Proyecto de compilación

Integrantes

- Enmanuel Verdesia Suárez C-411
- Samuel David Suárez Rodríguez C-412

Instalación / Ejecución

Para compilar un fichero de COOL se puede usar el comando:

```
python3 coolc.py "path/to/file.cl"
```

Este compila el fichero `file.cl` y almacena el código generado en un fichero del mismo nombre pero pero con extension `.mips`, ubicado en la carpeta raíz donde se ejecutó el comando.

Se pueden consultar las demás opciones de la línea de comandos para el compilador ejecutando `python3 coolc.py -h`.

```
usage: coolc.py [-h] [--out OUT] [--run | --no-run] [--verbose | --no-verbose] file

positional arguments:
  file                  COOL source file.

optional arguments:
  -h, --help            show this help message and exit
  --out OUT             Name for .mips generated file after compilation.
  --run, --no-run       Execute the file compiled with SPIM. (default: False)
  --verbose, --no-verbose
                        Verbose output. (default: False)
```

- `--out`: modifica el fichero de salida para el `.mips` generado, que por defecto es creado con el mismo nombre del fichero de COOL y en la dirección raíz donde se está ejecutando el compilador.
- `--run`: facilita la ejecución del programa escrito en COOL ejecutando automáticamente el fichero `.mips` de salida. Para realizar dicha acción es necesario tener `spim` instalado y en el path.
- `--verbose`: imprime en consola el AST generado después de concluido el análisis lexico, parsing y análisis semántico, asi como el código intermedio (CIL) generado previamente a la generación de código final.

Requerimientos

El proyecto fue desarrollado y probado bajo un ambiente en `Python 3.9.5`, asi que se espera compatibilidad con esta version y superiores (`3.9+`). No se garantiza la correctitud del compilador o que este sea ejecutable en versiones inferiores.

La única dependencia del compilador es `ply==3.11`, la cual puede ser instalada ejecutando el comando `python3 -m pip install ply==3.11`.

Arquitectura

Para la implementación del compilador de `COOL` se dividió el proceso de desarrollo en las etapas siguientes:

- Análisis Sintáctico
  - Lexing
  - Parsing
- Análisis Semántico
  - Recolección de tipos (Type collection)
  - Construcción de tipos (Type building)
  - Chequeo de tipos (Type checking)
- Generación de código (Code generation)
  - COOL -> CIL
  - CIL -> MIPS

Lexing

Para el análisis léxico se utilizó el módulo `ply` de `Python`, el cual permite generar esta parte del proceso de manera automática simplemente definiendo el conjunto de tokens del lenguaje.

Se emplearon dos estados exclusivos para el automata además del incial, uno para tokenizar los string y otros para los comentarios que ocupan multiples líneas.

Al concluir esta fase se obtuvo cada uno de los tokens relevantes involcrados en el código fuente, estos almacenan su lexema y tipo de token.

Parsing

De igual manera se utilizó `ply` para la fase de parsing debido a que soporta varios tipos de parsers como el parser `LALR(1)` que resuelve de manera eficiente la gramática de `COOL`, esta gramática se definió en base al manual oficial de `COOL` cool-manual, definida en su página 16.

En esta fase se estableció la precedencia de los operadores de acuerdo al manual en la sección 11.1. Se establecieron las reglas de la gramática de forma apropiada para obtener el AST una vez finalizado el proceso de parsing de los tokens y asegurar que cada nodo almacena la línea y columna correspondiente al token, esto permite mejorar la información de los errores en el chequeo semántico.

Recolección de tipos

Esta fase se encarga de recorrer el AST generado previamente y definir los tipos del lenguaje. Entre estos tipos tenemos los BUILT_IN (Bool, String, Int, IO, Object), así como los tipos definidos en el código por el usuario en las declaraciones de clases. En un recorrido posterior se realiza el chequeo y asignación de padres de tipos, debido a que en el lenguaje las declaraciones de clases se pueden encontrar en cualquier orden, por lo que es necesario recolectar los tipos primeramente. En esta fase se resuelven también las excepciones de herencia cíclica.

La lógica para esta fase se implementó en el archivo `collector.py`

Construcción de tipos

En esta fase se recorre el AST con el objetivo de visitar cada `feature`(método o atributo) de las clases para asignarla a cada tipo y chequear la existencia de una clase `Main` con el método `main`.

La lógica para esta fase se implementó en el archivo `builder.py`

Chequeo de tipos

Esta fase es la encargada de validar el uso correcto de los tipos definidos en el programa y detectar otros errores definidos dentro de la semántica de COOL, estos errores con su correspondiente descripcion pueden ser encontrados bajo el fichero `errors.py`.

Por ejemplo entre ellos se encuentran: - Verificar que las asignaciones de las variables sea el tipo adecuado a su definición, así como los argumentos de funciones. - `self` es de solo lectura. - no violar el número de argumentos y los tipos de estos cuando se sobreescribe una función. - No usar variables que no estén definidas previamente en el contexto. - Evitar herencia de los tipos Int, String y Bool. - Verificar el correcto uso de los operadores +, -, /, *, <, <=, =, etc. - Evitar ramas duplicadas en el `case of`.

En este recorrido sobre el AST además se crea el scope del programa para cada una de las clases, funciones, el `let in` y el `case of`. Los scopes establecen una jerarquía de herencia, de tal forma que el scope de una clase es hijo del scope de la clase que esta hereda, el scope de una función es hijo del scope de la clase en que esta se encuentra definida y los scopes de los `let in` y `case of` son hijos del scope del contexto en que se encuentren.

Los scopes permiten ocultar las definiciones de variables de los contextos superiores. La salida de este recorrido sobre el AST es el scope raíz resultante de la visita a cada uno de los nodos.

La lógica para esta fase se implementó en el archivo `checker.py`

Generación de código intermedio COOL -> CIL

Para compilar el código en `COOL` a un lenguaje de bajo nivel como `MIPS`, se utilizó un lenguaje intermedio para disminuir la dificultad en la generación de código entre estas dos partes. Para ello definimos un pseudolenguaje (`CIL`) que posee elementos similares al estudiado en clase más algunos agregados y permiten controlar de una manera más sencilla el flujo del programa al generar el código en `MIPS`. Entre estos añadidos tenemos:

- Abort

  `ABORT`: Termina el programa con un mensaje indicando el tipo desde el cual se llamó esta instrucción.

- Errores en tiempo de ejecución:

  `CASE_MATCH_RUNTIME_ERROR`: Devuelve un error en tiempo de ejecución con el mensaje `"RuntimeError: Case statement without a match branch"`. Se usa en las expresiones de tipo case of cuando ninguna rama conforma el resultado de la expresión.

  `EXPR_VOID_RUNTIME_ERROR`: Devuelve un error en tiempo de ejecución con el mensaje: `"RuntimeError: Expression is void"`. Usado para controlar excepciones con expresiones de tipo `void`

- Conforms

  `<var> = CONFORMS <expr> <Type>`: Usada para saber si el resultado de `<expr>` conforma el tipo `<Type>`. Se creó por la necesidad de saber en tiempo de ejecución si el tipo que retorna la expresión en un `case of` podía ser asignado a una rama de un tipo dado.

Dicho esto, se implementaron 2 visitors encargados de generar el código en `CIL`. El primero se encuentra en el archivo `types_data_visitor.py` y es el encargado de definir las secciones `.TYPES` y `.DATA`, en donde se alojarán los tipos definidos en `COOL` y los datos constantes (mensajes de excepción, strings definidos en el código, …) respectivamente. El segundo visitor encontrado en el archivo `code_visitor.py` se encarga de generar la sección `.CODE` en la cual se encuentra toda la lógica del programa. Ambos visitors fueron encargados de devolver un AST de `CIL` para el próximo paso de generación de código.

Manejo de `case of`

Este tipo de expresión tuvo un tratamiento especial debido a que la evaluación del tipo de la expresión dentro del `case` se debe realizar en tiempo de ejecución, y en base a esta seleccionar la rama correspondiente como valor de retorno. Para resolver esto hacemos uso de la instrucción definida anteriormente en `CIL`: `CONFORMS`, sin embargo, con esto solo sabemos si el tipo de este resultado conforma el tipo de la rama, pero según la definición del `case of` queremos el tipo más específico que lo cumpla. Esto lo resolvimos reordenando las ramas

del case en función a su profundidad en el árbol de tipos de mayor a menor, quedando como primeras ramas las más específicas. De esta manera, la primera rama que cumpla que `CONFORMS <expr> <branch.type>` sea verdadero, es la seleccionada, y será la más específica para ese tipo puesto que las de mayor profundidad ya fueron visitadas, por tanto se hace un salto para esa rama.

Generación de código intermedio: CIL -> MIPS

Una vez tenemos el AST de `CIL`, solo resta el paso final, generar el código final `MIPS` que será el que se ejecutará desde el emulador `spim`. Para esto definimos un visitor encargado de traducir cada instrucción.

Representación de Instancias en Memoria

Para representar los objetos de `CIL` en memoria desde `MIPS` usamos la siguiente notación:

```
Type: # Nombre del tipo
  .word 4 # Espacio necesario para una instancia de este tipo
  .word <Padre> # Label del tipo padre

  # Métodos de la instancia
  .word Type__init
  .word Type__abort
  .
  .
  .
  #
  .word Type_type_name # Label para direccionar el string del typename
  .asciiz "Type" # Typename (String del tipo)
```

Esta definición se encuentra en la sección del `.data` de MIPS y se puede decir que sirve de esqueleto para la las instancias.

Por lo visto anteriormente cada tipo contiene información relacionada con este, como la lista de atributos, métodos, el string correspondiente a su typename, así como una referencia al address del label de su padre. Todos los tipos son tratados como instancias de clases, incluyendo los `BUILT_IN` como `Int, String y Bool` en los que podemos ver su valor en el offset 4 de su dirección. Los métodos se representan con labels que son definidos posteriormente en la sección `.text` de `MIPS`. Existe un label especial llamado `main` que será el encargado de correr el programa, su función es instanciar el tipo `Main` y ejecutar sus instrucciones.

Cada una de las instancias posee como primer atributo un puntero a su tipo correspondiente (ubicado en `.data`). En los bytes continuos almacena cada uno de sus atributos.

Para resolver el método de una instancia primeramente se resuelve el tipo de la instancia, al cual apunta el primer byte de esta. Este puntero más el offset

del método permite encontrar el label de la función correspondiente en mips e invocarla.

Todos los procedimientos en mips esperan recibir las instancias por referencia, por tanto realizan internamente el unboxing de los atributos de estas. De igual forma al retornar un tipo Int, String, Bool, se retorna una nueva instancia con su tipo y valor correctamente asignados, no el valor que esta contiene.