Reporte

Integrantes

Yasmin Cisneros Cimadevila ---- C511

Jessy Gigato Izquierdo ---- C511

Arquitectura del Compilador

La estructura del compilador la modelamos de la siguiente manera:

- Tokenizador
- Parser (en este se genera el AST)
- Proceso de Semántica:
    - Colector de tipos
    - Constructor de tipos
    - Analisis de dependencia cíclica
    - Chequeo de tipos
- Generación de Código:
    - Transformación del AST
    - AST COOL-CIL
    - AST CIL-MIPS
    - Generación del archivo .mips

Tokenización y Parser

Para la construcción del lexer y el parser se utiliza la biblioteca `ply` de python, que provee herramientas para la generación del AST a partir de una gramática. El método de parseo utilizado fue `LALR`.

Semántica

Del proceso de análisis de la semántica se obtiene un nuevo AST en el que cada una de sus ramas ha adquirido el significado semántico que debe tener, es decir, se asegura de que se cumplan las reglas definidas que asignan un significado lógico a las expresiones dadas por la sintaxis del lenguaje.
La evaluación de las reglas semánticas define los valores de los atributos de los nodos del AST para la cadena de entrada. Para el caso de los operadores, cada nodo de este tipo, al terminar el análisis sabe si la operación asociada es aplicable o no.
En el colector y constructor de tipos se recopilan los mismos, tanto los existentes en el lenguaje como los creados por las clases definidas en el código de entrada, y por último en el chequeo de tipos se comprueba si en el contexto dado puede ser utilizado.

Generación de Código

En este punto ya tenemos un AST sintáctica y semánticamente correcto, y que quiere generar un programan en MIPS equivalente al código de entrada que COOL.

Transformación del AST:

Primeramente se hacen modificaciones en el AST actual agregando una nueva funcionalidad en la que los atributos declarados pero no asignados que no pertenecen a un tipo build-in (Int, Str, Bool) se les asigna una expresión nula, también en esta transformación se le asigna un constructor a las clases con el cual posteriormente se realizará la inicialización de la instancia de clase requerida.

Como pasar de COOL a MIPS directamente resulta complicado, se pasa primero del AST de COOL a un AST de CIL y luego a partir de este último se construye un AST de MIPS a partir del cual se obtiene un archivo `.mips` listo para ser ejecutado.

AST COOL-CIL:

La necesidad principal de tener un lenguaje intermedio es facilitar la trasformación del código al lenguaje final (MIPS) ya que este no presenta definiciones para tipos y atributos sino solamente muestra una serie de instrucciones que deben de ser ejecutadas en búsqueda de un resultado final por tanto CIL nos sirve de puente entre nuestro lenguaje de entrada (COOL) y el lenguaje de salida (MIPS). Este está diseñado para hacer menos engorrosa la búsqueda de errores intermedios y la agrupación de código que va a ser reutilizado.
Un ejemplo donde se evidencia la utilidad de este puente es en caso del `CaseNode` ya que el lenguaje MIPS no presenta nada parecido a esta cláusula por tanto este debe de pasar por un algoritmo que realice una búsqueda de tipos y semejanzas entre ellos para obtener el resultado final, en caso de que no existiera CIL sería complicadísimo lograr una compatibilidad.

AST CIL-MIPS:

Luego de lograr una reducción intermedia del lenguaje hacia CIL se convierten cada uno de los nuevos nodos de este AST en un conjunto de instrucciones que serán llevadas directamente a lenguaje MIPS

Puntos a destacar a nivel de implementación:

Manejo de la memoria:

El manejo de la memoria se realiza mediante el convenio de C, este hace creer a cada uno de los métodos que es dueño del stack

La sección de .data se encuentran contenidos los diferentes tipos con la siguiente estructura:

- tipo del objeto
- padre
- nombre de la dirección
- contructor
- métodos ...

El contructor es de gran ayuda ya que ayuda a aligerar código llamando a su etiqueta correspondiente cuando necesite crear una instancia de la clase en cuestión.

Con respecto a los métodos que son sobreescritos mediante herencia estos referencian a la dirección de memoria del método (la etiqueta) al cual le hacen override.

Un ejemplo visual sería el siguiente:

```
...
type_A: .word 56
type_A_inherits_from: .word type_Object
type_A_name_address: .word type_A_name_size
type_A___init__: .word function___init___at_A
type_A_abort: .word function_abort_at_Object
type_A_type_name: .word function_type_name_at_Object
type_A_copy: .word function_copy_at_Object
type_A_value: .word function_value_at_A
type_A_set_var: .word function_set_var_at_A
type_A_method1: .word function_method1_at_A
type_A_method2: .word function_method2_at_A
type_A_method3: .word function_method3_at_A
type_A_method4: .word function_method4_at_A
type_A_method5: .word function_method5_at_

type_B: .word 56
type_B_inherits_from: .word type_A
type_B_name_address: .word type_B_name_size
type_B___init__: .word function___init___at_B
type_B_abort: .word function_abort_at_Object  # heredado del tipo Object
type_B_type_name: .word function_type_name_at_Object  # heredado del tipo Object
type_B_copy: .word function_copy_at_Object # heredado del tipo Object
type_B_value: .word function_value_at_A # heredado del tipo A
type_B_set_var: .word function_set_var_at_A # heredado del tipo A
type_B_method1: .word function_method1_at_A # heredado del tipo A
type_B_method2: .word function_method2_at_A # heredado del tipo A
type_B_method3: .word function_method3_at_A # heredado del tipo A
type_B_method4: .word function_method4_at_A # heredado del tipo A
type_B_method5: .word function_method5_at_B
...
```

↑ Fragmento de código generado en el test `arith.cl` ↑

[Link del proyecto]