Informe Compilador COOL

Yandy Sánchez Orosa

tingui96

C-411

Juan Carlos Vázquez García

Juano97

C-412

Resolución

Primeramente, hemos de asegurarnos de tener una versión de Python superior a Python 3.7 e instalar **spim,** un contenedor que permite la ejecución de programas MIPS32. En nuestro caso utilizamos la versión 3.8.10 de Python. Se utilizará también, además de las librerías que se encuentran en el archivo ***requirements.txt***, la librería **PLY**, que contiene los módulos *lex* y *yacc*, herramientas que facilitan en gran medida el desarrollo del compilador.

Para ejecutar nuestro compilador se abre una terminal en **.src** y se ejecuta la siguiente instrucción:

```
$ ./cool.sh <arg>
```

Donde `arg` un archivo con extensión *.cl* con el código COOL a compilar y luego utiliza el mismo nombre y dirección de éste para crear un archivo donde se guardará el código MIPS generado.

Para desarrollar el compilador vamos a dividirlo en cuatro *procesos* que se ejecutan de forma secuencial (**Análisis Lexicográfico**, **Parsing**, **Análisis Semántico** y **Generación de Código**). Pasaremos a discutir cómo se desarrolló cada uno de estos procesos.

Análisis Lexicográfico

Se encarga de convertir el código COOL introducido a una secuencia de tokens.

Para esto definimos Token (`t`) cómo las palabras claves, literales, operadores e identificadores del lenguaje COOL, especificados en la documentación proporcionada, se crea una lista *tokens* con cada uno de estos tokens y luego a cada uno se le asigna la expresión regular que lo define en un método que posee nombre en la forma `t_<TOKEN>`.

```
tokens = [
    #Identifiers
    'TYPE', 'ID',
    #Primitive data types
    'INTEGER', 'STRING', 'BOOL',
    # Special keywords
    'ACTION',
    # Operators
      'ASSIGN',  'LESS',  'LESSEQUAL',  'EQUAL',  'PLUS',  'MINUS',  'MULT',  'DIV',
'INT_COMPLEMENT',
    #Literals
    'OPAR', 'CPAR', 'OBRACE', 'CBRACE', 'COLON', 'COMMA', 'DOT', 'SEMICOLON', 'AT',
] + list(reserved.values())


def t_INTEGER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Por último el módulo *lex* de **PLY** se encargará de crear el **lexer** al ejecutar `lexer = lex.lex()`.

Parsing

Para el proceso de parsing hemos de crear una gramática libre de contexto, que luego será utilizada por el módulo *yacc* para generar un parser **LALR**. Cada regla de la gramática posee una función de la forma p_<regla> donde el *docstring* de la función contiene la forma de la producción, escrita en **EBNF** (Extended Backus Naur Form).

```
def p_def_class(p):
    '''def_class : CLASS TYPE OBRACE feature_list CBRACE
                 | CLASS TYPE INHERITS TYPE OBRACE feature_list CBRACE'''
    if len(p) == 8:
        p[0] = ClassDeclarationNode(p, 2, 2, 6, 4)
    else:
        p[0] = ClassDeclarationNode(p, 2, 2, 4)
```

Cada una de estas producciones define un nodo del **AST** (árbol de sintaxis abstracta). Estos nodos reciben para su inicialización la producción, y luego los índices de la producción donde se encuentra la información correspondiente a cada atributo del nodo y se asignan los valores *lineno* y *colno*, necesarios para el manejo de errores. El **AST** se encuentra definido en el archivo **ast.py** dentro de la carpeta **parser**.

```
class ClassDeclarationNode(Node):
    def __init__(self, p, pos, type, features, parent_type=None):
        super().__init__()
        self.type = p[type]
        self.feature_nodes = p[features]
        self.parent_type = p[parent_type] if parent_type else parent_type
        self.lineno = p.lineno(pos)
        self.colno = find_column1(p, pos)
```

Análisis Semántico

Utilizando el método `semantic_check` comenzamos el análisis del **AST** devuelto por el **parser**, que empieza por una recolección de tipos que nos permite determinar los siguientes errores:

- Herencia cíclica
- Redefinición de clases
- Nombres de clases inválidos
- Inexistencia de la clase `Main`

```
def semantic_check(node):
    if type(node) is ProgramNode:
        program_visitor(node)
```

Luego se implementa un patrón *visitor* que se encarga de por cada nodo inferir el tipo estático de este, aplicar el patrón *visitor* a su descendencia obteniendo así un tipo estático inferido para cada una y detectar errores al aplicar las reglas de cada tipo. Todo esto se realiza apoyándonos en el archivo **types.py** donde se encuentran definidos los tipos *built_in* de COOL, así como la clase `Tipos` que abarca todo lo que corresponde a un tipo en COOL.

```
def program_visitor(program: ProgramNode):
    if not (CT.check_type(program) and CT.check_jerarquia(program)):
        return
    try:
        CT.TypesByName['Main']
    except KeyError:
        append_error_semantic(0, 0, f'Main class undeclared')
        return
      list_type_result  =  [CT.ObjectType,  CT.IntType,  CT.StringType,  CT.BoolType,
CT.IOType]
    classes = program.classes.copy()
    (...)
```

Durante el proceso de chequeo de tipos podemos encontrar errores como:

- Uso de tipos, métodos y variables no definidas
- Incompatibilidad de tipos
- Mal uso de `self` y `case`

Generación de Código (CIL)

Para la generación de código hacemos uso de un lenguaje intermedio donde para transformar un **AST** de lenguaje COOL a un **AST** de código CIL haremos un recorrido por los nodos del **AST** de COOL generando para cada nodo su conjunto de instrucciones equivalentes en el lenguaje intermedio.

```
class Node:
    pass
```

```
class ProgramNode(Node):
    def __init__(self, types, data, code, built_in_code):
        self.types = types
        self.data = data
```

```
self.code = code
self.built_in_code = built_in_code
```

Una instrucción que requirió especial interés fue `case` que para resolver la rama adecuada se ordenaron por herencia ascendente y determinamos el menor intervalo de herencia para cada tipo. Luego por cada rama de esta secuencia, se obtienen todos los tipos del programa que conforman a esta, y por cada uno que no haya sido tomado en cuenta en el procesamiento de ramas anteriores, se generan las instrucciones necesarias para comprobar si el tipo de la expresión principal del `case` coincide con él. En caso de coincidencia, se salta al bloque de las instrucciones generadas por el cuerpo de la rama; si no entonces se procede a comprobar con el tipo siguiente. Nótese que no se repiten comprobaciones.

Otra de las particularidades que se tuvo en cuenta para la generación de CIL fue la inicialización

de los atributos. Cuando se crea una instancia de una clase se deben inicializar todos sus atributos con su expresión inicial, si tiene. En otro caso se usa su expresión por defecto.

Con el objetivo de lograr esto se creó para cada tipo un constructor, cuyo cuerpo consiste en un bloque, dándole un valor a cada uno de sus atributos. Este es llamado cada vez que se instancia un tipo.

Los tipos *built-in* definidos en COOL son definidos en esta parte donde se hizo necesario crear nodos nuevos en CIL. `ExitNode` y `CopyNode` para la implementación de `abort` y `copy` de `Object`. Las instrucciones `Read` y `Print` fueron sustituidas por `ReadIntNode`, `ReadStrNode`, `PrintIntNode` y `PrintStrNode` con el objetivo de implementar de forma más sencilla las funciones `in_int`, `in_string`, `out_int` y `out_string` de `IO` respectivamente, y debido a que en MIPS se requiere un trato diferenciado al tipo `int` y `string` cuando se hacen llamados al sistema.

Para los métodos de `String` (`length`, `concat` y `substr`) se crearon los nodos `LengthNode`, `ConcatNode` y `SubstringNode`.

Para representar *instancias* en memoria necesita atributos especiales que no son definidos explícitamente en COOL, por lo que se llama a un `new A` para resolver esto en la generación de CIL. Se reserva espacio en memoria con la utilización del `allocate` y colocamos los atributos especiales de una instancia. Desde la fase de chequeo semántico se recolecta la información relativa a dichos atributos por lo que ya tendríamos esa información. Y luego se hace un llamado con a la función `_init_` del tipo construido en esta fase, inicializando primero la clase padre.

El acceso a la tabla virtual se resuelve con una nueva instrucción en CIL. De esta forma nos aseguramos que su traducción a MIPS vaya directamente al atributo donde se guardará la dirección a la tabla virtual, y no deba hacerse este proceso en la traducción de un llamado a función.

Generación de Código (MIPS)

Para generar el código de máquinas se usa una vez más el patrón *visitor*, recorriendo todos los

nodos de CIL que fueron creados. Entre las instrucciones se trata de manera especial la igualdad entre *strings*, que es realizada comparando caracter por caracter.

Para la creación de una instancia reservamos memoria utilizando `sbrk`. Cada una de las instancias tendrá como atributos especiales su tipo y su tamaño, pero este proceso se resuelve en la generación de código CIL. Para la localización de variables y el paso de argumentos utilizamos la pila liberando la memoria al terminar de ejecutar la función en el segundo caso.

Para cada uno de los tipos, definimos una función especial llamada `_init_`, que se crea en la fase de generación de código intermedio y se encarga de manejar la inicialización de los atributos heredados y los propios.

El espacio para guardar un *string* que introduce el usuario se reserva utilizando el `syscall` definido por `sbrk`. El espacio para un *string* resultante de una operación de cadena se reserva con el tamaño exacto necesario, que se conoce en ejecución a través de los argumentos del llamado.

```
register_pattern = re.compile(r'\$v[0-1]|\$a[0-3]|\$t[0-9]|\$s[0-7]')

def is_register(addr: str):
return register_pattern.match(addr) != None

BUFF = 1024
datos = []
CURRENT_FUNCTION = None
__VT__ = {}
__OFFSET_COUNT__= 0
__OFFSET__= {}
__EQUAL__=0
TYPE = {}
ADDR = {}
```