Cool Compiler

Autores ✒️

- **Carmen Irene Cabrera Rodríguez** - cicr99
- **David Guaty Domínguez** - Gu4ty
- **Enrique Martínez González** - kikeXD

Requerimientos 📋

Para la ejecución de este proyecto necesita tener istalado:

- Python3.7 o superior
- Las dependencias que se encuentran listadas en el archivo requirements.txt
- Spim para ejecutar programas MIPS32

Si lo desea, usted puede instalar todas las dependencias necesarias ejecutando en su terminal el siguiente comando, desde el directorio `<project_dir>/src`:

```
make install
```

Modo de uso

Para compilar y ejecutar un archivo en COOL deberá ejecutar el siguiente comando en la terminal desde el directorio `<project_dir>/src`:

```
make main <file_path>.cl
```

Si usted no proporciona ningún archivo, se tomará por defecto el archivo `code.cl` presente en dicho directorio. El comando anterior es equivalente a:

```
./coolc.sh <file_path>.cl
spim -file <file_path>.mips
```

Arquitectura del compilador

Para la implementación de este proyecto se utilizaron como base los contenidos y proyectos desarrollados en 3er año; añadiendo las funcionalidades faltantes y realizando modificaciones y mejoras sobre el código ya existente.

Fases

Las fases en que se divide el proceso de compilación se muestran a continuación y serán explicadas con más detalle en las secciones siguientes:

1. Lexer
2. Parsing
3. Recolección de tipos
4. Construcción de tipos
5. Chequeo / Inferencia de tipos
6. Traducción de COOL a CIL
7. Traducción de CIL a MIPS

Lexer

Para el análisis léxico se utilizó el módulo `lex.py` del paquete PLY de Python, que permite separar el texto de entrada (código COOL) en una colección de *tokens* dado un conjunto de reglas de expresiones regulares.

Para la obtención de los tokens de *string* y los comentarios multilíneas se definieron en el lexer, además del *INITIAL*, que es el estado que usa el lexer por defecto, dos estados exclusivos:

```
    states = (
        ("string", "exclusive"),
        ("comment", "exclusive"),
    )
```

Esto permitió tener en cuenta: el uso de caracteres inválidos en el primer caso, y los comentarios anidados en el segundo.

Además se llevaron a cabo cálculos auxiliares para obtener el valor de la columna de cada token, puesto que el lexer solo cuenta con el número de fila y el index.

Parsing

Se utilizó una modificación de la implementación previa del parser LR1 para llevar a cabo la fase de *parsing*; esta se realizó para poder almacenar el token, en lugar de solo su lexema; puesto que el token también guarda la posición *(fila, columna)*.

La gramática utilizada es S-atributada. Podrá encontrar la implementación de la misma en grammar.py

Recolección de tipos

Esta fase se realiza mediante la clase *Type Collector* que sigue los siguientes pasos:

- Definición de los *built-in types*, o sea, los tipos que son inherentes al lenguaje Cool : *Int*, *String*, *Bool*, *IO*, *Object*; incluyendo la definición de sus métodos. Además se añaden como tipos *SELF_TYPE*, *AUTO_TYPE*.
- Recorrido por las declaraciones hechas en el programa recolectando los tipos creados.
- Chequeo de los padres que están asignados a cada tipo. Como las clases pueden definirse de modo desordenado, el chequeo de la asignación correcta de padres para cada clase debe hacerse después de recolectar los tipos. De esta forma es posible capturar errores como que un tipo intente heredar

de otro que no existe. Aquellas clases que no tengan un padre explícito se les asigna *Object* como padre.
- Chequeo de herencia cíclica. En caso de detectar algún ciclo en la jerarquía de tipos, se reporta el error, y a la clase por la cual hubo problema se le asigna Object como padre, para continuar el análisis.
- Una vez chequeados los puntos anteriores, se reorganiza la lista de nodos de declaración de clases que está guardada en el nodo Program. La reorganización se realiza tal que para cada tipo A, si este hereda del tipo B (siendo B otra de las clases definidas en el programa) la posición de B en la lista es menor que la de A. De esta manera, cuando se visite un nodo de declaración de clase, todas las clases de las cuales él es descendiente, ya fueron visitadas previamente.

Construcción de tipos

La construcción de tipos se desarrolla empleando la clase *Type Builder*. Esta se encarga de visitar los *features* de las declaraciones de clase, dígase: funciones y atributos; tal que cada tipo contenga los atributos y métodos que lo caracterizan.

Además se encarga de chequear la existencia del tipo **Main** con su método **main** correspondiente, como es requerido en COOL.

En esta clase también se hace uso de la clase *Inferencer Manager* que permitirá luego realizar la inferencia de tipo. Por tanto, a todo atributo, parámetro de método o tipo de retorno de método, que esté definido como AUTO*TYPE se le asigna un _id* que será manejado por el manager mencionado anteriormente. Este id será guardado en el nodo en cuestión para poder acceder a su información en el manager cuando sea necesario.

Chequeo e Inferencia de tipos

En primer lugar se utiliza la clase *Type Checker* para validar el correcto uso de los tipos definidos. Toma la instancia de clase *Inferencer Manager* utilizada en el *Type Builder* para continuar la asignación de *id* a otros elementos en el código que también pueden estar definidos como *AUTO_TYPE*, como es el caso de las variables definidas en la expresión *Let*. Las variables definidas en el *Scope* se encargarán de guardar el *id* asignado; en caso de que no se les haya asignado ninguno, el id será *None*.

La instancia de *Scope* creada en el *Type Checker*, así como la de *Inferencer Manager* se pasarán al *Type Inferencer* para realizar la inferencia de tipos.

Ahora bien, la clase Inferencer Manager guarda las listas *conforms_to*, *conformed_by*, *infered_type*. El *id* asignado a una variable representa la posición donde se encuentra la información relacionada a la misma en las listas.

Sea una variable con *id = i*, que está definida como *AUTO_TYPE* y sea *A* el tipo estático que se ha de inferir:

- `conforms_to[i]` guarda una lista con los tipos a los que debe conformarse *A*; note que esta lista contiene al menos al tipo *Object*. El hecho de que *A* se conforme a estos tipos, implica que todos ellos deben encontrarse en el camino de él a Object en el árbol de jerarquía de tipos. En caso contrario se puede decir que hubo algún error en la utilización del *AUTO_TYPE* para esta variable. Sea *B* el tipo más lejano a *Object* de los que aparecen en la lista.

- `conformed_by[i]` almacena una lista con los tipos que deben conformarse a *A*. Luego el menor ancestro común (*LCA - Lowest Common Ancestor*) de dichos tipos deberá conformarse a A. Note que este siempre existirá, pues en caso peor será *Object*, que es la raíz del árbol de tipos. Sea *C* el *LCA* de los tipos guardados. Note que si la lista está vacía, (que puede suceder) *C* será *None*.
- Como *C* se conforma a *A* y *A* se conforma a *B*, tiene que ocurrir que *C* se conforma a *B*. En caso contrario, se reporta un uso incorrecto de *AUTO_TYPE* para esa variable. Todos los tipos en el camino entre *B* y *C* son válidos para inferir *A*; pues cumplen con todas las restricciones que impone el programa. En nuestro caso se elige *C*, que es el tipo más restringido, para la inferencia. En caso de que *C* sea *None* se toma *B* como tipo de inferencia.
- `infered_type[i]` guardará el tipo inferido una vez realizado el procedimiento anterior; mientras tanto su valor es *None*.

La clase *Inferencer Manager* además, está equipada con métodos para actualizar las listas dado un *id*, y para realizar la inferencia dados los tipos almacenados.

El *Type Inferencer* por su parte, realizará un algoritmo de punto fijo para llevar a cabo la inferencia:

1. Realiza un recorrido del AST (Árbol de Sintaxis Abstracta) actualizando los conjuntos ya mencionados. Cuando se visita un nodo, específicamente un *ExpressionNode*, este recibe como parámetro un conjunto de tipos a los que debe conformarse la expresión; a su vez retorna el tipo estático computado y el conjunto de tipos que se conforman a él. Esto es lo que permite actualizar las listas que están almacenadas en el *manager*.
2. Infiere todos los tipos que pueda con la información recogida.
3. - Si pudo inferir al menos uno nuevo, regresa al punto 1; puesto que este tipo puede influir en la inferencia de otros.
   - Si no pudo inferir ninguno, significa que ya no hay más información que se pueda inferir, por tanto se realiza un último recorrido asignando tipo *Object* a todos los AUTO_TYPES que no pudieron ser inferidos.

> Se considera que un tipo puede ser inferido, si no ha sido inferido anteriormente, y si su lista *conforms_to* contiene a otro tipo distinto de Object o su lista *conformed_by* contiene al menos un tipo.

Por último se realiza un nuevo recorrido del *AST* con el *Type Checker* para detectar nuevamente los errores semánticos que puedan existir en el código, ahora con los *AUTO_TYPES* sustituidos por el tipo inferido.

Traducción de COOL a CIL

Se definió un *visitor* en el que se recorre todo el *ast* generado en etapas anteriores y que recibe el contexto, que también fue creado previamente, para tener la información relacionada a los tipos que se encuentren en el código. El objetivo fundamental de este recorrido es generar otro *ast* que posee estructuras pertenecientes a CIL y que hará más fácil la generación de código MIPS posteriormente. Además, se generan chequeos que permitirán lanzar errores en tiempo de ejecución.

Primero que todo, se generan todos los tipos pertenecientes a COOL por defecto. Para ello, por cada tipo se crea un nodo que contenga sus atributos y funciones, lo que permite luego generarlos en MIPS. Después de este paso, comienza en sí el recorrido al *ast* de COOL.

En este recorrido se generan las 3 principales estructuras que posee el código de CIL:

- los **tipos**, donde se guarda un resumen de los *features* de cada uno de los tipos declarados en el código,
- los **datos**, sección en la que se encuentran todas las "macros" que serán utilizadas durante la ejecución,
- el **código**, donde son colocadas todas las funciones generadas a partir del recorrido.

En este recorrido por el ast, se define la estructura necesaria para la detección de ciertos errores en tiempo de ejecución. Entre los errores que se chequean se encuentran: la comprobación de que no se realicen divisiones por 0, el llamado a una función dinámica de una variable de tipo *void*, los índices en *strings* fuera de rango, entre otros. Agregar esta comprobación en el ast de CIL hace mucho más sencillo el proceso de recorrido de este *ast* posteriormente.

En el caso del *case* se chequea que la expresión principal no sea de tipo *void* y además, que se conforme a alguna rama en la ejecución de este. El algoritmo empleado para reconocer por cuál de las ramas continuará la ejecución del código comienza por: tomar el tipo de todas las ramas del *case*, llámese a este conjunto A; por cada elemento del conjunto A se toma la cantidad de tipos dentro del propio conjunto que se conforman a $a_i, i \in [1, |A|]$,de modo que se obtienen los pares $<a_i, |{a_j \leq a_i, \forall j, j\in[1, |A|]}|>$. Se define $|{a_j \leq a_i, \forall j, j\in[1, |A|]}|$ como a_{i_c}. Tomando los elementos a_i por el que menor a_{i_c} tenga, se estará tomando los nodos más abajos en el árbol de tipos dentro de cada posible rama de este. Si se ordenan las ramas del *case* por el que menor a_{i_c} se obtendrá una lista. Luego se recorre esta generando por cada elemento el subconjunto B_i donde $b_{i_i} \in B_i$ si $b_{i_i} <= a_i$. Se chequea si el tipo de la expresión principal del *case* aparece en este subconjunto. En el caso de que aparezca, el case se resuelve yendo por la rama que posee el tipo a_i.

Traducción de CIL a MIPS

Para la generación de código MIPS se definió un *visitor* sobre el *ast* de CIL generado en la etapa anterior. Este *visitor* produce un nuevo *ast* que representan las secciones: *.DATA*, *.TEXT* y las instrucciones en el código MIPS. Otro *visitor* definido esta vez sobre los nodos del *ast* del código MIPS se encarga de producir el código de MIPS que será ejecutado por el emulador SPIM.

Representación de objetos en memoria

El principal desafío en esta etapa es decidir como representar las instancias de tipos en memoria. Los objetos en memoria se representan de la siguiente manera:

Dirección x	Dirección x + 4	Dirección x + 8	...	Dirección x + a * 4
Tipo	Atributo 0	Atributo 1	...	Atributo $a - 1$

Por lo que un objeto es una zona continua de memoria de tamaño $1 + 4 * a$, donde a es la cantidad de atributos que posee el objeto. El tipo y cada atributo son de tamaño 1 *palabra*.

El campo *Tipo* es un número entre 0 y $n-1$, siendo n la cantidad total de tipos definidos en el programa de COOL a compilar. Un atributo puede guardar un valor específico o dicho valor puede ser interpretado como la dirección en memoria de otro objeto.

Para saber la cantidad de tipos y asignarles a cada uno un valor entre 0 y n, en el *visitor* sobre el *ast* de CIL primero se recorren todos los tipos definidos por el código CIL, asignándoles valores distintos de

manera ordenada según se van descubriendo. Además, por cada tipo se guardan también los nombres de sus parámetros y métodos en el orden en que se definieron en el tipo.

Para obtener o modificar un atributo específico de una instancia conociendo el nombre del atributo, se busca su índice en los atributos almacenados para el tipo en cuestión. Si el índice es i, entonces su valor estará en la dirección de memoria $(x+4) + (i * 4)$.

Inicialización

Cuando se crea una nueva instancia mediante la instrucción de CIL *ALLOCATE* se conoce el tipo del objeto a crear. Esta información se aprovecha para inicializar con valores por defecto la instancia de acuerdo a su tipo. Los tipos primitivos de COOL se inicializan de forma específica. Para los demás tipos, el código CIL de la etapa anterior genera para cada tipo una función *init* que se encarga de esta tarea, la cual es llamada en el código CIL y traducida a MIPS después.

LLamado de función dinámico

Para cada tipo, se guardan sus métodos en una lista llamada *dispatch*. Una lista *dispatch* de m métodos tiene la siguiente estructura

Dirección x	**Dirección x + 4**	**Dirección x + 8**	**...**	**Dirección x + (m-1) * 4**
Método 0	Método 1	Método 2	...	Método m-1

Se tendrán n listas, una por cada tipo. Cada celda es de una palabra y contiene la dirección a la primera instrucción del método correspondiente, o lo que es lo mismo, la dirección de la etiqueta generada para el método.

Los métodos en la lista se encuentran en el mismo orden en que fueron definidos en el tipo.

Estando una lista *dispatch* específica, se decide la ubicación del método buscado por un proceso análogo a los atributos en las instancias de los objetos explicado anteriormente. Si el índice del método dentro del tipo es i, entonces la dirección del método buscado estará en la dirección $x + 4 * i$.

Ahora solo faltaría saber por cuál de las listas *dispatch* decidirse para buscar el método dado un tipo.

Para eso se tiene otra lista llamada *virtual*. Su función es almacenar por cada tipo, la dirección a su lista *dispatch* . La lista *virtual* tiene la siguiente forma:

Dirección x	**Dirección $x + 4$**	**Dirección $x + 8$**	**...**	**Dirección $x + (n-1) * 4$**
dispatch 0	*dispatch* 1	*dispatch* 2	...	*dispatch* $n - 1$

Recordar que n es la cantidad de tipos.

Dado una instancia en memoria, se puede ver su tipo en la primera de sus direcciones continuas. Luego se hace otro proceso análogo a como se buscaron los atributos y métodos. Se obtiene el índice del tipo de la instancia y se decide por cual *dispatch* buscar el método que se quiere invocar. Si el índice del tipo es i, se buscará en la lista *dispatch* en la posición $x + 4*i$.

Estructura del proyecto

El *pipeline* que sigue el proceso de compilación se observa en el archivo main.py. Se hace uso de las funcionalidades implementadas en el paquete `compiler`, que presenta la siguiente estructura:

```
├── cmp
│   ├── ast.py
│   ├── automata.py
│   ├── cil_ast.py
│   ├── grammar.py
│   ├── __init__.py
│   ├── mips_ast.py
│   ├── pycompiler.py
│   ├── semantic.py
│   └── utils.py
├── __init__.py
├── lexer
│   ├── __init__.py
│   └── lex.py
├── parser
│   ├── __init__.py
│   ├── parser.py
│   └── utils.py
└── visitors
    ├── cil2mips
    │   ├── cil2mips.py
    │   ├── __init__.py
    │   ├── mips_lib.asm
    │   ├── mips_printer.py
    │   └── utils.py
    ├── cool2cil
    │   ├── cil_formatter.py
    │   ├── cool2cil.py
    │   └── __init__.py
    ├── __init__.py
    ├── semantics_check
    │   ├── formatter.py
    │   ├── __init__.py
    │   ├── type_builder.py
    │   ├── type_checker.py
    │   ├── type_collector.py
    │   └── type_inferencer.py
    ├── utils.py
    └── visitor.py
```

En su mayoría, los módulos que posee el paquete `cmp` fueron tomados de los proyectos y contenidos vistos en 3er año. Los paquetes `lexer` y `parser` definen la lógica para la tokenización y posterior parsing del texto de entrada respectivamente. El paquete `visitors` contiene las funcionalidades para llevar a cabo los recorridos sobre los *ast*, que en este caso serían: los *visitors* para realizar el chequeo semántico, el *visitor* que permite traducir de COOL a CIL, y finalmente, el *visitor* que permite traducir de CIL a MIPS.

Licencia

Este proyecto se encuentra bajo la Licencia (MIT License) - ver el archivo LICENSE.md para más detalles.