Proyecto final de Compilación

Eziel Christians Ramos Piñón c-511

---

1. Análisis Lexicográfico

Realice la implementación de dos lexer, en el archivo lexer_rules2.py está el lexer implementado sin el uso de ply, este contiene la función get_tokens() que devuelve una lista de tokens, y determina los errores lexicográficos. En el archivo lexer_rules.py está la implementación del lexer con el uso de ply.

2. Parsing

En el parser hice uso de la herramienta ply, para implementar un parser SLR con la gramática que se encuentra en el manual de COOL. La gramática es la siguiente:
program $\longrightarrow$ classSet
classSet $\longrightarrow$ class ; classSet | class ;
class $\longrightarrow$ CLASS TYPE _inherits { ffeature }
_inherits $\longrightarrow$ INHERITS TYPE | empty
ffeature $\longrightarrow$ feature ; ffeature | empty
feature $\longrightarrow$ ID (formal) : TYPE { expr } | ID () : TYPE { expr } | temp
temp $\longrightarrow$ idDots | idDots $\leftarrow$ expr
idDots $\longrightarrow$ ID : TYPE
formal $\longrightarrow$ idDots , formal | idDots
expression_list $\longrightarrow$ expression_list expr ; | expr ;
expr $\longrightarrow$ NOT expr
expr $\longrightarrow$ expr + expr | expr - expr | expr * expr | expr / expr
expr $\longrightarrow$ (expr) | ISVOID expr | block | conditional | loop | case | dispatch
block $\longrightarrow$ expression_list
expr $\longrightarrow\sim$expr
expr $\longrightarrow$ ID
expr $\longrightarrow$ INTEGER
expr $\longrightarrow$ STRING
expr $\longrightarrow$ TRUE | FALSE
expr $\longrightarrow$ NEW TYPE
expr $\longrightarrow$ let
dispatch $\longrightarrow$ expr.ID (arguments_list_opt) | expr@TYPE.ID (arguments_list_opt)
|ID(arguments_list_opt)
arguments_list $\longrightarrow$ arguments_list , expr | expr
arguments_list_opt $\longrightarrow$ arguments_list | empty
empty $\longrightarrow \epsilon$
let $\longrightarrow$ LET declaration_list IN expr
declaration_list $\longrightarrow$ temp , declaration_list | temp
conditional $\longrightarrow$ IF expr THEN expr ELSE expr FI
loop $\longrightarrow$ WHILE expr LOOP expr POOL
case $\longrightarrow$ CASE expr OF add ESAC

add $\longrightarrow$ derivate ; add | derivate ;
derivate $\longrightarrow$ idDots $\Rightarrow$ expr
expr $\longrightarrow$ expr < expr | expr <= expr | expr = expr
expr $\longrightarrow$ ID $\leftarrow$ expr

En el archivo AST.py se encuentran definidos los nodos del Arbol de Sintaxis Abstracta. Una ves ejecutado el parser obtenemos el AST sobre el cual realizamos el análisis semántico en caso de que no exista ningún error lexicografico o sintáctico.

3. Análisis Semántico

Inicialmente determine para cada clase cuales son sus atributos, el tipo de los atributos de la clase y los atributos heredados asi como cuales son los métodos, con sus argumentos, el tipo de los argumentos y el tipo de retorno. Realice un dfs para determinar que no hay ciclos en la relacion de herencia entre las clases. Revise que las clases estan definidas una sola vez y que los metodos redefinidos tienen el mismo tipo en los respectivos argumentos y tipo de retorno que en el método del ancestro. Revise que no existan atributos redefinidos

En el analisis semántico utilicé el patrón visitor para visitar cada uno de los nodos del AST obtenido por el parser. En cada uno de los nodos se le otorga el tipo estático al nodo. Por ejemplo cuando se visita el nodo Integer, que representa los enteros se le otorga el tipo estático Int y asi respectivamente con cada nodo según las especificaciones del manual.

Realice un chequeo de tipos en cada nodo por ejemplo que el nodo Plus que representa la operación de suma entre dos expresiones cumpla que la expresion izquierda y derecha tengan tipo estatico Int asi como en el nodo Conditional, la condicional tenga tipo estatico Bool. Determine entre otras especificaciones que en el dispatch los parametros sean del tipo correcto.

4. Generación de Código

Una vez revisado que el AST cumple con el chequeo semántico, a partir del AST de COOL obtengo el AST de CIL. En el archivo AST_CIL se encuentran definidos los nodos del AST de CIL, que este tiene una seccion .Data que contiene un diccionario donde estan guardados los string, una seccion .Type donde se guardan los diferentes tipos que existen y en la seccion .Code se guardan todas las funciones.

4.1. Generando CIL

En el archivo cool_to_cil.py, haciendo uso del patrón visitor, transformo cada nodo del AST de COOL en instrucciones del AST de CIL, obtengo una lista de funciones en CIL donde cada una esta formada por un conjunto de parametros, variables locales e instrucciones. En el archivo basic_classes.py estan definidas en CIL las funciones de las clases básicas del lenguaje COOL.

4.2. Generando MIPS

En mips implemente algunas funciones necesarias como es calcular la longitud de un string, comparar dos string, copiar un string, obtener un substring o concatenar dos string entre otras. Haciendo uso una vez mas del patrón visitor por cada instruccion en CIL la transformo en las respectivas instrucciones en MIPS. Haciendo uso del Heap para guardar los objetos, en la pila mantengo las varianles locales y la dirección de retorno. Cada Objeto guardado en el Heap tiene un primer campo Conform; que apunta a un array en la seccion .DATA de mips, donde se van a guardar los ancestros ordenados del Objeto desde el hasta Object, el segundo campo es un puntero que apunta a los métodos asociados a la clase del objeto, de ahi en adelante los demás campos son los correspondientes a los atributos del objeto.