Cool Compiler Project

Integrantes

- Thalía Blanco Figueras
- Nadia González Fernández
- José Alejandro Labourdette-Lartigue Soto

Introducción

El proyecto implementa un compilador capaz de interpretar el lenguaje COOL "The Classroom Object-Oriented Language". La solución está desarrollada en python.

Lexer y Parser (Utilizando PLY):

Para el desarrollo del lexer y el parser se utiliza la herramienta de parsing **PLY**. Esta es una implementación en python de lex/yacc.

Gramática

La gramática usada es libre de contexto y de recursión extrema izquierda. Debido a la forma en la que esta es definida, no presenta problemas de ambigüedad

```
program : class_list

class_list : class_def
           | class_def class_list

class_def : CLASS TYPE_ID LBRACE feature_list RBRACE SEMICOLON
          | CLASS TYPE_ID INHERITS TYPE_ID LBRACE feature_list RBRACE SEMICOLON

feature_list : attrs_def SEMICOLON feature_list
             | meth_def SEMICOLON feature_list
             | empty

attrs_def : attr_def
          | attr_def COMMA attrs_def

attr_def : OBJECT_ID COLON type
         | OBJECT_ID COLON type ASSIGN expr

meth_def : OBJECT_ID LPAREN param_list RPAREN COLON type LBRACE expr RBRACE

param_list : param other_param
           | empty

param : OBJECT_ID COLON type

other_param : COMMA param other_param
            | empty
```

```
expr : comparer LT open_expr_lvl1
     | comparer LTEQ open_expr_lvl1
     | comparer EQ open_expr_lvl1
     | open_expr_lvl1
     | comparer

open_expr_lvl1 : arith PLUS open_expr_lvl2
               | arith MINUS open_expr_lvl2
               | open_expr_lvl2

open_expr_lvl2 : term MULT open_expr_lvl3
               | term DIV open_expr_lvl3
               | open_expr_lvl3

open_expr_lvl3 : ISVOID open_expr_lvl3
               | INT_COMP open_expr_lvl3
               | open_expr

open_expr : LET let_var_list IN expr
          | OBJECT_ID ASSIGN expr
          | NOT expr

comparer : comparer LT arith
         | comparer LTEQ arith
         | comparer EQ arith
         | arith

arith : arith PLUS term
      | arith MINUS term
      | term

term : term MULT factor
     | term DIV factor
     | factor

factor : ISVOID factor
       | INT_COMP factor
       | atom

atom : INTEGER
     | OBJECT_ID
     | STRING
     | BOOL
     | LPAREN expr RPAREN
     | NEW TYPE_ID
     | IF expr THEN expr ELSE expr FI
     | WHILE expr LOOP expr POOL
     | LBRACE expr_list RBRACE
     | CASE expr OF branch_list ESAC
     | func_call

expr_list : expr SEMICOLON
          | expr SEMICOLON expr_list

let_var_list : OBJECT_ID COLON type
             | OBJECT_ID COLON type ASSIGN expr
             | OBJECT_ID COLON type COMMA let_var_list
```

```
             | OBJECT_ID COLON type ASSIGN expr COMMA let_var_list

branch_list : OBJECT_ID COLON type ACTION expr SEMICOLON
            | OBJECT_ID COLON type ACTION expr SEMICOLON branch_list

func_call : atom DOT OBJECT_ID LPAREN arg_list RPAREN
          | OBJECT_ID LPAREN arg_list RPAREN
          | atom AT TYPE_ID DOT OBJECT_ID LPAREN arg_list RPAREN

arg_list : expr other_arg
         | empty

other_arg : COMMA expr other_arg
          | empty

type : TYPE_ID
        | SELF_TYPE

empty :program : class_list

class_list : class_def
           | class_def class_list

class_def : CLASS TYPE_ID LBRACE feature_list RBRACE SEMICOLON
          | CLASS TYPE_ID INHERITS TYPE_ID LBRACE feature_list RBRACE SEMICOLON

feature_list : attrs_def SEMICOLON feature_list
             | meth_def SEMICOLON feature_list
             | empty

attrs_def : attr_def
          | attr_def COMMA attrs_def

attr_def : OBJECT_ID COLON type
         | OBJECT_ID COLON type ASSIGN expr

meth_def : OBJECT_ID LPAREN param_list RPAREN COLON type LBRACE expr RBRACE

param_list : param other_param
           | empty

param : OBJECT_ID COLON type

other_param : COMMA param other_param
            | empty

expr : comparer LT open_expr_lvl1
     | comparer LTEQ open_expr_lvl1
     | comparer EQ open_expr_lvl1
     | open_expr_lvl1
     | comparer

open_expr_lvl1 : arith PLUS open_expr_lvl2
               | arith MINUS open_expr_lvl2
               | open_expr_lvl2

open_expr_lvl2 : term MULT open_expr_lvl3
               | term DIV open_expr_lvl3
```

```
               | open_expr_lvl3

open_expr_lvl3 : ISVOID open_expr_lvl3
               | INT_COMP open_expr_lvl3
               | open_expr

open_expr : LET let_var_list IN expr
          | OBJECT_ID ASSIGN expr
          | NOT expr

comparer : comparer LT arith
         | comparer LTEQ arith
         | comparer EQ arith
         | arith

arith : arith PLUS term
      | arith MINUS term
      | term

term : term MULT factor
     | term DIV factor
     | factor

factor : ISVOID factor
       | INT_COMP factor
       | resoluted

resoluted : resoluted DOT OBJECT_ID LPAREN arg_list RPAREN
          | resoluted AT TYPE_ID DOT OBJECT_ID LPAREN arg_list RPAREN
          | atom

atom : INTEGER
     | OBJECT_ID
     | STRING
     | BOOL
     | LPAREN expr RPAREN
     | NEW TYPE_ID
     | IF expr THEN expr ELSE expr FI
     | WHILE expr LOOP expr POOL
     | LBRACE expr_list RBRACE
     | CASE expr OF branch_list ESAC
     | func_call

expr_list : expr SEMICOLON
          | expr SEMICOLON expr_list

let_var_list : let_var
             | let_var_assign
             | let_var COMMA let_var_list
             | let_var_assign COMMA let_var_list

let_var : OBJECT_ID COLON type

let_var_assign : OBJECT_ID COLON type ASSIGN expr

branch_list : branch
            | branch branch_list
```

```
branch : OBJECT_ID COLON type ACTION expr SEMICOLON

func_call : OBJECT_ID LPAREN arg_list RPAREN

arg_list : expr other_arg
         | empty

other_arg : COMMA expr other_arg
          | empty

type : TYPE_ID
     | SELF_TYPE

empty :
```

Las expresiones del lenguaje pueden separarse en dos grupos:

- Las que se conoce donde empiezan y terminan. Estas tienen un terminal específico para iniciarlas y otro para finalizarlas, y esos terminales siempre van a ser los mismos. Ejemplo las condiciones empiezan con `IF` y termina con `FI`, los ciclos con `WHILE` y `POOL`.
- Las que no cumplen la condición anterior. Estas empiezan o terminan con no terminales. En este grupo se presentan problemas de precedencia o conflictos y la gramática tiene que encontrar una forma de solucionarlos.

La definición del lenguaje manifiesta que las expresiones **let**, **assign** y **not** consumen todos los tokens que le siguen. Luego, con un nivel de precedencia mayor, están las expresiones de comparación y después las aritméticas. Para disminuir al mínimo, la precedencia de las expresiones let, assign y not, se utilizan los no terminales `open_expr_lvlx`. Una expresión intenta encontrar todas las comparaciones o expresiones aritméticas antes del terminal que marca el comienzo de la expresión let, assign o not. Luego,toda la cadena de tokens que sigue se sabe que pertenece al cuerpo de estas expresiones (que es también una expresión).

Para garantizar la precedencia correcta en las expresiones de comparación y aritméticas se emplea la distinción entre comparador, aritmético, término, factor y átomo. Donde un átomo se puede percibir como alguna de estas expresiones, donde su inicio y fin están bien definidos.

Lexer

En la implementación del lexer se declaran los tokens, que definen todos los posibles tokens que el compilador recibirá. Esta lista también se utiliza en el parser. Estos tokens se definen con la expresión regular compatible con el módulo **re** de Python

Ejemplo:

```
tokens = ['INTEGER', 'PLUS', 'MINUS', 'MULT', 'DIV']

t_PLUS = r'\+'
t_MINUS = r'-'
t_MULT = r'\*'
t_DIV = r'/'

def t_INTEGER(t):
    r'[0-9]+'
    t.value = int(t.value)
    return t
```

En el lexer se define el número de línea de cada token.

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
```

Parser

En la implementación del parser se utiliza `ast_hierarchy.py`. En este se definen los nodos del árbol que se construirá en el parser. Las reglas de gramática mostradas anteriormente se definen en python utilizando la biblioteca `ply`:

```
def p_class_list(p):
    """
    class_list : class_def
               | class_def class_list
    """
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = [p[1]] + p[2]
```

Los errores de esta fase son manejados por ply.yacc:

```
def p_error(p):
    if p is None:
        line_no = find_last_line(input_text)
        errors.append('(%s, 0) - SyntacticError: ERROR at or near EOF' %
line_no)
    else:
        col_no = find_column(input_text, p)
        errors.append(('(%s, %s) - SyntacticError: ERROR at or near
"%s"'.format(p) % (p.lineno, col_no, p.value)))
```

Los nodos del ast que se crean tienen en común los dos primeros argumentos: la fila y la columna en la que se encuentran. Por esta razón existen estructuras que podían ser representadas fácilmente como una tupla de datos, sin embargo, tiene más sentido hacerlas nodos propios capaces de almacenar la línea en la que se encuentran. Un ejemplo es el caso de la expresión **Case**: cada elemento de la `branch_list` es un nodo. Sucede muy a menudo que estos

son escritos en otras líneas del programa, y de ocurrir algún error en ellos se informaría que se encuentra en la línea en la que se empezó a definir la expresión del Case.

Lexer y Parser (Generado)

Para resolver el problema de Parsing del Lenguaje Cool, además de emplear **PLY**, se implementa un generador de parser y lexer, con un motor de expresiones regulares para este último.

Si se desea generar el lexer y parser con **PLY**, en el archivo `coolc.sh` se requiere que el archivo a emplear sea `main.py`:

```
...
python main.py $INPUT_FILE
```

En cambio, si se desea emplear el generador implementado, el archivo a emplear es `main_generated.py`.

```
...
python main_generated.py $INPUT_FILE
```

El parser generado es LR(1) cuya implementación se encuentra ubicada en la carpeta `parser_shr_lr1.py`, junto a la del parser SHIFT-REDUCE. Este archivo se encuentra en la carpeta `src`.

Para la construcción de autómata se emplea la clase Automaton ubicada en el archivo `automaton_class.py`, en la ruta `src/generated_utils`. En esta ruta se encuentra también el archivo `grammar_class.py` , con la implementación de las clases Símbolos, Producciones, No Terminales, Terminales, Item (LR(1)) .

Las clases Parser y Lexer, definidas en los archivos `parser_base.py` y `lexer_base.py` son empleadas como estructuras bases para el análisis sintáctico y léxico, ubicadas ambas en `src`; y requieren de importantes clases ubicadas en `src/generated_utils`, para poder llevar a cabo dichos chequeos: `token_class` incluye la definición de la clase Token así como los métodos necesarios para tokenizar, `serializer_class` garantiza la serialización y deserialización.

Para generar las gramáticas de los lenguajes, se requiere la entrada de un string que hace referencia al nombre con el que se desean crear los archivos de parser y lexer asociados a dicha gramática. Se sugiere que el nombre coincida con el de la gramática.

```
first_grammar = Grammar("first")
```

En el proyecto se implementa una gramática que define el lenguaje de expresiones regulares, que se requiere para el motor de regex, y poder machear los tokens del lenguaje Cool, durante el análisis
léxico. Dicha gramática se encuentra ubicada en el archivo `grammar_regex.py` y fue creada con el nombre **regex**:

```
regex_grammar = Grammar("regex")
```

En ese dicho archivo se generan los terminales de la gramática haciendo:

```
terminal=regex_grammar.terminal((token_name, lexeme, (extra)))
```

El campo extra es una tupla opcional, que no es empleada en la gramática de Regex pero sí en la de Cool.

Para generar los no terminales es similar:

```
Noterminal=regex_grammar.nonTerminals("Noterminal")
```

Por convenio, en la gramática generada, los terminales se inician con letra minúscula y los no terminales con mayúscula (en la generada con PLY es contrario). Las producciones tienen la forma:

```
Noterminal != FormaOracional + FormaOracional / atributos
```

donde las formas oracionales pueden ser o no épsilon.

Las gramáticas pueden contar con un método que especifique la forma en que son tratados los atributos, el cual ha de ser proporcionado en el momento en que se crea la gramática, para la gramática de Cool fue necesario:

```
def attr_decoder(attr, symbols_to_reduce, ast_class):
    def get_line_col(args:list):
        i=0
        l=len(args)
        while i<l:
            term=args[i]
            if hasattr(term,"line_no") and term.line_no !=-1:
                return term.line_no, term.col_no
            i+=1
        return -1,-1

    if attr:
        if len(attr)==2:
           ...

cool_grammar = Grammar("cool",attr_decoder)
```

Si no se especifica dicho método, la gramática emplea el que tiene implementado (en `grammar_class.py`):

```
 def attr_decoder(attr, symbols_to_reduce, ast_class):
        if attr:
            if len(attr)==2:
                attr_class, attr_pos = attr
                args = list(map(lambda i: symbols_to_reduce[i].tag, attr_pos))
                return getattr(ast_class, attr_class)(*args)
            ...
        return symbols_to_reduce[0].tag if len(symbols_to_reduce) else None
```

En el caso de la gramática de `regex` se emplea método para decodificar los atributos tiene empleado por defecto (ejemplo anterior). Las clases de su AST, se encuentran en el archivo `ast_regex.py`. Ya definidos todos los terminales, no terminales y producciones se generan el parser y lexer: `regex_parser.py` y `regex_lexer.py`.

Para generar dicha gramática se requiere estar ubicado en la ruta en que se encuentra `grammar_regex.py` (carpeta `src`) y correr el comando `python grammar_regex.py`. En el proyecto ya fueron ejecutados dichos comandos.

Para generar el motor de expresiones regulares se emplea el archivo `main_regex.py` en el que se genera el compilador de una determinada expresión regular, y retorna, a partir de una determinada entrada, si la expresión regular la reconoce. Nuestro motor de expresiones regulares reconoce solo caracteres ASCII y algunos UTF-8 (los comprendidos en los casos de prueba).

Luego este motor de expresiones regulares es empleado por la clase CoolMatch ubicada en el archivo `main_generated.py`, para machear los tokens del lexer de Cool. Los identificadores de dichos tokens, junto a otras clases necesarias para los atributos, se encuentran ubicados en `ast_cool_h_extender.py`:

```
class TOKEN_TYPE(Enum):
    LINEBREAK=auto()
    TAB=auto()
    NXTPAGE=auto()
    SPACE=auto()
    DIV=auto()
    MULT=auto()
    PLUS=auto()
    ...
```

En el archivo `gramar_cool.py` se encuentra la gramática para Cool y cuenta, con la función mostrada anteriormente, para el tratamiento de sus atributos. Esta gramática es análoga a la que se encuentran en los archivos `lexer_ply.py` y `parser_ply.py`, lo que estas se llevan a cabo con las especificaciones de PLY.

Para generar el parser y lexer de Cool se requiere estar ubicado en la ruta en que se encuentra `grammar_cool.py` (carpeta `src`) y correr el comando `python grammar_cool.py`. En el proyecto ya fueron ejecutados dichos comandos.

Análisis Semántico

Para el análisis semántico seguimos un patrón visitor. Se hacen 3 recorridos por el ast en esta fase:

- Type Collector
- Type Builder
- Type Checker

En el `TypeCollector` se hace un primer recorrido por el AST recolectando todos los tipos. También se añaden los tipos predefinidos por el lenguaje COOL.

En el `TypeBuilder` se analizan los cuerpos de cada clase. Por cada atributo o función visitado se crea ese feature en el tipo definido en el contexto. Adicionalmente se comprueba que los tipos utilizados para esos atributos, los tipos de retorno y de los parámetros de las funciones sean tipos declarados ya en el contexto.

En el `TypeChecker` se hace un análisis semántico a profundidad. Se visitan todos los nodos del ast y se comprueba que se correspondan los tipos esperados y que los llamados a funciones sean con el número y tipo de argumentos requeridos.
Además, se comprueba que se usen variables ya definidas en los **scopes**, y es que precisamente en este recorrido es en el que se crean los scopes de las variables del programa. Para resolver los problemas de herencia de atributos se asigna a la propiedad parent del scope que crea la clase que hereda, el scope creado por la clase heredada. Esta técnica es usada cada vez que interesa crear scopes específicos para ciertas secciones.

Un caso que prestó especial atención es el retorno de funciones que fueran **SELF_TYPE**. En tiempo de compilación no se puede asumir que el tipo de retorno de una función con **SELF_TYPE** puede ser sustituido por el tipo de la clase, y es que si se asume esto entonces se perdería todo el sentido del término y ejemplos como el siguiente no podrían ser reconocidos.

```
class A {
    f() : SELF_TYPE { self };
};

class B inherits A { };

class Main {
    x : B <- (new B).f();
    main() : B { x };
};
```

La herencia exige un comportamiento y es por ello que se mantiene este tipo de retorno **SELF_TYPE**. EL análisis de las expresiones dispatch fue el que puso en plenitud la definición de **SELF_TYPE** y es que se le intentará asignar como tipo de retorno a la expresión, el tipo del objeto al que se le manda a invocar la función. De esta forma es que se obtiene en el proyecto el chequeo de tipos para este tipo especial.

Generador de Código

El objetivo de esta etapa es bajar de **COOL**, que es un lenguaje de alto nivel que usa el concepto de herencia y polimorfismo, hasta código **MIPS**, que es de bajo nivel y donde está plano todo ese código. Como esta traducción del programa no es inmediata utilizamos un lenguaje intermedio que es **CIL** que nos facilita este cambio. Entonces se realizan dos traducciones: de **COOL** a **CIL**, y de **CIL** a **MIPS**

Cool a CIL

Para esta traducción se realiza un recorrido por el ast de COOL con un patrón visitor y toman las acciones pertinentes dependiendo del tipo de nodo que se analiza. Este recorrido devuelve una estructura que contiene, en nodos de CIL, el programa. Atendiendo a los principios de CIL existen 3 secciones para un programa escrito en ese lenguaje: la sección `Types` donde se define la estructura que tienen los tipos en COOL (entiéndase por esto los atributos o funciones que se definen en él, o en clases ancestros) y estos atributos tienen un orden específico; la sección `Data` donde están definidas las constantes del programa, en este caso tendremos todos los strings que se usan en él; y la sección `Code` donde están definidas todas las funciones del programa, siendo la primera de ellas la que se usa como función de inicio.

El gran mérito de este cambio de lenguaje está en generar instrucciones (definidas como nodos en el ast de CIL) a partir de las expresiones de COOL. Estas instrucciones, si bien no están definidas de manera inmediata en mips, son más cercanas a su paradigma de programación. Cuando se realice el próximo cambio de lenguaje cada una de estas instrucciones podrían acabar siendo varias instrucciones MIPS, pero el hecho de cambiar el sentido del lenguaje hace que sea más cómoda la equivalencia.

Para hacer instancias de los tipos de Cool se crea una función especial en CIL, que no es más que una especie de constructor, que inicializa cada uno de los atributos de esa clase.

Durante este proceso de traducción se tienen variables de instancias de la clase encargada de implementar el patrón visitor (COOLToCILVisitor). Estas variables son **dottypes**, **dotdata** y **dotcode** que continen los nodos correspondientes a estas secciones de las que hablamos recién; **current_type** y **current_method** que denotan el tipo y el método que se analizan actualmente en el ast de Cool; **current_function** que es el nodo función que se está creando actualmente de CIL.

CIL a MIPS

Para esta traducción se recorrerá ahora el ast de CIL generado en el proceso anterior. El primer problema a solucionar es como representar los objetos en memoria. Como convenio tendremos 5 secciones del espacio reservado para el objeto: La primera de estas (el offset 0) representa la posición en la que fue visitado ese tipo en un recorrido DFS partiendo por el tipo Object y visitando todos los hijos de cada tipo (un recorrido por el árbol de jerarquía de clases). La segunda sección es la encargada de representar el nombre de la clase, y lo que contendrá es la dirección del segmento de datos donde está contenido. La tercera representa la cantidad de posiciones que ocupa ese objeto en memoria, todas las clases a lo sumo ocupan 4 posiciones: 3 para estas 3 secciones que hemos visto y 1 para la próxima sección. La cuarta sección es la dirección a donde está definido en memoria las funciones que son declaradas o heredadas por la clase. Se detallará esto y luego se continuará con la quinta sección.

Existe un lugar en memoria donde están definidas las funciones que se heredan o declaran en cada clase, y estas funciones están dispuestas en orden. La forma de ordenarlos es que la posición en la que se encuentra el nombre de esa función va a ser la misma para todas las definiciones de funciones de otras clases que hay en memoria si estas clases se refieren a la misma función o a una redefinición de esta.

La quinta sección contiene el valor de los atributos creados o heredados por la clase, es la que cambia en tamaño. Cada una de las anteriores ocupaba 1 posición, sin embargo, esta sección ocupará tantas posiciones como atributos se definan o hereden en la clase y la posición de los atributos en esta sección importa. La forma de ordenarlos es similar al orden de los nombres de las funciones en las que se hablaba en el párrafo anterior. Aquí se definirán primero los atributos heredados (en el orden que los tiene la clase padre) y luego los atributos definidos en ella.

El offset se incrementa en 4 con cada posición, por lo que si queremos referirnos al valor del primer atributo de una clase A tendremos que referirnos al offset 16 (5ta posición (5-1)*4=16). Como se puede apreciar cada posición contiene el tamaño de 1 **word**

Sección 1	Sección 2	Sección 3	Sección 4	Sección 5
Id en DFS	Nombre	Tamaño	Dirección a funciones	Atributos

Hasta ahora se ha visto la representación en memoria de los objetos, se verá ahora la estrategia para llamar funciones. Para almacenar los argumentos de la función se utiliza la pila, son pusheados en orden contrario al que se piden para que se puedan sacar en orden luego utilizando el stack pointer (registro sp) como referencia. Las funciones también definen variables temporales y para almacenarlas se utilizará también la pila, como la cantidad de locals se conoce en CIL entonces se conoce el espacio que ocupan y se conoce además el orden que tienen. Nuevamente se puede acceder a ellos usando el stack pointer que está contenido en el registro sp. El valor de retorno de una función estará siempre contenido en el registro $a1.