Strings:

En cuanto se analiza el carácter " desde el estado inicial del parser, se pasa al estado strings, donde definimos un conjunto de reglas especializadas para tokenizar de manera correcta el string. Entre dichas reglas se pueden destacar las siguientes:

T_strings_newline : encargada de analizar '/n' , carácter muy importante en los strings que representa el cambio de línea.

T_strings_end : Esta regla procesa el " dentro del estado string . Por defición si aparece dicho carácter en este estado significaría el fin del string en cuestión, a no ser esté ante- cedido por el carácter '
', en cuyo caso, se considera como parte del string.

4.2 Tipos por valor y por referencia

Una de las problemáticas en el desarrollo del compilador fue la representación de los objetos built-in, en especial: Int , Bool y String que en realidad no son representados como objetos; los dos primeros son tratados por valor mientras el último aunque es por referencia no tiene campos y es guardado en el la sección .data de MIPS. Sin embargo, todos son descendientes de Object que es la raíz de todas las clases en COOL y por lo tanto, poseen los métodos que están definidos en ese tipo. Con el objetivo de optimizar la memoria ocupada en el heap estos objetos no son guardados allí y no poseen la estructura de los objetos explicada anteriormente. Todas estas clases tienen en común que no es posible heredar de ellas, por lo tanto, una vari able de tipo Int , Bool y String no es posible que sea otra cosa. Esto permite que no haya problemas a la hora de hacer dispatch: siempre es posible determinar la función a la que se va a llamar, por lo que las funciones de cada uno de estos tipos siempre se llaman estáticamente, determinando el nombre de la función. Los métodos

que heredan de Object son redefinidos, ya que estos están implementados teniendo en cuenta la representación de objetos expuesta anteriormente que estos tipos no tienen.

A su vez, se requiere un trato especial en nodos como TypeOfNode porque estas variables no tienen en su primer campo el nombre de su tipo. Para llevar a cabo este análisis se guarda, para cada variable, el tipo de dirección a la que hace referencia. Se tienen como tipo de dirección: String, Bool, Int y por Referencia, este último para designar a los objetos representados mediante el modelo de objetos.

Por otro lado, la única manera de tratar algún objeto tipo String, Bool o Int es haciendo un casteo de este objeto de manera indirecta: retornando uno de estos valores cuando el tipo de retorno del método es Object o pasándolos como parámetro cuando el tipo de argumento es Object o de manera general asignándolos a una variable tipo Object . En estos casos se espera que el valor del objeto sea por referencia, así que se hace Boxing, llevándolos a una representación de objetos, almacenándolos en el heap. Para esto se crea un nodo especial en CIL BoxingNode a partir del cual se llevará a cabo la conversión en MIPS. Notar que el proceso con trario Unboxing, en el que se convierte un objeto por referencia a uno por valor, no es necesario implementarlo en COOL: el casteo directo no existe en este lenguaje y asignar un objeto de tipo Object a uno Bool, Int o String consiste un error de tipos.

4.3 Valor Void en MIPS

Otro de los problemas que se tuvo en cuenta fue la representación del valor nulo en MIPS. Para modelar este valor se creó un nuevo tipo, denominado Void , que lo único que contiene es como primer campo el nombre del tipo Void. Para comprobar que un valor es void, por lo tanto, simplemente se comprueba que el nombre del tipo dinámico de una variable sea igual a Void. Los errores de runtime ocasionados por intentar llamar estática o dinámicamente a una variable en void son controlados, así como la evaluación de una expresión en case, que de ser void ocasionaría un error al intentar buscar la información del tipo. Por lo que lo único necesario para la representación de void fue ese campo de nombre del tipo. Comprobar si una expresión es void consiste, por lo tanto, solamente en determinar si su tipo es Void.