Compilador para lenguaje COOL

Isabella Sierra isabellamaria.sierra@estudiantes.matcom.uh.cu
Grupo C412

Adrian Tubal Páez Ruiz a.paez@estudiantes.matcom.uh.cu
Grupo C412

Eric Martín García e.martin@estudiantes.matcom.uh.cu
Grupo C411

Tutor(es):
Alejandro Piad, *Universidad de La Habana*

1. Lexer y Parser

Para resolver la sintáctica del compilador se utilizó el módulo de Python **PLY**. **PLY** es una implementación pura de Python de las populares herramientas de construcción de compiladores *lex* y *yacc*, generando el primer **AST** del proceso de compilación.

1.1 Tokenización con *lex*

1. Creación de los tokens: Una instancia de objeto **LexToken** (llamemos a este objeto **t**) tiene los siguientes atributos:

   - **t.type** que es el tipo de token (por ejemplo: 'INT' , 'PLUS' , etc.).
   - **t.value** que es el lexema
   - **t.lineno** que es el número de línea actual

2. Creación de las reglas: Las reglas de expresiones regulares pueden definirse como un **string** o como una función. En cualquier caso, el nombre de la variable tiene el prefijo **t_** para denotar que es una regla para hacer coincidir tokens.

   Para tokens simples, la expresión regular se especificó como **string**: $t\_PLUS = r'\backslash+'$

   Para tokens mas complejos se definieron funciones:

```
def t_STRING(t):
    r'"'
    t.lexer.string_start = t.lexer.lexpos
    t.lexer.begin('string')
```

3. Creacion del lexer: $lexer = lex.lex()$

2. Chequeo semántico

Para realizar el chequeo semántico se utilizó un `visitor` de una pasada que realiza las siguientes acciones sobre un nodo:

- Aplica el `visitor` correspondiente a las expresiones hijas involucradas y de éste se obtiene el tipo estático inferido de cada una.
- Se aplican las reglas de chequeo de tipos definidas para cada una de las operaciones para detectar errores.
- Se infiere el tipo estático de la expresión representada por el nodo y se retorna para ser utilizado por sus padres.

2.1 CoolType

Para auxiliarnos en el chequeo semántico definimos una clase `CoolType` que envuelve la significancia de un tipo en *COOL*.

Contiene los atributos:

- `name`: Representa el nombre del tipo.
- `inherits`: Determina si la clase es heredera de alguna clase definida en el programa.
- `parent`: Es una instancia `CoolType` que representa el padre de la clase definida.
- `methods`: Contiene todos los métodos definidos por la clase (De tipo `CoolTypeMethod`).
- `attributes`: Contiene los atributos definidos por la clase (De tipo `CoolTypeAttribute`).

y los métodos:

- `get_method`: Obtiene el método determinado por el argumento `id`, teniendo en cuenta si pudo haber sido definido por la clase guardada en `parent`.
- `get_all_methods`: Obtiene todos los métodos definidos por la clase.
- `add_method`: Agrega el método con nombre `id` en el conjunto de métodos de la clase. Esta función tiene implícitas las reglas de definición de métodos.
- Análogos métodos se declaran para los atributos.

3. Generación de código

3.1 Código Intermedio: CIL

Para facilitar el paso desde la sintaxis de *COOL* hacia MIPS, nos servimos de CIL como lenguaje intermedio.

Para generar un AST de CIL utilizamos el patrón `visitor` sobre los nodos del AST de *COOL*. La selección del visitor adecuado se realiza a partir de un diccionario de la forma `{type:visitor}`.

3.1.1 AST

Un programa CIL se representa con un `ProgramNode` y consta de cuatro partes:

1. Una sección *data*, representada por una lista de `DataNode`, que a su vez consiste en la declaración de una constante involucrada en el programa.
2. Una sección *types*, representada por una lista de `TypeNode`, que a su vez consiste en la declaración de un tipo con los identificadores de sus métodos y sus atributos.
3. Una sección *text*, representada por una lista de `DefFuncNode`, que a su vez consiste en la declaración de una función.

DataNode Un nodo de tipo *data* consiste en una constante de tipo *string* y una etiqueta que lo mapea:

```
data_1="Hello world\n"
```

TypeNode Un nodo de tipo *type* consiste en una etiqueta que representa el nombre del tipo y un conjunto de *features* que pueden ser atributos con su tipo o métodos. Un método será mapeado al método real a ejecutar de ser llamado éste. Se representa de la forma:

```
Type:
   attr1:    Type_1
   method_1: actual_method_1
   method_2: actual_method_2
```

DefFuncNode Un nodo de tipo deffunc constituye la declaración de un cuerpo de función, dentro de ella se ejecutará un bloque de código CIL, siguiendo las convenciones usuales.

3.1.2 CILBlock

Para representar un bloque abstracto de código CIL utilizamos una clase llamada `CILBlock` que contiene un `body`, un `locals` y un `value`.

Para convertir un AST de *COOL* a un AST de CIL visitaremos cada uno de los nodos del primer AST y generaremos un `CILBlock` correspondiente, donde cada uno de los elementos del `body` consiste en un nodo del AST de CIL.

3.1.3 Ejemplo: asignación

Sea el código en *COOL* `c<-a+b`, parseado a un árbol de la forma `AssignNode(VarNode(a),...expression...)` y con chequeo semántico perfecto.

AssignNode : El nodo más externo creará un `CILBlock` de la forma:

```
body = concat(expression.body, [
   CILAssignNode(a, expression.value)
])
value = expression.value

locals = concat(expression.locals, [
   a
]
```

3.2 Código MIPS

Para generar código MIPS utilizamos el mismo patrón `visitor` sobre los nodos del AST de CIL generado, para entonces generar un AST de código MIPS de forma similar a la generación de código intermedio.

3.2.1 Utilización de registros

Los registros se utilizaron de la siguiente forma:

- `$fp` para referenciar el inicio de la memoria correspondiente a la función actual.
- `$sp` para referenciar la posición donde se encuentra el tope de la pila.
- `$t0-t9` para realizar operaciones intermedias.
- `$ra` para guardar la dirección de retorno después de la ejecución de una función.
- `$v0` para guardar el valor de retorno de una función.
- `$a0-$a3` para guardar los argumentos necesarios en la ejecución de los `syscall`

3.2.2 Utilización de memoria

Paso de argumentos: En el paso de argumentos utilizamos la pila, liberando la memoria utilizada al terminar la ejecución de la función.

Creación de una instancia: Para la creación de una instancia reservamos memoria utilizando `sbrk`. Cada una de las instancias tendrá como atributos especiales su tipo y su tamaño, en la primera y última posición respectivamente; por tanto, al reservar memoria tenemos que hacerlo con un tamaño igual a la cantidad de atributos más dos.

Localización de variables: Para la localización de variables utilizamos la pila, y el acceso a cada una se controla a partir de un *offset* que las representa dentro de `$fp` para cada función. Este control se realiza a través de un diccionario en tiempo de compilación.

3.2.3 Llamado a función

Para realizar un llamado a función se realizan los siguientes pasos:

1. Se salvan los registros utilizados: `$fp, $ra`. (son los únicos que trascienden un llamado de función).
2. Se pasan los argumentos a la pila.
3. Se realiza la instrucción `Jal function_name`, que guarda la dirección de retorno en el registro `$ra`.
4. Se sacan los argumentos de la pila.
5. Se restauran los registros.
6. Se toma el valor de retorno de `$v0`.

La función llamada debe:

1. Mover `$fp` a donde se encuentra `$sp`.
2. Tomar sus argumentos de la pila.
3. Reservar memoria para todas sus variables locales.
4. Realizar las instrucciones que le correponden.
5. Liberar el espacio de la pila correspondiente a todas las variables locales.
6. Realizar un salto al contenido del registro `$ra`.