Proyecto de Complementos de Compilación
COOL to MIPS

Est. **Sheila Mederos Miranda** S.MEDEROS@ESTUDIANTES.MATCOM.UH.CU

Est. **Alejandro Oliva González** A.OLIVA@ESTUDIANTES.MATCOM.UH.CU

Est. **Emmanuel de la Rosa Pita** E.ROSA@ESTUDIANTES.MATCOM.UH.CU

C-411

Índice

1 Gramática, Lexer y Parser

Se utilizó *C#* como lenguaje de programación y *Antlr* como herramienta para el reconocimiento del lenguaje. La gramática utilizada está diseñada de forma ambigua, aunque esta situación es remediada por la forma en que trabaja *Antlr*, pues para la desambiguación se escoge la primera producción que aparece. De esta forma, *Antlr* proporciona un lexer y un parser totalmente funcionales a partir de la gramática otorgada, devolviendo un árbol de derivación del programa.

2 AST

A partir del árbol de derivación otorgado por el parser *Antlr*, se construye el árbol de sintaxis abstracta del código *COOL* para la eliminación de nodos redundantes. Este árbol facilita mucho el trabajo posterior tal como el chequeo semántico y la transpilación a lenguaje intermedio.

3 Chequeo Semántico

Para la realización del chequeo semántico se utilizó la clase siguiente:

```
public class IType
{

    public IType Father { get; set; }

    public string Name { get; }

    public List<Attribute> Attributes { get; set; }

    public List<Method> Methods { get; set; }

    public IType(string name, IType father)...

    public IType(string name, IType father,
        List<Attribute> attr, List<Method> mtd)...
    ...
}
```

En donde, para cada tipo definido en COOL, *Name* es su nombre, *Father* es el tipo de cual hereda y *Attributes* y *Methods* son sus listas de atributos y métodos respectivamente.

Luego, para realizar el chequeo semántico sobre un nodo raíz del AST de COOL, se analizaron los siguientes aspectos principales:

1. Las palabras reservadas no pueden ser identificadores.
2. Las clases están definidas solo una vez.
3. No se puede heredar de tipos String, Int, Bool.

4. Todos los tipos referenciados están definidos en algún lugar del programa.

   Estos primeros aspectos semánticos se verifican a la vez en una misma clase *IVisitor*. La primera condición se comprueba mediante un recorrido bottom-up a través de los nodos del AST, aunque la propia gramática de COOL y el parser *Antlr*, evita que a nivel de árbol de derivación lleguen palabras reservadas como identificadores.

   El segundo y tercer criterio semántico, son verificados solo hasta nivel de nodos $Class_Def$. Esto es debido a que tanto las definiciones de tipos, como las definiciones de herencias, no se tratan en nodos de niveles inferiores.

   El cuarto aspecto es verificado al igual que el primero mediante un recorrido bottom-up a través de los nodos del AST, de manera que si se llega a una referencia a una clase, se verifica si existe una definición para esta en el programa.

5. El grafo de la herencia es un árbol.

   Para la comprobación de este aspecto solo se realiza un recorrido sobre los nodos del AST que representan definiciones de clases, bajando por cada rama del grafo de la jerarquía de clases, de manera que si se encuentra una clase *A* con padre *B* y la clase *B* ya se había analizado en esa rama, entonces se incumple el criterio analizado.

6. Todos los identificadores referenciados en un contexto (scope) determinado están definidos.

7. Todos los métodos referenciados en un contexto (scope) determinado están definidos.

8. No existen atributos de una clase padre redefinidos en una clase hijo.

9. Los métodos de una clase padre redefinidos en una clase hijo tienen la misma signatura.

   Para la verificación de estos aspectos, se utilizó la siguiente clase:

```
public class ContextType
{

    public IType ActualType { get; set; }

    public Dictionary<string, IType> Types { get; set; }

    public List<Method> Methods { get; set; }

    public Stack<Tuple<string, IType>> Symbols { get; set; }

    public ContextType(Dictionary<string, IType> types)...
    ...
}
```

En donde *ActualType* representa el tipo actual, *Types* serían todos los tipos definidos en el programa con sus respectivos nombres y *Methods* son los métodos que han sido definidos hasta un instante, esta propiedad se utiliza para verificar que en una misma clase no existen dos definiciones del mismo método. La pila *Symbols*, representa cada símbolo definido en el *scope*, de manera que si existen dos variables definidas con el mismo nombre (Ej: un atributo y una variable en un *let*) entonces se toma como la que tiene mayor precedencia a la que se encuentra más cercana al tope de la pila (la del *let*). La verificación de los últimos dos aspectos, se realiza utilizando la propiedad *Father* de la clase *IType*.

10. Chequeo de tipos.

    Este chequeo se realiza para verificar si para un nodo particular del AST, el uso de los tipos es consistente. Este proceso se realizó mediante un recorrido en post-orden sobre el AST ya que la consistencia del uso de tipos en un nodo particular depende de los tipos en sus nodos componentes. Luego, se va computando los tipos asociados a los nodos "hijos", y en el retorno se va chequeando en cada "padre" la consistencia y se computa el tipo del padre. Para dar la mayor cantidad de errores posibles, si un nodo no es consistente, este tendrá tipo *null* y continuará con el chequeo.

4 COOL - CIL

Se decidió realizar una transpilación del código de *COOL* a lenguaje intermedio en primer lugar, con el objetivo de facilitar la generación del código *MIPS*, agregando un nuevo nivel de abstracción. Para esto, se implementó un árbol de sintaxis abstracta de *CIL* y se utilizó una clase *IVisitor*, donde el método *Visit* devolvía un *string* que representa la dirección en memoria la dirección en memoria donde se guarda la información del nodo correspondiente.

5 Generación de código

Para realizar la transpilación de *CIL* a código *MIPS*, se utilizó nuevamente una clase que implementa *IVisitor*, donde el método *Visit* devuelve *void*. Toda la información referente al código *MIPS* se va a encontrar en las propiedades de tipo *string* "*Data*" y "*Text*" de esta clase.

5.1 Manejo de memoria en métodos

Se realizó el manejo de memoria de los métodos mediante la simulación del comportamiento de la pila y a partir del control sobre los punteros, se va guardando la memoria necesaria para cada llamada del método. Por ejemplo, al realizarse un llamado a una función, se guardan los valores de *FP* y *RA*, los de los argumentos y variables locales y luego se reserva una memoria necesaria para la ejecución del mismo. Después de estas operaciones, se mueven los punteros *FP* y *SP* de manera tal, que estos contengan cada elemento analizado anteriormente. Luego, antes de retornar la función, se cambian los valores de *FP* y *RA* a sus valores anteriores al igual que *SP* y se termina la función.

5.2 Manejo de memoria en objetos

Los objetos ocupan un espacio en memoria en dependencia de los atributos que contengan los mismo. En la primera posición se encuentra su tamaño, en la segunda posición una referencia a la variable que almacena su nombre, en la tercera posición una referencia a la tabla virtual de este, la cual contiene una referencia a la tabla virtual del padre del objeto y una referencia a todos sus métodos. Por último tenemos los atributos del objeto uno a continuación del otro.

5.3 Manejo de excepciones

Las principales excepciones manejadas, son:

- División por cero
- Índice fuera de rango
- Referencia vacía (null)

Cada una de estas precondiciones, se comprueban en el momento que sea necesario, y si se cumple una de ellas se lanza una excepción y se para la ejecución del programa.