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 `ply` de `python` que implementa las populares herramientas de construcción de compiladores `lex` y `yacc`, que utilizamos para generar el primer **AST** del proceso de compilación.

1.1 Tokenización con `lex`

1.1.1 Creación de los *tokens*

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

- **t.type**: tipo de *token* (por ejemplo: `INT`, `PLUS`, etc.).
- **t.value**: lexema.
- **t.lineno**: número de línea actual

1.1.2 Creación de las reglas

Las reglas de expresiones regulares pueden definirse como una expresión regular propia de `python` o como una función si su comportamiento depende del contexto. 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: `t\_PLUS = r'\backslash+'`

Para *tokens* más complejos se definen las funciones. Veamos el ejemplo de cómo se hace el proceso en un *string*:

1. Detectando el inicio de un *string*:

```
def t\_STRING(t):
   r'"'
   t.lexer.string\_start = t.lexer.lexpos
   t.lexer.begin('string')
```

   En este caso, al detectar el inicio de un *string*, se guarda su posición inicial y se ingresa al estado `string` en el *lexer*.

2. Continuando el procesamiento del *string*:

```
def t\_string_body(t):
   r'([^\n\"\\]|\\.)+'
   if t.value.rfind('\0') != -1:
      add_lexer_error(...,"Null character")
```

   En este caso, se están tomando todos los caracteres válidos, incluyendo explícitamente caracteres especiales presentes en la expresión regular, y se comprueba que el caracter nulo no pertenezca a la misma.

3. Detectando cambio de línea:

```
def t\_string_newline(t):
   r'\\n'
   t.lexer.lineno += 1
```

   En este caso estamos detectando el cambio de línea en el código de *COOL*, la única acción a realizar es el aumento de la línea actual.

4. Detectando cambio de línea inválido:

```
def t\_string_error(t):
   if t.value[0] == '\n':
      add_lexer_error(..., "Unterminated
          string constant")
      t.lexer.lineno += 1
      t.lexer.skip(1)
      t.lexer.begin('INITIAL')
```

   En este caso detectamos un cambio de línea sin el uso del caracter `\`, generando un error, ignorando el cambio de línea y regresando al estado inicial del lexer.

5. Detectando fin de documento:

```
   def t\_string_eof(t):
      add_lexer_error(..., "Unterminated
          string constant")
```

En este caso detectamos el fin del documento antes del cierre de un *string*.

6. Detectando fin de cadena:

```
def t\_string_CLOSESTRING(t):
   r'"'
   t.value = [string_start : lexpos - 1]
   t.type = 'STRING'
   t.lexer.begin('INITIAL')
   return t
```

En este caso detectamos el fin de cadena, guardamos su valor en el *token* correspondiente y enviamos el *lexer* al estado inicial.

1.1.3 Creación del lexer

```
lexer = lex.lex()
```

1.2 Parsing con yacc

Una vez terminado el proceso de tokenización se realiza el proceso de *parsing* utilizando la herramienta `yacc`. Luego de la creación de una gramática libre del contexto `yacc` generará un parser **LALR**.

La gramática que se utilizó para el proceso de parsing es la sigiente:

```
program -> class_list

class_list -> def_class ; class_list
| def_class ;

def_class -> CLASS TYPE { feature_list }
| CLASS TYPE INHERITS TYPE { feature_list }

feature_list -> def_attr ; feature_list
| def_func ; feature_list
| empty

def_attr -> ID : TYPE <- expr
| ID : TYPE

def_func -> ID ( params ) : TYPE { expr }

params -> param_list

params -> empty

param_list -> param , param_list
| param empty

param -> ID : TYPE

expr -> LET let_attrs IN expr
| CASE expr OF case_list ESAC
| IF expr THEN expr ELSE expr FI
| WHILE expr LOOP expr POOL

expr -> ID <- expr

expr -> expr @ TYPE . ID ( arg_list )
| expr . ID ( arg_list )
| ID ( arg_list )

expr -> expr + expr
| expr - expr
| expr * expr
| expr \ expr
| expr < expr
| expr <= expr
| expr = expr

expr -> NOT expr
| ISVOID expr
| LNOT expr

expr -> ( expr )

expr -> atom

let_attrs -> def_attr , let_attrs
| def_attr

case_list -> case_elem ; case_list
| case_elem ;

case_elem -> ID : TYPE => expr

arg_list -> arg_list_ne
| empty

arg_list_ne -> expr , arg_list_ne
| expr

atom -> INT
atom -> ID
atom -> NEW TYPE
atom -> block
atom -> BOOL
atom -> STRING

block -> { block_list }

block_list -> expr ; block_list
| expr ;
```

Cada regla de la gramática será definida por una función donde el *string* de documentación de esa función contiene la especificación apropiada. Las declaraciones que conforman el cuerpo de la función implementan las acciones sintácticas de la regla junto con las definiciones de los nodos del **AST**.

```
def p\_program(p):
   '''program : class\_list'''
   p[0] = ProgramNode(p[1])
```

Al finalizar el proceso de *parsing*, estarán todas las condiciones creadas para iniciar el chequeo semántico.

2. Chequeo semántico

2.0.1 Recolección de tipos

El chequeo semántico comienza con un recorrido de recolección de tipos que sólo detecta la redefinición de clases. Esta fase se realiza utilizando el método `check_type_declaration`. Una vez finalizada la recolección se chequea la jerarquía de tipos.

2.0.2 Chequeo de jerarquía

En este proceso se valida la relación de herencia, verificando que no ocurran ninguno de los siguientes errores:

- Dependencia cíclica.
- Clases que heredan de tipos *built-in*.
- Herencia de tipos inexistentes.

*La herencia múltiple no es permitida en *COOL*, pero ésta se detecta en el proceso de *parsing*.

2.0.3 Continuación del chequeo semántico

Para culminar un chequeo semántico detallado, se utiliza un `visitor` (implementado en `src/semantic`) 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 en cada uno de los `visitors` de las expresiones 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` en `src/cool_types` que envuelve la significancia de un tipo en *COOL*. Su definición se complementa con las estructuras `CoolTypeMethod` y `CoolTypeAttribute`.

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`).
- `childs`: Contiene todas los `CoolType` que heredan directamente de la clase.
- `order_interval`: Atributo que define un intervalo (x, y) tal que si $A \leq B$, entonces (x_A, y_A) estará contenido en (x_B, y_B). Más adelante explicaremos para qué se utiliza este atributo.

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, así como los definidos por alguna clase superior en su línea de herencia.
- `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.
- `add_attr`: Agrega el atributo con nombre `id` en el conjunto de métodos de la clase. Esta función tiene implícitas las reglas de definición de atributos.
- `get_all_attributes`: Obtiene todos los atributos, heredados y propios.
- `get_self_attributes`: Obtiene todos los atributos propios.

2.1.1 Determinación del `order_interval`

El `order_interval` se expresa de la forma (x, y), donde x representa un índice que cumple que, si $A \leq B$, entonces $x_A \leq x_B$ y $y = x_{LAST}$ donde $LAST$ es el último de los hijos de A a la derecha en el árbol de herencia. Para crear este atributo utilizamos un sólo recorrido por el árbol de herencia, realizando acciones en preorden y en post-orden:

```
def set_types_order(current_type: CoolType,
    current_order: int) -> int:
  current_type.min_order = current_order
  for t in current_type.childs:
    current_order = set_types_order(t,
        current_order)
  current_type.order = current_order
  return current_order + 1
```

2.1.2 Tipos *built-in*

Los tipos *built-in* `ObjectType, IOType, StringType, IntType, BooltType` son declarados en el archivo `src/cool_types/types.py`, así como sus métodos de instancia.

2.1.3 Operación `pronounced_join`

Para realizar la operación `pronounced_join` simplemente buscamos el primer ancestro común auxiliándonos de las operaciones definidas en `CoolType`:

```
def pronounced_join(type_a, type_b):
  h = _type_hierarchy_(type_b)
  while type_a is not None:
```

```
        if type_a in h:
            break
        type_a = type_a.parent
    return type_a
```

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 Resolución de problemas en la fase de generación de CIL

Para facilitar el flujo de trabajo en la generación de código MIPS, por mucho más engorrosa; decidimos resolver ciertos problemas en la fase de generación de código intermedio:

- **Representación de una instancia en memoria:** Una instancia en memoria necesita atributos especiales que no son definidos explicítamente en *COOL*. Por tanto, las particularidades de un llamado a `new A` se resuelven en la generación de código intermedio en los siguientes pasos:
  1. Inicialmente se identifica el tipo `A` y en caso de ser `SELF_TYPE` se hace un llamado a la función `GetTypeAddr`(que explicaremos más adelante) de CIL con el argumento `self` y ese tipo se guarda como el tipo correcto a inicializar, decisión que se atrasa a ejecución utilizando la entrada a la función constructor de la *Tabla Virtual*.
  2. Se realiza la instrucción `Allocate` de CIL, que se traduce únicamente a reservar memoria en MIPS.
  3. Auxiliándonos de la función `SetAttr` de CIL, colocamos los atributos especiales de una instancia: `type_name`, `size`, `inheritance_interval_min`, `inheritance_interval_max`. Para hacerlo sin causar interferencia con la declaración de otros atributos, rompemos la convención de nombre de MIPS colocando una `@` al inicio. Desde la fase de chequeo semántico se recolecta la información relativa a dichos atributos en la clase `CoolType`, que se arrastra hasta la generación de código.
  4. Se hace un llamado con `VCall` a la función `_init_` del tipo, construida en ésta fase, con el siguiente cuerpo:
     (a) Generamos un llamado con `VCall` a la función `_init_` de la clase padre.
     (b) Colocamos y, de ser necesario, inicializamos todos los atributos **propios** de la clase.
- **Acceso a la tabla virtual:** El acceso a la tabla virtual se resuelve con una nueva instrucción en CIL llamada `GetTypeAddr`. De esta forma nos aseguramos que su traducción a MIPS vaya directamente al atributo donde sabremos que se guardará la dirección a la tabla virtual, y no deba hacerse este proceso en la traducción de un llamado a función. La traducción de un `VCall type function` sólo debe acceder al *offset* determinado en compilación para `function` con la dirección de *Tabla*

Virtual guardada en `type` en ejecución a través del llamado a `GetTypeAddr`.

- **Resolución de la instrucción `case`:** Para resolver la rama adecuada en la instrucción `case` realizamos los siguientes pasos:

  1. Ordenamos las ramas en orden de herencia ascendente, es decir $T_1 < T_2 < ... < T_n$.
  2. Determinamos directamente en compilación el `inheritance_interval_min(max)` para cada uno de los tipos que representan las ramas.
  3. Generamos las instrucciones `GetAttr expression @type_interval_min(max)` para obtener en ejecución el intervalo representativo de la expresión del `case`.
  4. Generamos las instrucciones asociadas a las comparaciones, así como la decisión de la rama a seleccionar a partir de éstas: la primera rama con tipo `A`, tal que $interval(expression) \subseteq interval(A)$ será seleccionada.

3.1.4 Optimización de `LOCALS`

Una vez generado el AST de CIL, pudimos detectar dos optimizaciones fundamentales en cuanto al uso de `LOCALS`:

- Si una variable resultante de una operación nunca se utiliza, la operación se remueve del cuerpo de instrucciones.
- Si el tiempo de vida de dos `LOCALS` no tiene intersección en un bloque de código, entonces se puede utilizar un sólo `LOCAL` para realizar la función de los dos.

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.

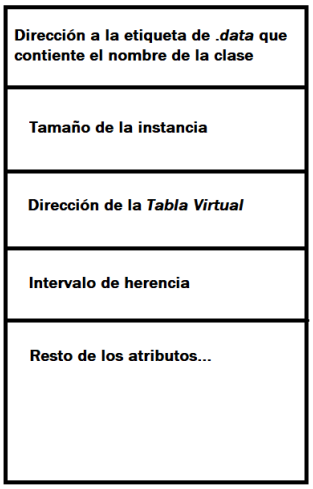
3.2.1 Utilización de registros

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

Representación de instancia: Una instancia se representa en memoria de la siguiente forma:



*El único atributo especial que se coloca en la fase de generación de código MIPS durante la traducción de la función `Allocate` es la dirección a la *Tabla Virtual*, el resto se resuelve en la fase de generación de código CIL según explicamos anteriormente.

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, pero este proceso es ajeno a la generación de código MIPS puesto que se resuelve en la generación de código CIL.

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

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

Tabla Virtual: Al inicio de la ejecución del programa se construye una *Tabla Virtual* de la siguiente forma:

1. Se reserva un espacio para cada tipo en la sección `.data`.
2. Se guarda, para la función X, la dirección de la etiqueta de la función Y adecuada, definida desde la generación de código intermedio. El *offset* respecto al inicio de la tabla virtual se prefija en compilación de manera tal que si A implementa X y B lo redefine, entonces X se encuentra en el mismo *offset* en ambas *Tablas Virtuales*.

Decisión de la función a llamar para `dynamic dispatch`: Para representar un llamado a función tenemos la instrucción `VCall type function` de CIL. Para

decidir qué función se debe llamar, basta localizar el *offset* predefinido en compilación de `function`, a partir de la dirección de *Tabla Virtual* guardada en `type`.

Pasos del llamado a función:

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` en el caso de saber la función precisa a llamar; de lo contrario, se utiliza la *Tabla Virtual* y el salto se realiza utilizando la instrucción `Jalr $ti`, donde `$ti` guarda la dirección de la función correcta. En cualquier caso 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`.

3.2.4 Constructor de instancia en MIPS

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.

3.2.5 Manejo de *strings* en memoria

- Un *string* constante se reserva en la sección `.data` en el inicio del programa, la estructura que contiene dichos *strings* está construida desde la fase de generación de código intermedio.
- El espacio para guardar un *string* que introduce el usuario se reserva utilizando el `syscall` definido por `sbrk` con un tamaño de *buffer* de `1024 bytes`.
- El espacio para un *string* resultante de una operación de cadena se reserva con el `syscall` definido por `sbrk` con el tamaño exacto necesario, que se conoce en ejecución a través de los argumentos del llamado.

3.2.6 Determinación del tipo en ejecución

Dentro de cada instancia, como representamos anteriormente, se guarda un atributo especial que consiste en la dirección en memoria de la tabla virtual del tipo de ésta. Para acceder a esta dirección construimos una instrucción apropiada desde la fase de código intermedio, que se traduce en MIPS al acceso directo a la dirección en memoria predefinida donde sabemos que se encuentra este atributo.

3.3 Ejemplo gráfico del *workflow* de generación de código: suma aritmética