

Universidad de San Carlos
Facultad de Ingeniería
Organización de Lenguajes y Compiladores 2
Proyecto 1

Manual Técnico TytusDB

Diego Estuardo Gómez Fernández 201612141
Jeralmy Alejandra de León Samayoa 201612139
André Mendoza Torres 201612154
Carlos Manuel García Escalante 201612276

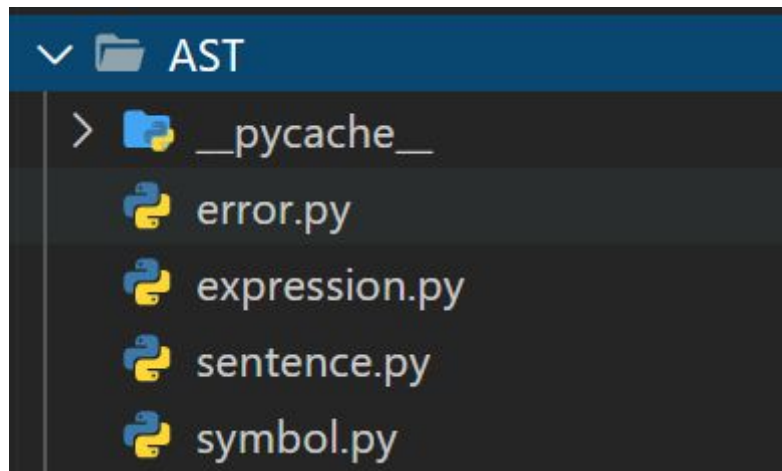
Interfaz Gráfica

Parser con PLY

Para la construcción del parser se utilizó el generador de analizadores PLY, que para su definición consta de dos grandes secciones:

- Analizador Léxico: se definen los tokens en diferentes lista de la forma nombre_lista(<lista de strings separados por coma>), se definió una lista para palabras reservadas a utilizar en el lenguaje SQL, luego en una distinta los nombres de los símbolos a utilizar y en una última los nombres de las expresiones regulares a utilizar. Luego se procede a definir el valor de los tokens que difieren de su nombre y necesitan una expresión regular como `t_<nombre> = r'ER'`. Los que necesiten algún tipo de manipulación respecto a su valor (numéricos, strings, etc) pueden ser definidos por una función con el mismo patrón de nombre anteriormente mencionado, se definen finalmente los caracteres ignorados y el manejo de los errores para los lexemas que no coincidan con nada de lo anterior.
- Analizador Sintáctico: utilizando como base el análisis realizado anteriormente, se define la precendencia de los operadores en una lista con nombre precedence de ser necesaria y se comienza con la definición de la gramática, utilizando funciones de la form `'def p_<nombre producción>(t):'` dentro de la función lo primero que aparece es un string que representa la producción con el siguiente formato `'noTerminal : noTerminal TERMINAL'`, para implementar todo el lenguaje a definir. Dentro del parámetro de t se encuentran los atributos sintetizados que el usuario vaya alojando en la construcción ascendente de la entrada.

Construcción del AST



Para la construcción del AST se utilizaron las siguientes clases con sus subclases:










- Error
- Expression
 - Value
 - Arithmetic
 - Range
 - Logical
 - Relational
 - Unary
 - MathFunction

- TrigonometricFunction
- ArgumentListFunction
- AggFunction
- ExpressionAsStringFunction
- NSeparator
- Alias
- Sentence
 - CreateDatabase
 - ShowDatabases
 - DropDatabase
 - DropTable
 - Use
 - AlterDatabaseRename
 - AlterDatabaseOwner
 - AlterDatabaseDropColumn
 - AlterTableAddConstraintUnique
 - AlterTableAddForeignKey
 - AlterTableAlterColumnSetNull
 - AlterTableAlterColumnType
 - AlterTableAddColumn
 - AlterTableDropConstraint
 - Insert
 - InsertAll
 - Delete
 - Truncate
 - Update
 - CreateType
 - CreateTable
 - Select
 - SelectMultiple
 - CreateTableOpt
 - ColumnId
 - ColumnCheck
 - ColumnConstraint
 - ColumnUnique
 - ColumnPrimaryKey
 - ColumnForeignKey

Cada una de estas clases almacena información necesaria para cada una de las sentencias o expresiones para realizar la ejecución de las mismas. Además cada una tiene su método `graphAST` que se explicara a detalle más adelante.

Ejecución de Instrucciones

Para la ejecución de instrucciones se ejecuta la lista de instrucciones recolectadas con el AST, con un método que verifica la instancia de que clase es la instrucción y la redirecciona al método que se encarga de realizar las acciones específicas de esa clase. Consta de los siguientes métodos.

-  execute_result.py
-  execute.py
-  executeCreate.py
-  executeDrop.py
-  executeExpression.py
-  executeInsert.py
-  executeSelect.py
-  executeSentence.py
-  executeShow.py
-  executeUse.py
-  executeValue.py
-  generateASTReport.py
-  insertTestValues.py

```
from .executeSentence import executeSentence
from .generateASTReport import graphAST
from .execute_result import *
andremendozatorres, 3 days ago | 2 authors (You and others)
class Execute():
    nodes = []
    errors = []
    messages = []
    querys = []
    types = {
        1: 'Entero',
        2: 'Decimal',
        3: 'Cadena',
        4: 'Variable',
        5: 'Regex'
    }
    def __init__(self, nodes):
        self.nodes = nodes
        self.errors = []
        self.messages = []
        self.querys = []
    # def __init__(self, nodes, errors):
    #     self.nodes = nodes
    #     self.errors = errors
    #Aqui va metodo principal ejecutar, al que se le enviara la raiz del AST
    #y se encargaran de llamar el resto de metodos
    def execute(self):
        if(self.nodes is not None):
            for node in self.nodes:
                executeSentence(self,node)
        dotAST = graphAST(self)
        result = execute_result(dotAST, self.errors, self.messages, self.querys)
        return result
```

TypeChecker

Al ser una base de datos cuando se hace una inserción a una tabla se debe verificar que cumpla con el tipo que se ha especificado en la creación de una tabla es por ello que en un archivo haciendo uso de diccionarios se almacenan los tipos que posee cada una de las tablas que corresponden a las bases de datos creadas. para ello se tiene un archivo llamado TypeChecker.py el cual contiene métodos que permite guardar los tipos y leer los tipos para comparar que se cumplan y así ya poder ejecutar los métodos que realizan el proceso de almacenamiento de la información. el TypeChecker también es utilizado para saber cual es la base de datos sobre la cual se está trabajando, y permite almacenar los modos sobre los cuales se crearon las bases de datos ya que son datos que sirven para saber qué método llamar del lado donde se manejan los árboles y la información.

Generación de Reportes

Para la generación de reportes se hace uso de la herramientas Graphviz, el lenguaje dot y el lenguaje Markdown, el reporte gramatical se genera en cada producción de la gramática mientras se va analizando la entrada y se construyen las definiciones dirigidas por la sintaxis que genera cada producción, mientras que el reporte de AST se genera mediante la función graphAST de cada clase del AST, construyendo un arbol n-ario que representa el análisis sintáctico de la entrada.

Ejemplo de construcción de una porción de reporte gramatical

```
# Grammar definition
def p_start(t):
    '''start : sentences'''
    global grammarreport
    grammarreport = "<start> ::= <sentences> { start.val = sentences.val }\n" + grammarreport
    grammarreport = reporthead + "``bnf\n" + grammarreport + "``\n" + "## Entrada\n" + "``sql\n" + input + "``"
    global noderoot
    noderoot = t[1]
```

Ejemplo de un método graphAST que recibe 2 parámetros, dot que es un string con el dot anterior y parent que es el padre de los nuevos nodos a generar.

```
class Arithmetic(Expression):
    def __init__(self, value1, value2, type):
        self.value1 = value1
        self.value2 = value2
        self.type = type
    def graphAST(self, dot, parent):
        dot += str(parent) + '->' + str(hash(self)) + '\n'
        dot += str(hash(self)) + '[label="' + str(self.type) + '"]\n'
        dot += self.value1.graphAST('', hash(self))
        dot += self.value2.graphAST('', hash(self))
        return dot
```

El método apunta el padre a su hijo que es un hash del nodo actual ya que este es único, luego se le asigna un label a ese hash del nodo actual y en el caso específico de esta operación se procede a generar el subárbol de los dos valores.