UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERIA

ESCUELA DE CIENCIAS Y SISTEMAS

LABORATORIO DE LENGUAJES Y COMPILADORES 2

MANUAL TECNICO

NOMBRES: ROBIN OMAR BUEZO DIAZ

BRANDON ANDY JEFFERSON TEJAXUN PICHIYA

KEWIN MASLOVY PATZAN TZUN

CARNET: 201944994

202112030

202103206

SECCION: A

ANALIZADOR LEXICO(SCANNER)

Reglas Léxicas: Creación de las reglas léxicas que el compilador utilizara durante la ejecución del programa.

```
reserveds = {
    # EJECUCION DDL
    'BEGIN'      : 'RW_begin',
    'END'        : 'RW_end',
    'SELECT'     : 'RW_select',
    'FROM'       : 'RW_from',
    'WHERE'      : 'RW_where',
    'DECLARE'    : 'RW_declare',
    'SET'        : 'RW_set',
    'CREATE'     : 'RW_create',
    'DATA'       : 'RW_data',
    'BASE'       : 'RW_base',
    'USE'        : 'RW_use',
    'TABLE'      : 'RW_table',
    'PRIMARY'    : 'RW_primary',
    'FOREING'    : 'RW_foreing',
    'KEY'        : 'RW_key',
    'REFERENCE'  : 'RW_ref',
    'ALTER'      : 'RW_alter',
    'ADD'        : 'RW_add',
    'DROP'       : 'RW_drop',
    'COLUMN'     : 'RW_column',
    'RENAME'     : 'RW_rename',
    'TO'         : 'RW_to',
    'INSERT'     : 'RW_insert',
    'INTO'       : 'RW_into',
    'VALUES'     : 'RW_values',
    'AS'         : 'RW_as',
    'UPDATE'     : 'RW_update',
    'TRUNCATE'   : 'RW_truncate',
    'DELETE'     : 'RW_delete',
    'THEN'       : 'RW_then',
    'WHEN'       : 'RW_when',
    'NOT'        : 'RW_not',
```

```
    # EJECUCION DML
    'IF'         : 'RW_if',
    'ELSE'       : 'RW_else',
    'CASE'       : 'RW_case',
    'WHILE'      : 'RW_while',
    'FOR'        : 'RW_for',
    'IN'         : 'RW_in',
    'LOOP'       : 'RW_loop',
    'BREAK'      : 'RW_break',
    'CONTINUE'   : 'RW_continue',
    'FUNCTION'   : 'RW_function',
    'RETURNS'    : 'RW_returns',
    'RETURN'     : 'RW_return',
    'PROCEDURE'  : 'RW_procedure',
    'PRINT'      : 'RW_print',
    'TRUNCATE'   : 'RW_truncate',
    'CONCATENAR' : 'RW_concatenar',
    'SUBSTRAER'  : 'RW_substraer',
    'HOY'        : 'RW_hoy',
    'CONTAR'     : 'RW_contar',
    'CAST'       : 'RW_cast',
    # TIPOS DE DATOS
    'INT'        : 'RW_int',
    'BIT'        : 'RW_bit',
    'DECIMAL'    : 'RW_decimal',
    'DATE'       : 'RW_date',
    'DATETIME'   : 'RW_datetime',
    'NCHAR'      : 'RW_nchar',
    'NVARCHAR'   : 'RW_nvarchar',
    'NULL'       : 'RW_null',
}
```

```
tokens = tuple(reserveds.values()) + (
    'TK_lpar',
    'TK_rpar',
    'TK_semicolon',
    'TK_comma',
    'TK_dot',
    'TK_plus',
    'TK_minus',
    'TK_mult',
    'TK_div',
    'TK_mod',
    'TK_equalequal',
    'TK_equal',
    'TK_notequal',
    'TK_lessequal',
    'TK_greatequal',
    'TK_less',
    'TK_great',
    'TK_and',
    'TK_or',
    'TK_not',
    'TK_id',
    'TK_field',
    'TK_date',
    'TK_datetime',
    'TK_nvarchar',
    'TK_decimal',
    'TK_int',
)
```

```
# SIGNOS DE AGRUPACIÓN Y FINALIZACIÓN
t_TK_lpar       = r'\('
t_TK_rpar       = r'\)'
t_TK_semicolon  = r'\;'
t_TK_comma      = r'\,'
t_TK_dot        = r'\.\.'
# OPERACIONES ARITMETICAS
t_TK_plus       = r'\+'
t_TK_minus      = r'\-'
t_TK_mult       = r'\*'
t_TK_div        = r'\/'
t_TK_mod        = r'\%'
# OPERADORES RELACIONALES
t_TK_equalequal = r'\=\='
t_TK_equal      = r'\='
t_TK_notequal   = r'\!\='
t_TK_lessequal  = r'\<\='
t_TK_greatequal = r'\>\='
t_TK_less       = r'\<'
t_TK_great      = r'\>'
t_TK_and        = r'\&\&'
t_TK_or         = r'\|\|'
t_TK_not        = r'\!'
```

Expresión Regulares: Definición de las expresiones regulares que el programa podrá utilizar en la lectura de las entradas que esta recibe.

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

t_ignore = ' \t'

def t_comments(t):
    r'\-\-([^\r\n]*)?'
    t.lexer.lineno += 1
    t.lexer.skip(1)

def t_commentm(t):
    r'[/][*][^*]*[*]+([^/*][^*]*[*]+)*[/]'
    t.lexer.lineno += len(t.value.split('\n'))
    t.lexer.skip(1)

def t_TK_id(t):
    r'\@(\_)*[a-zA-Z][a-zA-Z0-9\_]*'
    return t

def t_TK_field(t):
    r'(\_)*[a-zA-Z][a-zA-Z0-9\_]*'
    t.type = reserveds.get(t.value.upper(), 'TK_field')
    return t

def t_TK_date(t):
    r'\"\d\d\d\d\-\d\d\-\d\d\"'
    t.value = t.value[1 : len(t.value) - 1]
    return t
```

```
def t_TK_nvarchar(t):
    r'\"(([^\n\"\\]|\\.)*)\"'
    t.value = t.value[1 : len(t.value) - 1]
    return t

def t_TK_decimal(t):
    r'[0-9]+\.[0-9]+'
    return t

def t_TK_int(t):
    r'[0-9]+'
    return t

def t_error(t):
    errors.append(Error(t.lexer.lineno, t.lexer.lexpos + 1, TypeError.LEXICAL, f'Caracter no reconocido. «{t.value[0]}»'))
    t.lexer.skip(1)

import ply.lex as Scanner
scanner = Scanner.lex()
```

Descripción	Patrón	Expresión Regular	Ejemplo	Nombre del Token
Reservada begin	Palabra begin	begin	begin	RW_begin
Reservada end	Palabra end	end	end	RW_end
Reservada select	Palabra select	select	select	RW_select
Reservada from	Palabra from	from	from	RW_from
Reservada where	Palabra where	where	where	RW_where
Reservada declare	Palabra declare	declare	declare	RW_declare
Reservada set	Palabra set	set	set	RW_set
Reservada create	Palabra create	create	create	RW_create
Reservada data	Palabra data	data	data	RW_data
Reservada base	Palabra base	base	base	RW_base
Reservada use	Palabra use	use	use	RW_use
Reservada table	Palabra table	table	table	RW_table
Reservada primary	Palabra primary	primary	primary	RW_primary
Reservada foreing	Palabra foreing	foreing	foreing	RW_foreing
Reservada key	Palabra key	key	key	RW_key
Reservada reference	Palabra reference	reference	reference	RW_reference
Reservada alter	Palabra alter	alter	alter	RW_alter
Reservada add	Palabra add	add	add	RW_add
Reservada drop	Palabra drop	drop	drop	RW_drop

Reservada column	Palabra columna	column	column	RW_column
Reservada rename	Palabra rename	rename	rename	RW_rename
Reservada to	Palabra to	to	to	RW_to
Reservada insert	Palabra insert	insert	insert	RW_insert
Reservada into	Palabra into	into	into	RW_into
Reservada values	Palabra values	values	values	RW_values
Reservada as	Palabra as	as	as	RW_as
Reservada update	Palabra update	update	update	RW_update
Reservada truncate	Palabra truncate	truncate	truncate	RW_truncate
Reservada delete	Palabra delete	delete	delete	RW_delete
Reservada then	Palabra then	then	then	RW_then
Reservada when	Palabra when	when	when	RW_when
Reservada not	Palabra not	not	not	RW_not
Reservada if	Palabra if	if	if	RW_if
Reservada else	Palabra else	else	else	RW_else
Reservada case	Palabra case	case	case	RW_case
Reservada while	Palabra while	while	while	RW_while
Reservada for	Palabra for	for	for	RW_for
Reservada in	Palabra in	in	in	RW_in
Reservada loop	Palabra loop	loop	loop	RW_loop

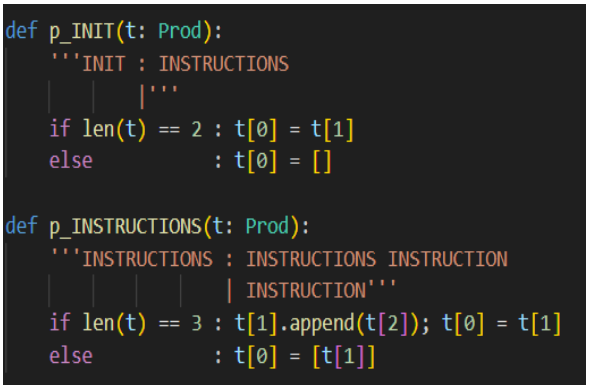
Reservada break	Palabra break	break	break	RW_break
Reservada continue	Palabra continue	continue	continue	RW_continue
Reservada function	Palabra function	function	function	RW_function
Reservada return	Palabra return	return	return	RW_return
Reservada returns	Palabra returns	returns	returns	RW_returns
Reservada procedure	Palabra procedure	procedure	procedure	RW_procedure
Reservada print	Palabra print	print	print	RW_print
Reservada concatenar	Palabra concatenar	concatenar	concatenar	RW_concatenar
Reservada subtraer	Palabra subtraer	subtraer	subtraer	RW_subtraer
Reservada hoy	Palabra hoy	hoy	hoy	RW_hoy
Reservada contar	Palabra contar	contar	contar	RW_contar
Reservada cast	Palabra cast	cast	cast	RW_cast
Signo más	Caracter +	+	+	TK_plus
Signo menos	Caracter -	-	-	TK_minus
Signo multiplicacion	Caracter *	*	*	TK_mult
Signo división	Caracter /	/	/	TK_div
Comparación	Caracter ==	==	==	TK_equequ
Diferente de	Caracter !=	!=	!=	TK_notequ
Menor o igual	Caracter <=	<=	<=	TK_lessequ
Mayor o igual	Caracter >=	>=	>=	TK_moreequ

Igual	Caracter =	=	=	TK_equal
Menor	Caracter <	<	<	TK_less
Mayor	Caracter >	>	>	TK_more
AND	Caracter AND	AND	AND	TK_and
OR	Caracter OR	OR	OR	TK_or
NOT	Caracter NOT	NOT	NOT	TK_not
Parentesis abierto	Caracter (	(	(	TK_lpar
Parentesis cerrado	Caracter)	)	)	TK_rpar
Llave abierto	Caracter {	{	{	TK_lbrc
Llave cerrado	Caracter }	}	}	TK_rbrc
Corchete abierto	Caracter [	[	[	TK_lbrk
Corchete cerrado	Caracter]	]	]	TK_rbrk
Coma	Caracter ,	,	,	TK_comma
Dos puntos	Caracter :	:	:	TK_colon
Puntos y coma	Caracter ;	;	;	TK_semicolon
Comentarios Simple	Caracter --	"--"([^\r\n]*)?	-- comentario simple	
Comentarios Multilineas	Caracter /**/	[/][][^*][]+([^/*][^*][]+)[/]	// comentario simple	
Caracteres Alfabéticos	Caracter: a, b, c, ..., y,z, A, B, ..., Y, Z	"({([^\n\"\\]|\.)}*)"	a, c, D, E	TK_nchar
Numeros enteros	Caracter: 0, 1, 2, ...	[0-9]+	0, 1, 2	Tk_int

Numeros decimales	Caracter: 0.5, 1.5, 2.5, ...	[0-9]+.[0-9]+	0.5, 1.5, 2.5	Tk_decimal
Identificadores	Secuencia de caracteres alfanumericos	"({([^\n\"\\]|\.)}*)"	Cadena, num_pares	TK_nvarchar
Caracteres numericos	Caracter: “dd/mm/yy”	"\'\d\d\d\d\-\d\d\-\d\d\'"	"\'\d\d\d\d\-\d\d\-\d\d\'"	TK_date
Caracteres numericos	Caracter: “dd/mm/yy”	"\'\d\d\-\d\d\-\d\d\d\d\ \d\d\:\d\d\'"	"\'\d\d\-\d\d\-\d\d\d\d\ \d\d\:\d\d\'"	TK_datetime

ANALIZADOR SINTACTICO(PARSER)

Gramática: Creación de las reglas sintácticas que el programa aceptará y leerá durante la ejecución de este mismo.



```
def p_INIT(t: Prod):
    '''INIT : INSTRUCTIONS
            |'''
    if len(t) == 2 : t[0] = t[1]
    else           : t[0] = []

def p_INSTRUCTIONS(t: Prod):
    '''INSTRUCTIONS : INSTRUCTIONS INSTRUCTION
                    | INSTRUCTION'''
    if len(t) == 3 : t[1].append(t[2]); t[0] = t[1]
    else           : t[0] = [t[1]]
```

```
def p_INSTRUCTION(t: Prod):
    '''INSTRUCTION  : CREATEDB TK_semicolon
                    | USEDB TK_semicolon
                    | CREATETABLE TK_semicolon
                    | ALTERTAB TK_semicolon
                    | DROPTAB TK_semicolon
                    | INSERTREG TK_semicolon
                    | UPDATETAB TK_semicolon
                    | TRUNCATETAB TK_semicolon
                    | DELETETAB TK_semicolon
                    | SELECT TK_semicolon
                    | DECLAREID TK_semicolon
                    | ASIGNID TK_semicolon
                    | IFSTRUCT TK_semicolon
                    | CASESTRUCT_S TK_semicolon
                    | WHILESTRUCT TK_semicolon
                    | FORSTRUCT TK_semicolon
                    | FUNCDEC TK_semicolon
                    | CALLFUNC TK_semicolon
                    | ENCAP TK_semicolon
                    | PRINT TK_semicolon
                    | RW_break TK_semicolon
                    | RW_continue TK_semicolon
                    | RW_return EXP TK_semicolon
                    | RW_return TK_semicolon'''
    types = ['RW_break', 'RW_continue', 'RW_return']
    if not t.slice[1].type in types                    : t[0] = t[1]
    elif t.slice[1].type == 'RW_break'                 : pass
    elif t.slice[1].type == 'RW_continue'              : pass
    elif t.slice[1].type == 'RW_return' and len(t) == 4 : t[0] = Return(t.lineno(1), t.lexpos(1), t[2])
    elif t.slice[1].type == 'RW_return'                : t[0] = Return(t.lineno(1), t.lexpos(1), None)
```

```python
# Crear DB
def p_CREATEDB(t: Prod):
    '''CREATEDB     : RW_create RW_data RW_base TK_field'''
    xml.createDataBase(t[4])

# Usar DB
def p_USEDB(t: Prod):
    '''USEDB    : RW_use TK_field'''
    setUsedDatabase(t[2])

# Declaración de Variables
def p_DECLAREID(t: Prod):
    '''DECLAREID    : RW_declare DECLIDS
                    | RW_declare TK_id TYPE TK_equal EXP'''
    if len(t) == 3:   t[0] = InitID(t.lineno(1), t.lexpos(1), t[2][0], t[2][1], None)
    elif len(t) == 6: t[0] = InitID(t.lineno(1), t.lexpos(1), t[2],    t[3],    t[5])

def p_DECLIDS(t: Prod):
    '''DECLIDS  : DECLIDS TK_comma DECLID
                | DECLID'''
    if len(t) == 4: t[1][0].append(t[3][0]); t[1][1].append(t[3][1]); t[0] = t[1]
    else:           t[0] = [[t[1][0]], [t[1][1]]]

def p_DECLID(t: Prod):
    '''DECLID : TK_id TYPE'''
    t[0] = [t[1], t[2]]

# Asignación de Variables
def p_ASIGNID(t: Prod):
    '''ASIGNID : RW_set TK_id TK_equal EXP'''
    t[0] = AsignID(t.lineno(1), t.lexpos(1), t[2], t[4])
```

```python
# Mostrar valores de Variables
def p_SELECT(t: Prod):
    '''SELECT   : RW_select FIELDS RW_from TK_field RW_where EXP
                | RW_select FIELDS RW_from TK_field
                | RW_select LIST_IDS'''
    if len(t) == 7   : t[0] = Select(t.lineno(1), t.lexpos(1), t[4], t[2], t[6])
    elif len(t) == 5 : t[0] = Select(t.lineno(1), t.lexpos(1), t[4], t[2], None)
    else             : t[0] = Select_prt(t.lineno(1), t.lexpos(1), t[2])

def p_FIELDS(t: Prod):
    '''FIELDS   : LIST_IDS
                | TK_mult'''
    t[0] = t[1]

def p_LIST_IDS(t: Prod):
    '''LIST_IDS : LIST_IDS TK_comma IDS
                | IDS'''
    if len(t) == 4 : t[1].append(t[3]); t[0] = t[1]
    else           : t[0] = [t[1]]

def p_IDS(t: Prod):
    '''IDS  : EXP RW_as TK_nvarchar
            | EXP RW_as TK_field
            | EXP'''
    if len(t) == 4 : t[0] = [t[1], t[3]]
    else           : t[0] = [t[1], '']

# Creación de Tablas
def p_CREATETABLE(t: Prod):
    '''CREATETABLE : RW_create RW_table TK_field TK_lpar ATTRIBUTES TK_rpar'''
    t[0] = CreateTable(t.lineno(1), t.lexpos(1), t[3], t[5])
```

```
def p_ATTRIBUTES(t: Prod):
    '''ATTRIBUTES   : ATTRIBUTES TK_comma ATTRIBUTE
                    | ATTRIBUTE'''
    if len(t) == 4: t[1].append(t[3]); t[0] = t[1]
    else:           t[0] = [t[1]]

def p_ATTRIBUTE(t: Prod):
    '''ATTRIBUTE    : TK_field TYPE TK_lpar TK_int TK_rpar PROPS
                    | TK_field TYPE PROPS
                    | TK_field TYPE TK_lpar TK_int TK_rpar
                    | TK_field TYPE
                    | RW_foreing RW_key TK_lpar TK_field TK_rpar RW_ref TK_field TK_lpar TK_field TK_rpar'''
    if len(t) == 7   : t[0] = Attribute(t.lineno(1), t.lexpos(1), t[1], t[2], t[4], t[6])
    elif len(t) == 4 : t[0] = Attribute(t.lineno(1), t.lexpos(1), t[1], t[2], None, t[3])
    elif len(t) == 6 : t[0] = Attribute(t.lineno(1), t.lexpos(1), t[1], t[2], t[4])
    elif len(t) == 3 : t[0] = Attribute(t.lineno(1), t.lexpos(1), t[1], t[2], None)
    else             : t[0] = ForeignKey(t.lineno(1), t.lexpos(1), t[4], t[7], t[9])

def p_PROPS(t: Prod):
    '''PROPS    : RW_not RW_null RW_primary RW_key FKEY
                | RW_not RW_null RW_primary RW_key
                | RW_primary RW_key RW_not RW_null FKEY
                | RW_primary RW_key RW_not RW_null
                | RW_not RW_null FKEY
                | RW_not RW_null
                | RW_primary RW_key FKEY
                | RW_primary RW_key
                | FKEY'''
    if len(t) == 5 or len(t) == 6 and t.slice[1].type == 'RW_not'     : t[0] = {'notNull': True,  'primaryKey': True }
    if len(t) == 5 or len(t) == 6 and t.slice[1].type == 'RW_primary' : t[0] = {'notNull': True,  'primaryKey': True }
    if len(t) == 3 or len(t) == 4 and t.slice[1].type == 'RW_not'     : t[0] = {'notNull': True,  'primaryKey': False}
    if len(t) == 3 or len(t) == 4 and t.slice[1].type == 'RW_primary' : t[0] = {'notNull': False, 'primaryKey': True }
    if len(t) == 2                                                    : t[0] = {'notNull': False, 'primaryKey': False}

    if len(t) == 6 or len(t) == 4 or len(t) == 2                      : t[0].update(t[len(t) - 1])
    else                                                              : t[0].update({'foreignKey': False})
```

```
def p_FKEY(t: Prod):
    '''FKEY : RW_ref TK_field TK_lpar TK_field TK_rpar'''
    t[0] = {'foreignKey': True, 'table': t[2], 'field': t[4]}

# Alter Table
def p_ALTERTAB(t: Prod):
    '''ALTERTAB : RW_alter RW_table TK_field ACTION'''
    t[0] = AlterTable(t.lineno(1), t.lexpos(1), t[3], t[4][0], t[4][1], t[4][2], t[4][3])

def p_ACTION(t: Prod):
    '''ACTION   : RW_add TK_field TYPE
                | RW_drop TK_field
                | RW_rename RW_to TK_field
                | RW_rename RW_column TK_field RW_to TK_field'''
    if   t.slice[1].type == 'RW_add'    : t[0] = [t[1],        t[2], None, t[3]]
    elif t.slice[1].type == 'RW_drop'   : t[0] = [t[1],        t[2], None, None]
    elif t.slice[2].type == 'RW_to'     : t[0] = [t[1] + t[2], t[3], None, None]
    elif t.slice[2].type == 'RW_column' : t[0] = [t[1] + t[2], t[3], t[5], None]

# Eliminar Tabla
def p_DROPTAB(t: Prod):
    '''DROPTAB : RW_drop RW_table TK_field'''
    t[0] = DropTable(t.lineno(1), t.lexpos(1), t[3])

# Insertar registros
def p_INSERTREG(t: Prod):
    '''INSERTREG : RW_insert RW_into TK_field TK_lpar LIST_ATTRIBS TK_rpar RW_values TK_lpar LIST_EXPS TK_rpar'''
    t[0] = InsertTable(t.lineno(1), t.lexpos(1), t[3], t[5], t[9])


def p_LIST_ATTRIBS(t: Prod):
    '''LIST_ATTRIBS : LIST_ATTRIBS TK_comma TK_field
                    | TK_field'''
    if len(t) == 4 : t[1].append(t[3]); t[0] = t[1]
    else           : t[0] = [t[1]]
```

```python
def p_LIST_EXPS(t: Prod):
    '''LIST_EXPS    : LIST_EXPS TK_comma EXP
                    | EXP'''
    if len(t) == 4 : t[1].append(t[3]); t[0] = t[1]
    else           : t[0] = [t[1]]

# Actualizar Tabla
def p_UPDATETAB(t: Prod):
    '''UPDATETAB : RW_update TK_field RW_set VALUESTAB RW_where EXP'''
    t[0] = UpdateTable(t.lineno(1), t.lexpos(1), t[2], t[4][0], t[4][1], t[6])

def p_VALUESTAB(t: Prod):
    '''VALUESTAB    : VALUESTAB TK_comma VALUETAB
                    | VALUETAB '''
    if len(t) == 4: t[1][0].append(t[3][0]); t[1][1].append(t[3][1]); t[0] = t[1]
    else:           t[0] = [[t[1][0]], [t[1][1]]]

def p_VALUETAB(t: Prod):
    '''VALUETAB : TK_field TK_equal EXP'''
    t[0] = [t[1], t[3]]

# Truncate
def p_TRUNCATETAB(t: Prod):
    '''TRUNCATETAB : RW_truncate RW_table TK_field'''
    t[0] = TruncateTable(t.lineno(1), t.lexpos(1), t[3])

# Eliminar Registros
def p_DELETETAB(t: Prod):
    '''DELETETAB : RW_delete RW_from TK_field RW_where EXP'''
    t[0] = DeleteTable(t.lineno(1), t.lexpos(1), t[3], t[5])
```

```python
# Estructura IF
def p_IFSTRUCT(t: Prod):
    '''IFSTRUCT : RW_if EXP RW_then INSTRUCTIONS RW_else INSTRUCTIONS RW_end RW_if
                | RW_if EXP RW_then INSTRUCTIONS RW_end RW_if
                | RW_if EXP RW_begin INSTRUCTIONS RW_end'''
    if len(t) == 9   : t[0] = If(t.lineno(1), t.lexpos(1), t[2], Block(t.lineno(1), t.lexpos(1), t[4]), Block(t.lineno(1), t.lexpos(1), t[6]))
    elif len(t) == 7 : t[0] = If(t.lineno(1), t.lexpos(1), t[2], Block(t.lineno(1), t.lexpos(1), t[4]), None)
    elif len(t) == 6 : t[0] = If(t.lineno(1), t.lexpos(1), t[2], Block(t.lineno(1), t.lexpos(1), t[4]), None)

# Estructura CASE
def p_CASESTRUCT_S(t: Prod):
    '''CASESTRUCT_S : RW_case EXP WHENELSE RW_end RW_as TK_field
                    | RW_case EXP WHENELSE RW_end RW_as TK_nvarchar
                    | RW_case EXP WHENELSE RW_end
                    | RW_case WHENELSE RW_end RW_as TK_field
                    | RW_case WHENELSE RW_end RW_as TK_nvarchar
                    | RW_case WHENELSE RW_end'''
    if len(t) == 7   : t[0] = Case(t.lineno(1), t.lexpos(1), t[2], t[3][0], t[3][1], t[6])
    elif len(t) == 5 : t[0] = Case(t.lineno(1), t.lexpos(1), t[2], t[3][0], t[3][1], None)
    elif len(t) == 6 : t[0] = Case(t.lineno(1), t.lexpos(1), None, t[2][0], t[2][1], t[5])
    elif len(t) == 4 : t[0] = Case(t.lineno(1), t.lexpos(1), None, t[2][0], t[2][1], None)

def p_WHENELSE(t: Prod):
    '''WHENELSE : WHENS ELSE
                | WHENS
                | ELSE'''
    if len(t) == 3                                  : t[0] = [t[1], t[2]]
    elif len(t) == 2 and t.slice[1].type == 'WHENS' : t[0] = [t[1], None]
    else                                            : t[0] = [None, t[1]]

def p_WHENS(t: Prod):
    '''WHENS    : WHENS WHEN
                | WHEN'''
    if len(t) == 3 : t[1].append(t[2]); t[0] = t[1]
    else           : t[0] = [t[1]]
```

```
def p_WHEN(t: Prod):
    '''WHEN : RW_when EXP RW_then EXP'''
    t[0] = When(t.lineno(1), t.lexpos(1), t[2], t[4])

def p_ELSE(t: Prod):
    '''ELSE : RW_else RW_then EXP'''
    t[0] = t[3]

# PRINT
def p_PRINT(t: Prod):
    '''PRINT : RW_print EXP'''

# Estructura WHILE
def p_WHILESTRUCT(t: Prod):
    '''WHILESTRUCT : RW_while EXP ENCAP'''
    t[0] = While(t.lineno(1), t.lexpos(1), t[2], t[3])

# Estructura FOR
def p_FORSTRUCT(t: Prod):
    '''FORSTRUCT : RW_for TK_id RW_in EXP TK_dot EXP ENCAP RW_loop'''

# Funciones y métodos
def p_FUNCDEC(t: Prod):
    '''FUNCDEC  : RW_create RW_function TK_field TK_lpar PARAMS TK_rpar RW_returns TYPE ENCAP
                | RW_create RW_function TK_field TK_lpar TK_rpar RW_returns TYPE ENCAP
                | RW_create RW_procedure TK_field PARAMS RW_as ENCAP
                | RW_create RW_procedure TK_field RW_as ENCAP
                | RW_create RW_procedure TK_field TK_lpar PARAMS TK_rpar ENCAP
                | RW_create RW_procedure TK_field ENCAP'''
    if len(t) == 10  : t[0] = Function(t.lineno(1), t.lexpos(1), t[3], t[5], t[9], t[8])
    elif len(t) == 9 : t[0] = Function(t.lineno(1), t.lexpos(1), t[3],   [], t[8], t[7])
    elif len(t) == 7 : t[0] = Function(t.lineno(1), t.lexpos(1), t[3], t[4], t[6], Type.NULL)
    elif len(t) == 6 : t[0] = Function(t.lineno(1), t.lexpos(1), t[3],   [], t[5], Type.NULL)
    elif len(t) == 8 : t[0] = Function(t.lineno(1), t.lexpos(1), t[3], t[5], t[7], Type.NULL)
    else             : t[0] = Function(t.lineno(1), t.lexpos(1), t[3],   [], t[4], Type.NULL)
```

```
def p_PARAMS(t: Prod):
    '''PARAMS   : PARAMS TK_comma PARAM
                | PARAM'''
    if len(t) == 4 : t[1].append(t[3]); t[0] = t[1]
    else           : t[0] = [t[1]]

def p_PARAM(t: Prod):
    '''PARAM    : TK_id RW_as TYPE'''
    t[0] = Parameter(t.lineno(1), t.lexpos(1), t[1], t[3])

# Encapsulamiento de Sentencias
def p_ENCAP(t: Prod):
    '''ENCAP    : RW_begin INSTRUCTIONS RW_end
                | RW_begin RW_end'''
    if len(t) == 4 : t[0] = Block(t.lineno(1), t.lexpos(1), t[2])
    else           : t[0] = Block(t.lineno(1), t.lexpos(1), [])

# Llamada a funciones y métodos
def p_CALLFUNC(t: Prod):
    '''CALLFUNC : TK_field TK_lpar ARGS TK_rpar
                | TK_field TK_lpar TK_rpar'''
    if len(t) == 5 : t[0] = CallFunction(t.lineno(1), t.lexpos(1), t[1], t[3])
    else           : t[0] = CallFunction(t.lineno(1), t.lexpos(1), t[1],  [] )

def p_ARGS(t: Prod):
    '''ARGS : ARGS TK_comma EXP
            | EXP'''
    if len(t) == 4 : t[1].append(t[3]); t[0] = t[1]
    else           : t[0] = [t[1]]
```

```
def p_EXP(t: Prod):
    '''EXP  : ARITHMETICS
            | RELATIONALS
            | LOGICS
            | CAST
            | NATIVEFUNC
            | CALLFUNC
            | TERNARY
            | TK_id
            | TK_field
            | TK_nvarchar
            | TK_int
            | TK_decimal
            | TK_date
            | TK_datetime
            | RW_null
            | TK_lpar EXP TK_rpar'''
    types = ['ARITHMETICS', 'RELATIONALS', 'LOGICS', 'CAST', 'NATIVEFUNC', 'CALLFUNC', 'TERNARY']
    if t.slice[1].type in types           : t[0] = t[1]
    elif t.slice[1].type == 'TK_id'       : t[0] = AccessID(t.lineno(1), t.lexpos(1), t[1])
    elif t.slice[1].type == 'TK_field'    : t[0] = Field(t.lineno(1), t.lexpos(1), t[1])
    elif t.slice[1].type == 'TK_nvarchar' : t[0] = Primitive(t.lineno(1), t.lexpos(1), t[1], Type.NVARCHAR)
    elif t.slice[1].type == 'TK_int'      : t[0] = Primitive(t.lineno(1), t.lexpos(1), t[1], Type.INT)
    elif t.slice[1].type == 'TK_decimal'  : t[0] = Primitive(t.lineno(1), t.lexpos(1), t[1], Type.DECIMAL)
    elif t.slice[1].type == 'TK_date'     : t[0] = Primitive(t.lineno(1), t.lexpos(1), t[1], Type.DATE)
    elif t.slice[1].type == 'TK_datetime' : t[0] = Primitive(t.lineno(1), t.lexpos(1), t[1], Type.DATETIME)
    elif t.slice[1].type == 'RW_null'     : t[0] = Primitive(t.lineno(1), t.lexpos(1), t[1], Type.NULL)
    else                                  : t[0] = t[2]
```

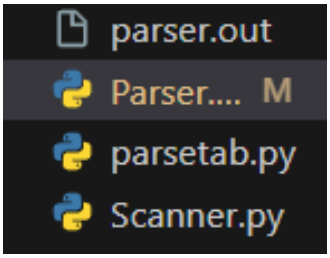
```
def p_ARITHMETICS(t: Prod):
    '''ARITHMETICS  : EXP TK_plus EXP
                    | EXP TK_minus EXP
                    | EXP TK_mult EXP
                    | EXP TK_div EXP
                    | EXP TK_mod EXP
                    | TK_minus EXP %prec TK_uminus'''
    if t.slice[1].type != 'TK_minus' : t[0] = Arithmetic(t.lineno(1), t.lexpos(1), t[1], t[2], t[3])
    else                             : t[0] = Arithmetic(t.lineno(1), t.lexpos(1), None, t[1], t[2])

def p_RELATIONALS(t: Prod):
    '''RELATIONALS  : EXP TK_equalequal EXP
                    | EXP TK_equal EXP
                    | EXP TK_notequal EXP
                    | EXP TK_lessequal EXP
                    | EXP TK_greatequal EXP
                    | EXP TK_less EXP
                    | EXP TK_great EXP'''
    t[0] = Relational(t.lineno(1), t.lexpos(1), t[1], t[2], t[3])

def p_LOGICS(t: Prod):
    '''LOGICS   : EXP TK_and EXP
                | EXP TK_or EXP
                | TK_not EXP'''
    if t.slice[2].type != 'RW_not'   : t[0] = Logic(t.lineno(1), t.lexpos(1), t[1], t[2], t[3])
    else                             : t[0] = Logic(t.lineno(1), t.lexpos(1), None, t[2], t[3])

def p_CAST(t: Prod):
    '''CAST : RW_cast TK_lpar EXP RW_as TYPE TK_rpar'''
    t[0] = Cast(t.lineno(1), t.lexpos(1), t[3], t[5])
```

```python
# Funciones Nativas
def p_NATIVEFUNC(t: Prod):
    '''NATIVEFUNC   : RW_concatenar TK_lpar EXP TK_comma EXP TK_rpar
                    | RW_substraer TK_lpar EXP TK_comma EXP TK_comma EXP TK_rpar
                    | RW_hoy TK_lpar TK_rpar'''
    if len(t) == 7   : t[0] = Concatenar(t.lineno(1), t.lexpos(1), t[3], t[5])
    elif len(t) == 9 : t[0] = Substraer(t.lineno(1), t.lexpos(1), t[3], t[5], t[7])
    else             : t[0] = Hoy(t.lineno(1), t.lexpos(1))

def p_TERNARY(t: Prod):
    '''TERNARY : RW_if TK_lpar EXP TK_comma EXP TK_comma EXP TK_rpar'''

def p_TYPE(t: Prod):
    '''TYPE : RW_int
            | RW_bit
            | RW_decimal
            | RW_date
            | RW_datetime
            | RW_nchar
            | RW_nvarchar'''
    if t.slice[1].type == 'RW_int'        : t[0] = Type.INT
    elif t.slice[1].type == 'RW_bit'      : t[0] = Type.BIT
    elif t.slice[1].type == 'RW_decimal'  : t[0] = Type.DECIMAL
    elif t.slice[1].type == 'RW_date'     : t[0] = Type.DATE
    elif t.slice[1].type == 'RW_datetime' : t[0] = Type.DATETIME
    elif t.slice[1].type == 'RW_nchar'    : t[0] = Type.NCHAR
    elif t.slice[1].type == 'RW_nvarchar' : t[0] = Type.NVARCHAR

from interpreter.Scanner import *

def p_error(t: LexToken):
    errors.append(Error(t.lineno, t.lexpos + 1, TypeError.SYNTAX, f'No se esperaba «{t.value}»'))

import ply.yacc as Parser
parser = Parser.yacc()
```

Precedencias: Durante la ejecución el programa tomara algunas reglas léxicas (Token's) como prioridad.

```python
precedence = (
    ('left', 'TK_or'),
    ('left', 'TK_and'),
    ('right', 'TK_not'),
    ('left', 'TK_equalequal', 'TK_equal', 'TK_notequal'),
    ('left', 'TK_less', 'TK_lessequal', 'TK_great', 'TK_greatequal'),
    ('left', 'TK_plus', 'TK_minus'),
    ('left', 'TK_mult', 'TK_div', 'TK_mod'),
    ('right', 'TK_uminus'),
)
```

Archivos generados por la herramienta PLY:



PATRON INTERPRETE

Para la creación de este proyecto se utilizo el patrón interprete como modelo a seguir para la implementación de las deferentes clases donde se trabajaron las funcionalidades de este mismo.

Clase Expresión: La clase Expresión tiene un constructor que inicializa atributos como la línea y columna del código fuente donde aparece la expresión, así como el tipo de la expresión (typeExp). Además, cuenta con atributos para gestionar etiquetas de salto condicional (trueLabel y falseLabel), que pueden ser utilizadas en la generación de código intermedio o código ensamblador.

Los métodos abstractos como setField, execute, compile, y ast definen operaciones clave para las expresiones. Por ejemplo, el método execute se encarga de ejecutar la expresión, el método compile se utiliza para compilar la expresión, y ast se emplea para construir un árbol sintáctico abstracto (AST) correspondiente a la expresión.

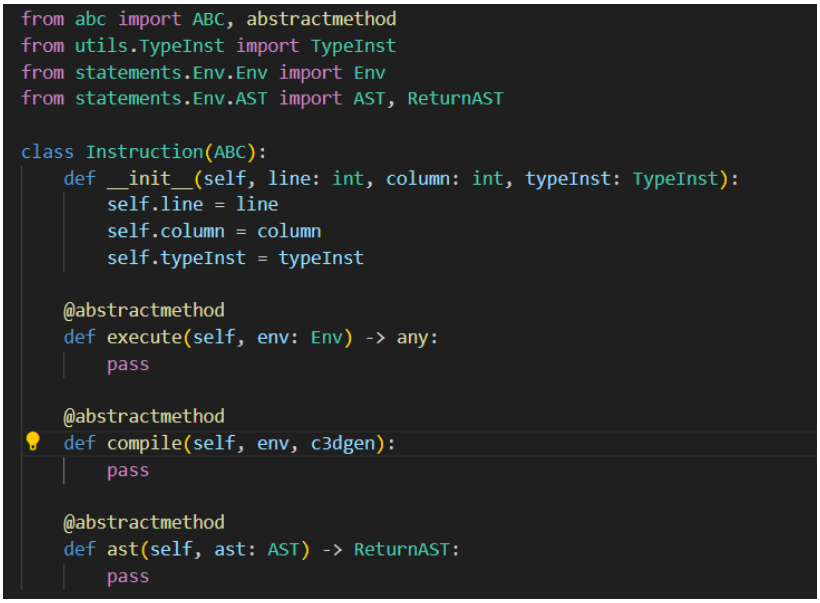
```python
from abc import ABC, abstractmethod
from utils.TypeExp import TypeExp

class Expression(ABC):
    def __init__(self, line: int, column: int, typeExp: TypeExp):
        self.line = line
        self.column = column
        self.typeExp = typeExp
        self.trueLabel = ''
        self.falseLabel = ''

    @abstractmethod
    def setField(self, field):
        pass

    @abstractmethod
    def execute(self, env):
        pass

    @abstractmethod
    def compile(self, env, c3dgen):
        pass

    @abstractmethod
    def ast(self, ast):
        pass
```

Clase Instrucción: La clase Instruction tiene un constructor que inicializa atributos como la línea y columna del código fuente donde aparece la instrucción, así como el tipo de la instrucción (typeInst). También, incluye tres métodos abstractos que deben ser implementados por las clases derivadas para adaptarse a la lógica específica de cada instrucción.

execute: Este método se encarga de ejecutar la instrucción en el contexto de un entorno (env). La función devuelve algún valor, y su implementación variará según el tipo específico de instrucción.

compile: Este método se utiliza para compilar la instrucción. La implementación de este método será específica para cada tipo de instrucción y estará vinculada al proceso de generación de código intermedio o código ensamblador.

ast: Este método se utiliza para construir un árbol sintáctico abstracto (AST) correspondiente a la instrucción. La implementación de este método contribuirá a la construcción de la estructura del AST que representa el programa.



```python
from abc import ABC, abstractmethod
from utils.TypeInst import TypeInst
from statements.Env.Env import Env
from statements.Env.AST import AST, ReturnAST

class Instruction(ABC):
    def __init__(self, line: int, column: int, typeInst: TypeInst):
        self.line = line
        self.column = column
        self.typeInst = typeInst

    @abstractmethod
    def execute(self, env: Env) -> any:
        pass

    @abstractmethod
    def compile(self, env, c3dgen):
        pass

    @abstractmethod
    def ast(self, ast: AST) -> ReturnAST:
        pass
```

Las expresiones que el programa utilizara son las siguientes:

AccessID

La clase AccessID representa una expresión de acceso a un identificador (variable o campo) en un programa. Su propósito es permitir la ejecución de esta expresión, devolviendo el valor y el tipo correspondientes al identificador en un entorno dado. Además, la clase incluye métodos abstractos para la generación de código intermedio (C3D) y la construcción del árbol sintáctico abstracto (AST), aunque no hay implementaciones específicas en la clase AccessID para estos métodos, ya que se espera que se implementen en sus clases derivadas. En caso de que el identificador no exista en el entorno, la expresión devuelve un valor nulo (NULL). La clase se utiliza para representar y manipular expresiones de acceso a identificadores durante el análisis y ejecución de programas.

```python
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.Objects.Table import Field
from statements.Abstracts.Expression import Expression
from statements.Env.Symbol import Symbol
from utils.TypeExp import TypeExp
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class AccessID(Expression):
    def __init__(self, line: int, column: int, id: str):
        super().__init__(line, column, TypeExp.ACCESS_ID)
        self.id = id

    def setField(self, _: dict[str, Field]):
        pass

    def execute(self, env: Env) -> ReturnType:
        value: Symbol | None = env.getValue(self.id)
        if value:
            return ReturnType(value.value, value.type)
        return ReturnType('NULL', Type.NULL)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="{self.id}"];'
        return ReturnAST(dot, id)
```

Arithmetic

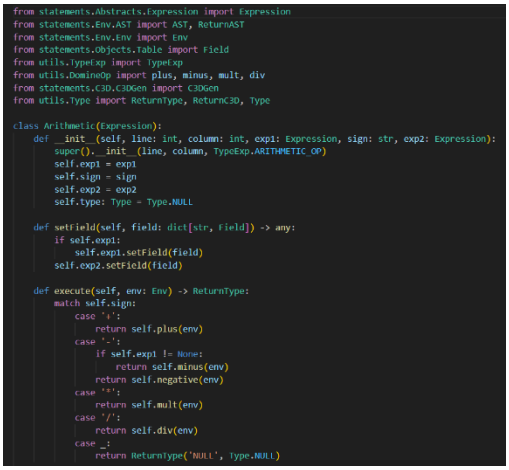
El código define una clase llamada Arithmetic, que hereda de la clase abstracta Expression. Esta clase representa una expresión aritmética en un lenguaje de programación. La expresión aritmética puede ser una suma (+), resta (-), multiplicación (*), o división (/) de dos subexpresiones.

La clase tiene un constructor que recibe información sobre la posición en el código fuente (line y column), así como las dos subexpresiones (exp1 y exp2) y el operador aritmético (sign). Durante la ejecución, la clase determina el tipo de operación y realiza la operación correspondiente en función de los tipos de las subexpresiones. La ejecución puede resultar en un error si los tipos no son compatibles para la operación aritmética.

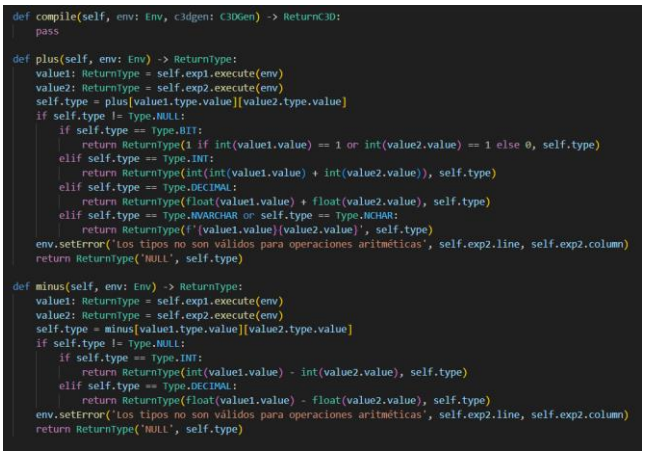
La clase también implementa el método ast para la construcción del árbol sintáctico abstracto (AST). Cada instancia de la clase contribuye a la construcción del AST,

representando el operador aritmético como un nodo en el árbol y conectándolo a los nodos de las subexpresiones.

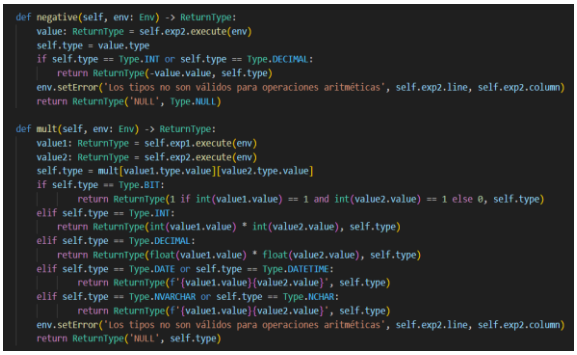
Es importante destacar que el método compile aún no está implementado, y la lógica de generación de código intermedio (C3D) para esta expresión aritmética debería agregarse en futuras implementaciones.



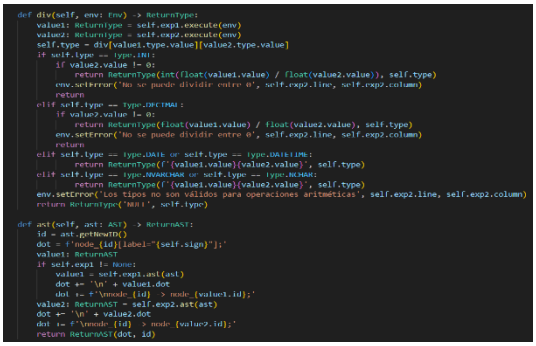
```python
from statements.Abstracts.Expression import Expression
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.Objects.Table import Field
from utils.TypeExp import TypeExp
from utils.DominoOp import plus, minus, mult, div
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class Arithmetic(Expression):
    def __init__(self, line: int, column: int, exp1: Expression, sign: str, exp2: Expression):
        super().__init__(line, column, TypeExp.ARITHMETIC_OP)
        self.exp1 = exp1
        self.sign = sign
        self.exp2 = exp2
        self.type: Type = Type.NULL

    def setField(self, field: dict[str, Field]) -> any:
        if self.exp1:
            self.exp1.setField(field)
        self.exp2.setField(field)

    def execute(self, env: Env) -> ReturnType:
        match self.sign:
            case '+':
                return self.plus(env)
            case '-':
                if self.exp1 != None:
                    return self.minus(env)
                return self.negative(env)
            case '*':
                return self.mult(env)
            case '/':
                return self.div(env)
            case _:
                return ReturnType('NULL', Type.NULL)
```



```python
def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
    pass

def plus(self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    self.type = plus[value1.type.value][value2.type.value]
    if self.type != Type.NULL:
        if self.type == Type.BIT:
            return ReturnType(1 if int(value1.value) == 1 or int(value2.value) == 1 else 0, self.type)
        elif self.type == Type.INT:
            return ReturnType(int(int(value1.value) + int(value2.value)), self.type)
        elif self.type == Type.DECIMAL:
            return ReturnType(float(value1.value) + float(value2.value), self.type)
        elif self.type == Type.NVARCHAR or self.type == Type.NCHAR:
            return ReturnType(f'{value1.value}{value2.value}', self.type)
    env.setError('Los tipos no son válidos para operaciones aritméticas', self.exp2.line, self.exp2.column)
    return ReturnType('NULL', self.type)

def minus(self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    self.type = minus[value1.type.value][value2.type.value]
    if self.type != Type.NULL:
        if self.type == Type.INT:
            return ReturnType(int(value1.value) - int(value2.value), self.type)
        elif self.type == Type.DECIMAL:
            return ReturnType(float(value1.value) - float(value2.value), self.type)
    env.setError('Los tipos no son válidos para operaciones aritméticas', self.exp2.line, self.exp2.column)
    return ReturnType('NULL', self.type)
```



```python
def negative(self, env: Env) -> ReturnType:
    value: ReturnType = self.exp2.execute(env)
    self.type = value.type
    if self.type == Type.INT or self.type == Type.DECIMAL:
        return ReturnType(-value.value, self.type)
    env.setError('Los tipos no son válidos para operaciones aritméticas', self.exp2.line, self.exp2.column)
    return ReturnType('NULL', Type.NULL)

def mult(self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    self.type = mult[value1.type.value][value2.type.value]
    if self.type == Type.BIT:
            return ReturnType(1 if int(value1.value) == 1 and int(value2.value) == 1 else 0, self.type)
    elif self.type == Type.INT:
        return ReturnType(int(value1.value) * int(value2.value), self.type)
    elif self.type == Type.DECIMAL:
        return ReturnType(float(value1.value) * float(value2.value), self.type)
    elif self.type == Type.DATE or self.type == Type.DATETIME:
            return ReturnType(f'{value1.value}{value2.value}', self.type)
    elif self.type == Type.NVARCHAR or self.type == Type.NCHAR:
            return ReturnType(f'{value1.value}{value2.value}', self.type)
    env.setError('Los tipos no son válidos para operaciones aritméticas', self.exp2.line, self.exp2.column)
    return ReturnType('NULL', self.type)
```

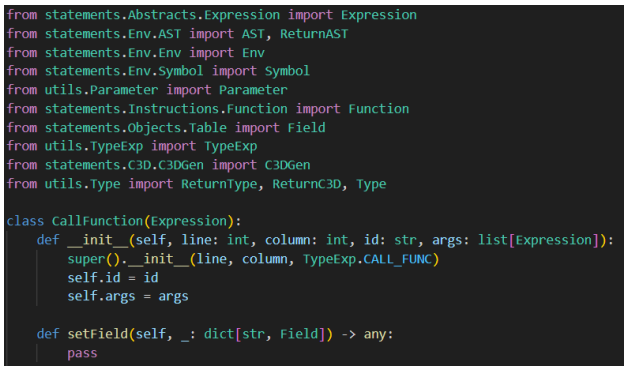


```python
def div(self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    self.type = div[value1.type.value][value2.type.value]
    if self.type == type.INT:
        if value2.value != 0:
            return ReturnType(int(float(value1.value) / float(value2.value)), self.type)
        env.setError('No se puede dividir entre 0', self.exp2.line, self.exp2.column)
        return
    elif self.type == Type.DECIMAL:
        if value2.value != 0:
            return ReturnType(float(value1.value) / float(value2.value), self.type)
        env.setError('No se puede dividir entre 0', self.exp2.line, self.exp2.column)
        return
    elif self.type == type.DATE or self.type == type.DATETIME:
            return ReturnType(f'{value1.value}{value2.value}', self.type)
    elif self.type == type.NVARCHAR or self.type == type.NCHAR:
            return ReturnType(f'{value1.value}{value2.value}', self.type)
    env.setError('Los tipos no son válidos para operaciones aritméticas', self.exp2.line, self.exp2.column)
    return ReturnType('NULL', self.type)

def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="{self.sign}"];'
    value1: ReturnAST
    if self.exp1 != None:
        value1 = self.exp1.ast(ast)
        dot += '\n' + value1.dot
        dot += f'\nnode_{id} -> node_{value1.id};'
    value2: ReturnAST = self.exp2.ast(ast)
    dot += '\n' + value2.dot
    dot += f'\nnode_{id} -> node_{value2.id};'
    return ReturnAST(dot, id)
```

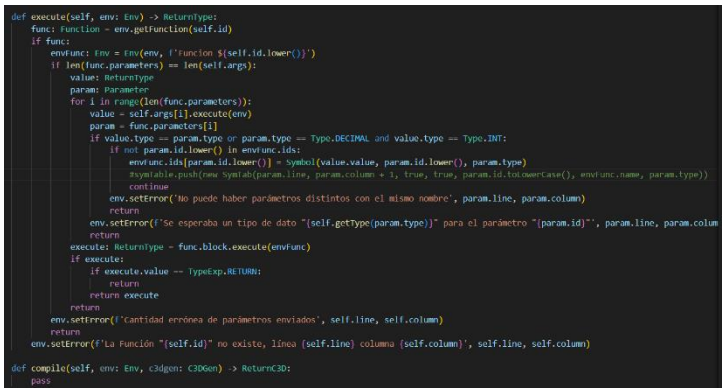
CallFunction

La función CallFunction representa una llamada a una función en un lenguaje de programación. La función toma como argumentos el nombre de la función (id) y una lista de expresiones que representan los argumentos de la función (args).

La función también implementa un método ast para la construcción del árbol sintáctico abstracto (AST). Cada instancia de la clase contribuye a la construcción del AST, representando la llamada a la función como un nodo y conectándolo a los nodos de las expresiones de los argumentos.



```python
from statements.Abstracts.Expression import Expression
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.Env.Symbol import Symbol
from utils.Parameter import Parameter
from statements.Instructions.Function import Function
from statements.Objects.Table import Field
from utils.TypeExp import TypeExp
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class CallFunction(Expression):
    def __init__(self, line: int, column: int, id: str, args: list[Expression]):
        super().__init__(line, column, TypeExp.CALL_FUNC)
        self.id = id
        self.args = args

    def setField(self, _: dict[str, Field]) -> any:
        pass
```



```python
def execute(self, env: Env) -> ReturnType:
    func: Function = env.getFunction(self.id)
    if func:
        envFunc: Env = Env(env, f'Función ${self.id.lower()}')
        if len(func.parameters) == len(self.args):
            value: ReturnType
            param: Parameter
            for i in range(len(func.parameters)):
                value = self.args[i].execute(env)
                param = func.parameters[i]
                if value.type == param.type or param.type == Type.DECIMAL and value.type == Type.INT:
                    if not param.id.lower() in envFunc.ids:
                        envFunc.ids[param.id.lower()] = Symbol(value.value, param.id.lower(), param.type)
                        #symtable.push(new Symtab(param.line, param.column + 1, true, true, param.id.toLowerCase(), envFunc.name, param.type))
                        continue
                    env.setError('No puede haber parámetros distintos con el mismo nombre', param.line, param.column)
                    return
                env.setError(f'Se esperaba un tipo de dato "{self.getType(param.type)}" para el parámetro "{param.id}"', param.line, param.colum
                return
            execute: ReturnType = func.block.execute(envFunc)
            if execute:
                if execute.value == TypeExp.RETURN:
                    return
                return execute
            return
        env.setError(f'Cantidad errónea de parámetros enviados', self.line, self.column)
        return
    env.setError(f'La Función "{self.id}" no existe, línea {self.line} columna {self.column}', self.line, self.column)

def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
    pass
```

```python
    pass

def getType(type: Type) -> str:
    match type:
        case Type.INT:
            return "INT"
        case Type.DECIMAL:
            return "DECIMAL"
        case Type.NVARCHAR:
            return "NVARCHAR"
        case Type.BOOLEAN:
            return "BOOLEAN"
        case Type.DATE:
            return "DATE"
        case Type.TABLE:
            return "TABLE"
        case _:
            return "NULL"

def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="CALL FUNC"]'
    dot += f'\nnode_{id}_name[label="{self.id}"]'
    dot += f'\nnode_{id} -> node_{id}_name'
    param: ReturnAST
    if len(self.args) > 0:
        for i in range(len(self.args)):
            param = self.args[i].ast(ast)
            dot += '\n' + param.dot
            dot += f'\nnode_{id}_name -> node_{param.id}'
    return ReturnAST(dot, id)
```

Cast

La clase Cast representa una operación de conversión de tipo (cast) en un lenguaje de programación. La instancia de esta clase se crea con tres parámetros: la expresión que se va a convertir (value), el tipo al que se desea convertir (destinyType), y la posición en el código fuente (línea y columna). La clase implementa métodos para la ejecución de la conversión, la construcción del AST y otros aspectos.

```python
import re
from statements.Objects.Table import Field
from utils.TypeExp import TypeExp
from statements.Env.AST import AST, ReturnAST
from statements.Abstracts.Expression import Expression
from statements.Env.Env import Env
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class Cast(Expression):
    def __init__(self, line: int, column: int, value: Expression, destinyType: Type):
        super().__init__(line, column, TypeExp.CAST)
        self.value = value
        self.destinyType = destinyType

    def setField(self, _: dict[str, Field]) -> any:
        pass

    def execute(self, env: Env) -> ReturnType:
        value: ReturnType = self.value.execute(env)
        if value.value == Type.BIT:
            if self.destinyType == Type.INT:
                return ReturnType(int(value.value), Type.INT)
            if self.destinyType == Type.NVARCHAR:
                return ReturnType(str(value.
            if self.destinyType == Type.NCHA
                return ReturnType(str(value.value), Type.NCHAR)
            env.setError(f'No hay casteo de "{self.getType(value.type)}" a "{self.getType(self.destinyType)}"', self.value.line, self.value.column)
            return ReturnType('NULL', Type.NULL)
```

(variable) value: Literal[Type.BIT]

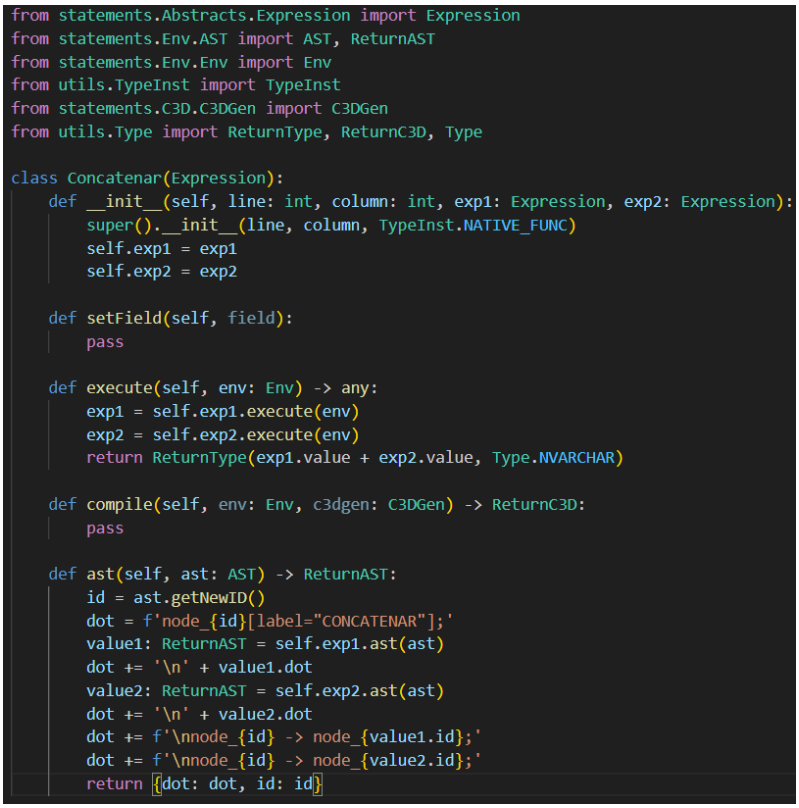
```python
        if value.type == Type.INT:
            if self.destinyType == Type.DECIMAL:
                return ReturnType(float(value.value), Type.DECIMAL)
            if self.destinyType == Type.NVARCHAR:
                return ReturnType(str(value.value), Type.NVARCHAR)
            env.setError(f'No hay casteo de "{self.getType(value.type)}" a "{self.getType(self.destinyType)}"', self.value.line, self.value.column)
            return ReturnType('NULL', Type.NULL)

        if value.type == Type.DECIMAL:
            if self.destinyType == Type.INT:
                return ReturnType(int(value.value), Type.INT)
            if self.destinyType == Type.NVARCHAR:
                return ReturnType(str(value.value), Type.NVARCHAR)
            if self.destinyType == Type.NCHAR:
                return ReturnType(str(value.value), Type.NCHAR)
            env.setError(f'No hay casteo de "{self.getType(value.type)}" a "{self.getType(self.destinyType)}"', self.value.line, self.value.column)
            return ReturnType('NULL', Type.NULL)

        if value.type == Type.DATE:
            if self.destinyType == Type.NVARCHAR:
                return ReturnType(str(value.value), Type.NVARCHAR)
            if self.destinyType == Type.NCHAR:
                return ReturnType(str(value.value), Type.NCHAR)
            env.setError(f'No hay casteo de "{self.getType(value.type)}" a "{self.getType(self.destinyType)}"', self.value.line, self.value.column)
            return ReturnType('NULL', Type.NULL)

        if value.type == Type.NVARCHAR:
            if self.destinyType == Type.INT:
                asciiz = sum(ord(character) for character in value.value)
                return ReturnType(int(asciiz), Type.INT)
            if self.destinyType == Type.BOOLEAN:
                return ReturnType(value.value.toLowerCase() == 'true', Type.BOOLEAN)
            env.setError(f'No hay casteo de "{self.getType(value.type)}" a "{self.getType(self.destinyType)}"', self.value.line, self.value.column)
            return ReturnType('NULL', Type.NULL)
```

```python
def getType(self, type: Type) -> str:
    match type:
        case Type.INT:
            return "INT"
        case Type.DECIMAL:
            return "DECIMAL"
        case Type.NVARCHAR:
            return "NVARCHAR"
        case Type.BOOLEAN:
            return "BOOLEAN"
        case Type.DATE:
            return "DATE"
        case Type.TABLE:
            return "TABLE"
        case _:
            return "NULL"

def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="CAST"];'
    value1: ReturnAST = self.value.ast(ast)
    dot += '\n' + value1.dot
    dot += f'\nnode_{id}_type[label="{self.getType(self.destinyType)}"];'
    dot += f'\nnode_{id} -> node_{value1.id};'
    dot += f'\nnode_{id} -> node_{id}_type;'
    return ReturnAST(dot, id)
```

```python
    if value.type == Type.NVARCHAR:
        if self.destinyType == Type.INT:
            asciiz = sum(ord(character) for character in value.value)
            return ReturnType(int(asciiz), Type.INT)
        if self.destinyType == Type.BOOLEAN:
            return ReturnType(value.value.toLowerCase() == 'true', Type.BOOLEAN)
        env.setError(f'No hay casteo de "{self.getType(value.type)}" a "{self.getType(self.destinyType)}"', self.value.line, self.value.column)
        return ReturnType('NULL', Type.NULL)

    if value.type == Type.NCHAR:
        if self.destinyType == Type.INT:
            asciiz = sum(ord(character) for character in value.value)
            return ReturnType(int(asciiz), Type.INT)
        if self.destinyType == Type.BOOLEAN:
            return ReturnType(value.value.toLowerCase() == 'true', Type.BOOLEAN)
        env.setError(f'No hay casteo de "{self.getType(value.type)}" a "{self.getType(self.destinyType)}"', self.value.line, self.value.column)
        return ReturnType('NULL', Type.NULL)
    env.setError(f'No hay casteo de "{self.getType(value.type)}" a "{self.getType(self.destinyType)}"', self.value.line, self.value.column)
    return ReturnType('NULL', Type.NULL)

def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
    pass
```

Concatenar

La clase Concatenar representa una operación de concatenación de cadenas en un lenguaje de programación. La instancia de esta clase se crea con tres parámetros: dos expresiones que se van a concatenar (exp1 y exp2), y la posición en el código fuente (línea y columna). La clase implementa métodos para la ejecución de la concatenación, la construcción del AST y otros aspectos, la clase Concatenar encapsula la lógica para realizar la operación de concatenación de cadenas en el contexto de un lenguaje de programación. La ejecución de la operación se realiza en tiempo de ejecución, y la clase se encarga de construir nodos en el AST para representar la operación de concatenación.



```python
from statements.Abstracts.Expression import Expression
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from utils.TypeInst import TypeInst
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class Concatenar(Expression):
    def __init__(self, line: int, column: int, exp1: Expression, exp2: Expression):
        super().__init__(line, column, TypeInst.NATIVE_FUNC)
        self.exp1 = exp1
        self.exp2 = exp2

    def setField(self, field):
        pass

    def execute(self, env: Env) -> any:
        exp1 = self.exp1.execute(env)
        exp2 = self.exp2.execute(env)
        return ReturnType(exp1.value + exp2.value, Type.NVARCHAR)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="CONCATENAR"];'
        value1: ReturnAST = self.exp1.ast(ast)
        dot += '\n' + value1.dot
        value2: ReturnAST = self.exp2.ast(ast)
        dot += '\n' + value2.dot
        dot += f'\nnode_{id} -> node_{value1.id};'
        dot += f'\nnode_{id} -> node_{value2.id};'
        return {dot: dot, id: id}
```

Field

La clase Field representa un campo (o identificador) en un lenguaje de programación, la clase Field encapsula la lógica para acceder a campos en un lenguaje de programación. Su método execute determina si el campo es un nombre de campo o un identificador y realiza la acción correspondiente, devolviendo un objeto ReturnType con el resultado de la ejecución. Además, el método ast construye un nodo en el AST para representar el campo en la estructura del programa.

```python
from utils.TypeExp import TypeExp
from statements.Env.AST import AST, ReturnAST
from statements.Abstracts.Expression import Expression
from statements.Env.Env import Env
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class Field(Expression):
    def __init__(self, line: int, column: int, id: str):
        super().__init__(line, column, TypeExp.FIELD)
        self.id = id
        self.field: dict[str, any] = {}
        self.isFieldName: bool = False

    def setIsFieldName(self, isFieldName: bool):
        self.isFieldName = isFieldName

    def setField(self, field: dict[str, any]) -> any:
        self.field = field

    def execute(self, env: Env) -> ReturnType:
        if not self.isFieldName:
            if self.id.lower() in self.field:
                return self.field[self.id.lower()].values[0].getData()
            env.setError(f'No existe el campo {self.id.lower()}', self.line, self.column)
            return ReturnType('NULL', Type.NULL)
        return ReturnType(self.id, Type.NULL)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="{self.id}"];'
        return ReturnAST(dot, id)
```

Hoy

La clase Hoy representa una expresión en un lenguaje de programación que devuelve la fecha y hora actuales, la clase Hoy encapsula la lógica para obtener la fecha y hora actuales. Su método execute devuelve un objeto ReturnType con la fecha y hora formateadas como una cadena en el formato especificado. Además, el método ast construye un nodo en el AST para representar la expresión en la estructura del programa.

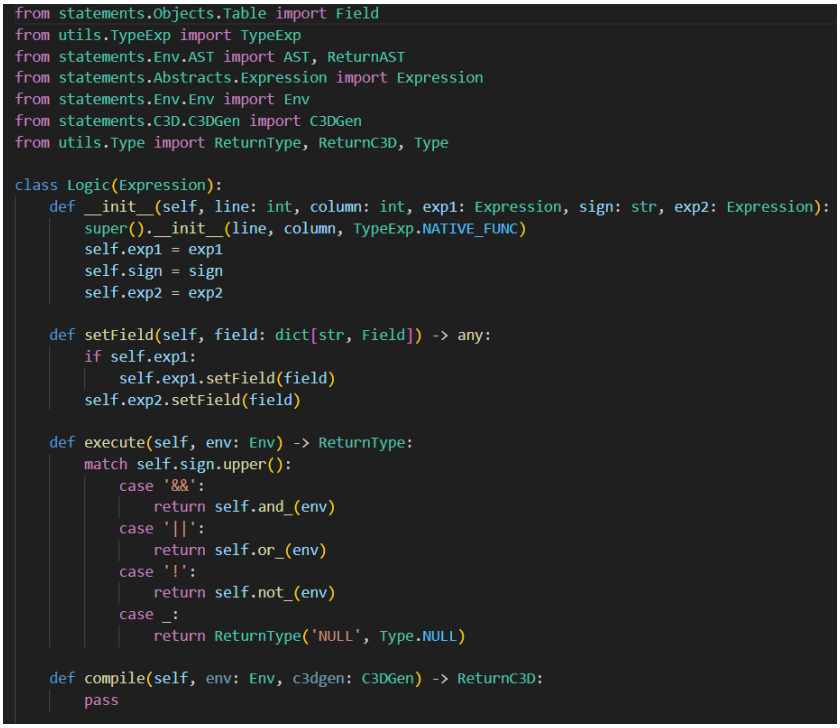
```python
from statements.Abstracts.Expression import Expression
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from utils.TypeInst import TypeInst
from datetime import datetime
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class Hoy(Expression):
    def __init__(self, line: int, column: int):
        super().__init__(line, column, TypeInst.NATIVE_FUNC)

    def setField(self, field):
        pass

    def execute(self, _: Env) -> any:
        dateT = datetime.now()
        f = "%d-%m-%Y %H:%M"
        return ReturnType(dateT.strftime(f), Type.NVARCHAR)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="HOY"];'
        return ReturnAST(dot, id)
```

Logic

La clase Logic en este código representa expresiones lógicas en un lenguaje de programación. Se instancian objetos de esta clase con información sobre su posición en el código fuente y el tipo de instrucción. La lógica de la expresión se evalúa a través del método execute, que interpreta operadores lógicos como && (AND), || (OR) y ! (NOT). La ejecución de estas operaciones se realiza mediante métodos específicos (and_, or_, not_) que calculan y devuelven resultados lógicos en forma de objetos ReturnType con el tipo Type.BOOLEAN.

La clase también proporciona funcionalidades para la construcción del árbol sintáctico abstracto (AST) mediante el método ast. Este método genera nodos en el AST para representar la estructura de la expresión lógica, conectando nodos según la relación de la expresión. Además, la clase incluye un método setField que permite asignar un diccionario de campos a la instancia, lo que puede ser útil para evaluar expresiones que involucran nombres de campos.

Aunque la compilación específica no está implementada en el método compile, la clase Logic desempeña un papel central en la representación y evaluación de expresiones lógicas, contribuyendo a la capacidad del lenguaje de programación para manejar operaciones lógicas de manera coherente y estructurada.



```python
from statements.Objects.Table import Field
from utils.TypeExp import TypeExp
from statements.Env.AST import AST, ReturnAST
from statements.Abstracts.Expression import Expression
from statements.Env.Env import Env
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class Logic(Expression):
    def __init__(self, line: int, column: int, exp1: Expression, sign: str, exp2: Expression):
        super().__init__(line, column, TypeExp.NATIVE_FUNC)
        self.exp1 = exp1
        self.sign = sign
        self.exp2 = exp2

    def setField(self, field: dict[str, Field]) -> any:
        if self.exp1:
            self.exp1.setField(field)
        self.exp2.setField(field)

    def execute(self, env: Env) -> ReturnType:
        match self.sign.upper():
            case '&&':
                return self.and_(env)
            case '||':
                return self.or_(env)
            case '!':
                return self.not_(env)
            case _:
                return ReturnType('NULL', Type.NULL)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass
```

```python
def and_(self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    self.type = Type.BOOLEAN
    return ReturnType(value1.value and value2.value, self.type)

def or_(self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    self.type = Type.BOOLEAN
    return ReturnType(value1.value or value2.value, self.type)

def not_(self, env: Env) -> ReturnType:
    value: ReturnType = self.exp2.execute(env)
    self.type = Type.BOOLEAN
    return ReturnType(not value.value, self.type)

def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="{self.sign}"];'
    value1: ReturnAST
    if self.exp1 != None:
        value1 = self.exp1.ast(ast)
        dot += '\n' + value1.dot
        dot += f'\nnode_{id} -> node_{value1.id};'
    value2: ReturnAST = self.exp2.ast(ast)
    dot += '\n' + value2.dot
    dot += f'\nnode_{id} -> node_{value2.id};'
    return ReturnAST(dot, id)
```

Primitive

La clase Primitive en este código representa expresiones que contienen valores primitivos en un lenguaje de programación. Estos valores pueden ser de tipos como INT, DECIMAL, DATE, DATETIME, o cualquier otro tipo no especificado. Los objetos de esta clase contienen información sobre la posición en el código fuente, el valor y el tipo del primitivo.

El método execute evalúa y devuelve el primitivo con el tipo correcto. Se realiza un manejo especial para los tipos de cadena, donde se reemplazan secuencias de escape como \\n, \\t, \\", \\', y \\\\. Además, para las cadenas, se eliminan las barras invertidas dobles que se usan para escapar caracteres especiales en el código fuente.

El método compile genera código de tres direcciones (C3D) para representar el primitivo en la memoria. Para las cadenas, se utiliza un enfoque especial que asigna cada carácter en el heap de memoria y devuelve un temporizador que apunta al inicio de la cadena.

La clase también contribuye a la construcción del árbol sintáctico abstracto (AST) mediante el método ast, que crea un nodo en el AST para representar el primitivo. Este nodo tiene como etiqueta el valor del primitivo.

```python
from statements.Abstracts.Expression import Expression
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.Objects.Table import Field
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type
from utils.TypeExp import TypeExp

class Primitive(Expression):
    def __init__(self, line: int, column: int, value: any, type: Type):
        super().__init__(line, column, TypeExp.PRIMITIVE)
        self.value = value
        self.type = type

    def setField(self, _: dict[str, Field]) -> any:
        pass

    def execute(self, _: Env) -> ReturnType:
        match self.type:
            case Type.INT:
                return ReturnType(int(self.value), self.type)
            case Type.DECIMAL:
                return ReturnType(float(self.value), self.type)
            case Type.DATE:
                return ReturnType(str(self.value), self.type)
            case Type.DATETIME:
                return ReturnType(str(self.value), self.type)
            case _:
                self.value = self.value.replace('\\n', '\n')
                self.value = self.value.replace('\\t', '\t')
                self.value = self.value.replace('\\"', '\"')
                self.value = self.value.replace("\\'", '\'')
                self.value = self.value.replace('\\\\', '\\')
                return ReturnType(self.value, self.type)
```

```python
def compile(self, _: Env, c3dgen: C3DGen) -> ReturnC3D:
    match self.type:
        case Type.INT:
            return ReturnC3D(isTmp = False, strValue = str(self.value), type = self.type)
        case Type.DECIMAL:
            return ReturnC3D(isTmp = False, strValue = str(self.value), type = self.type)
        case _:
            self.value = self.value.replace('\\n', '\n')
            self.value = self.value.replace('\\t', '\t')
            self.value = self.value.replace('\\"', '\"')
            self.value = self.value.replace("\\'", "\'")
            self.value = self.value.replace('\\\\', '\\')
            tmp = c3dgen.newTmp()
            c3dgen.addAsign(tmp, 'H')
            for ascii in self.value:
                c3dgen.addSetHeap('H', ord(ascii))
                c3dgen.nextHeap()
            c3dgen.addSetHeap('H', '-1')
            c3dgen.nextHeap()
            return ReturnC3D()

def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="{self.value}"];'
    return ReturnAST(dot, id)
```

Relacional

La clase Relational representa operaciones relacionales (como igualdad, desigualdad, mayor que, menor que, etc.) en un lenguaje de programación. Está diseñada para manejar expresiones que involucran comparaciones entre dos operandos (exp1 y exp2) con un operador relacional específico (sign). La clase hereda de la clase Expression y contribuye a la construcción del árbol sintáctico abstracto (AST) para representar estas operaciones.

En el método execute, se evalúa la operación relacional según el operador (sign). Por ejemplo, se implementan funciones como equal para la igualdad, notEqual para la desigualdad, greatEqual para mayor o igual, lessEqual para menor o igual, great para mayor que, y less para menor que. La evaluación se realiza considerando los tipos de los operandos y generando un objeto ReturnType que contiene el resultado y el tipo resultante de la operación relacional.

El método compile está presente pero no está implementado, indicando que aún no se ha desarrollado la lógica para la generación de código de tres direcciones (C3D) relacionada con estas operaciones.

El método ast contribuye a la creación del AST, generando un nodo en el árbol con una etiqueta que representa el operador relacional (sign). Además, se agregan nodos para las expresiones (exp1 y exp2) conectados al nodo principal.

```python
from statements.Objects.Table import Field
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type
from utils.TypeExp import TypeExp
from statements.Env.AST import AST, ReturnAST
from statements.Abstracts.Expression import Expression
from statements.Env.Env import Env

class Relational(Expression):
    def __init__(self, line: int, column: int, exp1: Expression, sign: str, exp2: Expression):
        super().__init__(line, column, TypeExp.RELATIONAL_OP)
        self.exp1 = exp1
        self.sign = sign
        self.exp2 = exp2

    def setField(self, field: dict[str, Field]) -> any:
        self.exp1.setField(field)
        self.exp2.setField(field)
```

```python
def execute(self, env: Env) -> ReturnType:
    match self.sign:
        case '==':
            return self.equal(env)
        case '=':
            return self.equal(env)
        case '!=':
            return self.notEqual(env)
        case '>=':
            return self.greatEqual(env)
        case '<=':
            return self.lessEqual(env)
        case '>':
            return self.great(env)
        case '<':
            return self.less(env)
        case _:
            return ReturnType('NULL', Type.NULL)

def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
    pass

def equal (self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    if value1.type in [Type.INT, Type.DECIMAL]:
        if value2.type in [Type.INT, Type.DECIMAL]:
            return ReturnType(value1.value == value2.value, Type.BOOLEAN)
        env.setError("Los tipos no son válidos para operaciones relacionales (==)", self.exp2.line, self.exp2.column)
        return ReturnType('NULL', Type.NULL)
    if value1.type in [Type.NVARCHAR, Type.NCHAR] and value2.type in [Type.NVARCHAR, Type.NCHAR]:
        return ReturnType(value1.value == value2.value, Type.BOOLEAN)
    env.setError("Los tipos no son válidos para operaciones relacionales (==)", self.exp2.line, self.exp2.column)
    return ReturnType('NULL', Type.NULL)
```

```python
def notEqual(self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    if value1.type == Type.INT or value1.type == Type.DECIMAL:
        if value2.type == Type.INT or value2.type == Type.DECIMAL:
            return ReturnType(value1.value != value2.value, Type.BOOLEAN)
        env.setError("Los tipos no son válidos para operaciones relacionales (!=)", self.exp2.line, self.exp2.column)
        return ReturnType('NULL', Type.NULL)
    if value1.type in [Type.NVARCHAR, Type.NCHAR] and value2.type in [Type.NVARCHAR, Type.NCHAR]:
        return ReturnType(value1.value != value2.value, Type.BOOLEAN)
    env.setError("Los tipos no son válidos para operaciones relacionales (!=)", self.exp2.line, self.exp2.column)
    return ReturnType('NULL', Type.NULL)

def greatEqual(self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    if value1.type in [Type.INT, Type.DECIMAL]:
        if value2.type in [Type.INT, Type.DECIMAL]:
            return ReturnType(value1.value >= value2.value, Type.BOOLEAN)
        env.setError("Los tipos no son válidos para operaciones relacionales (>=)", self.exp2.line, self.exp2.column)
        return ReturnType('NULL', Type.NULL)
    if value1.type in [Type.NVARCHAR, Type.NCHAR] and value2.type in [Type.NVARCHAR, Type.NCHAR]:
        return ReturnType(value1.value >= value2.value, Type.BOOLEAN)
    env.setError("Los tipos no son válidos para operaciones relacionales (>=)", self.exp2.line, self.exp2.column)
    return ReturnType('NULL', Type.NULL)
```

```python
def lessEqual(self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    if value1.type in [Type.INT, Type.DECIMAL]:
        if value2.type in [Type.INT, Type.DECIMAL]:
            return ReturnType(value1.value <= value2.value, Type.BOOLEAN)
        env.setError("Los tipos no son válidos para operaciones relacionales (<=)", self.exp2.line, self.exp2.column)
        return ReturnType('NULL', Type.NULL)
    if value1.type in [Type.NVARCHAR, Type.NCHAR] and value2.type in [Type.NVARCHAR, Type.NCHAR]:
        return ReturnType(value1.value <= value2.value, Type.BOOLEAN)
    env.setError("Los tipos no son válidos para operaciones relacionales (<=)", self.exp2.line, self.exp2.column)
    return ReturnType('NULL', Type.NULL)

def great(self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    if value1.type in [Type.INT, Type.DECIMAL]:
        if value2.type in [Type.INT, Type.DECIMAL]:
            return ReturnType(value1.value > value2.value, Type.BOOLEAN)
        env.setError("Los tipos no son válidos para operaciones relacionales (>)", self.exp2.line, self.exp2.column)
        return ReturnType('NULL', Type.NULL)
    if value1.type in [Type.NVARCHAR, Type.NCHAR] and value2.type in [Type.NVARCHAR, Type.NCHAR]:
        return ReturnType(value1.value > value2.value, Type.BOOLEAN)
    env.setError("Los tipos no son válidos para operaciones relacionales (>)", self.exp2.line, self.exp2.column)
    return ReturnType('NULL', Type.NULL)
```

```python
def less(self, env: Env) -> ReturnType:
    value1: ReturnType = self.exp1.execute(env)
    value2: ReturnType = self.exp2.execute(env)
    if value1.type in [Type.INT, Type.DECIMAL]:
        if value2.type in [Type.INT, Type.DECIMAL]:
            return ReturnType(value1.value < value2.value, Type.BOOLEAN)
        env.setError("Los tipos no son válidos para operaciones relacionales (<)", self.exp2.line, self.exp2.column)
        return ReturnType('NULL', Type.NULL)
    if value1.type in [Type.NVARCHAR, Type.NCHAR] and value2.type in [Type.NVARCHAR, Type.NCHAR]:
        return ReturnType(value1.value < value2.value, Type.BOOLEAN)
    env.setError("Los tipos no son válidos para operaciones relacionales (<)", self.exp2.line, self.exp2.column)
    return ReturnType('NULL', Type.NULL)

def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="{self.sign}"];'
    value1: ReturnAST = self.exp1.ast(ast)
    dot += '\n' + value1.dot
    dot += f'\nnode_{id} -> node_{value1.id};'
    value2: ReturnAST = self.exp2.ast(ast)
    dot += '\n' + value2.dot
    dot += f'\nnode_{id} -> node_{value2.id};'
    return ReturnAST(dot, id)
```

Return

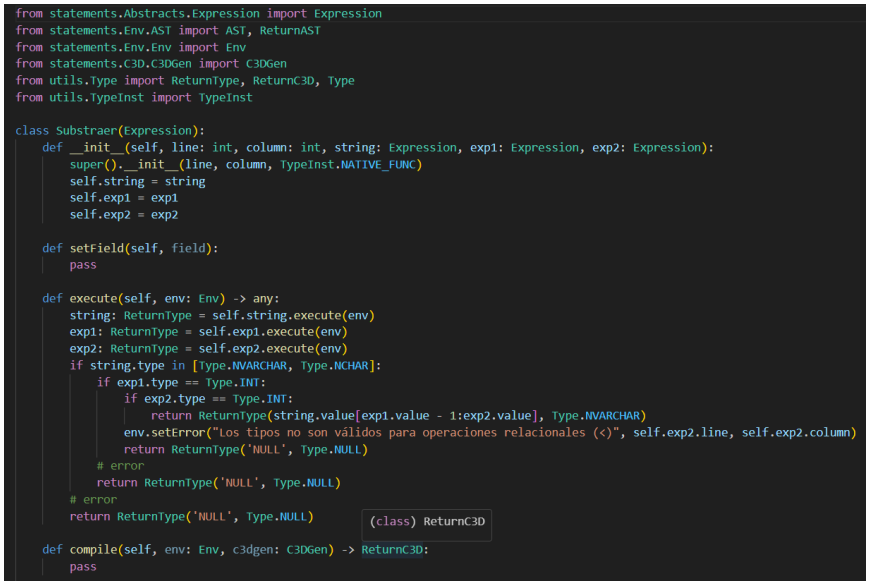
La clase Return representa una instrucción de retorno en el lenguaje de programación, utilizada para devolver un valor de una función. Su constructor recibe la línea y columna de la declaración, junto con una expresión que representa el valor a devolver. El método execute ejecuta la expresión asociada y devuelve un objeto ReturnType con el valor y tipo resultantes, o un objeto con tipo TypeExp.RETURN y un valor nulo si no hay expresión. Además, la clase contribuye a la construcción del árbol sintáctico abstracto (AST) mediante el método ast, generando un nodo etiquetado como "RETURN" y conectando un nodo de expresión si está presente. Sin embargo, la lógica de compilación para generar código de tres direcciones (C3D) aún no ha sido implementada en el método compile.

```python
from statements.Objects.Table import Field
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type
from utils.TypeExp import TypeExp
from statements.Env.AST import AST, ReturnAST
from statements.Abstracts.Expression import Expression
from statements.Env.Env import Env

class Return(Expression):
    def __init__(self, line: int, column: int, exp: Expression):
        super().__init__(line, column, TypeExp.RETURN)
        self.exp = exp

    def setField(self, _: dict[str, Field]) -> any:
        pass

    def execute(self, env: Env) -> ReturnType:
        if self.exp:
            value: ReturnType = self.exp.execute(env)
            return ReturnType(value.value, value.type)
        return ReturnType(self.typeExp, Type.NULL)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="RETURN"];'
        if self.exp:
            value1: ReturnAST = self.exp.ast(ast)
            dot += '\n' + value1.dot
            dot += f'\nnode_{id} -> node_{value1.id};'
        return ReturnAST(dot, id)
```

Subtraer

La clase Substraer representa una operación de subcadena en un lenguaje de programación. Su constructor toma la línea y la columna de la declaración, junto con tres expresiones que representan la cadena original, el índice de inicio y el índice de fin de la subcadena. El método execute evalúa estas expresiones en el entorno proporcionado y devuelve una subcadena de la cadena original según los índices especificados. La clase contribuye al árbol sintáctico abstracto (AST) mediante el método ast, generando un nodo etiquetado como "SUBSTRAER" y conectando nodos para cada una de las expresiones involucradas.

Cabe destacar que la lógica de compilación para generar código de tres direcciones (C3D) aún no ha sido implementada en el método compile. Además, la clase maneja ciertos casos de error, como tipos inválidos para la operación de subcadena, devolviendo un objeto ReturnType con tipo Type.NULL en caso de error.



```python
from statements.Abstracts.Expression import Expression
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type
from utils.TypeInst import TypeInst

class Substraer(Expression):
    def __init__(self, line: int, column: int, string: Expression, exp1: Expression, exp2: Expression):
        super().__init__(line, column, TypeInst.NATIVE_FUNC)
        self.string = string
        self.exp1 = exp1
        self.exp2 = exp2

    def setField(self, field):
        pass

    def execute(self, env: Env) -> any:
        string: ReturnType = self.string.execute(env)
        exp1: ReturnType = self.exp1.execute(env)
        exp2: ReturnType = self.exp2.execute(env)
        if string.type in [Type.NVARCHAR, Type.NCHAR]:
            if exp1.type == Type.INT:
                if exp2.type == Type.INT:
                    return ReturnType(string.value[exp1.value - 1:exp2.value], Type.NVARCHAR)
                env.setError("Los tipos no son válidos para operaciones relacionales (<)", self.exp2.line, self.exp2.column)
                return ReturnType('NULL', Type.NULL)
            # error
            return ReturnType('NULL', Type.NULL)
        # error
        return ReturnType('NULL', Type.NULL)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass
```

(class) ReturnC3D

```python
def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="SUBSTRAER"];'
    string: ReturnAST = self.string.ast(ast)
    dot += '\n' + string.dot
    value1: ReturnAST = self.exp1.ast(ast)
    dot += '\n' + value1.dot
    value2: ReturnAST = self.exp2.ast(ast)
    dot += '\n' + value2.dot
    dot += f'\nnode_{id} -> node_{string.id};'
    dot += f'\nnode_{id} -> node_{value1.id};'
    dot += f'\nnode_{id} -> node_{value2.id};'
    return ReturnAST(dot, id)
```

Las Instrucciones que el programa utilizara son las siguientes:

AlterTable

La clase AlterTable representa una instrucción de alteración de tabla en un lenguaje de programación. Su constructor toma la línea y la columna de la declaración, junto con información como el identificador de la tabla, la acción a realizar (agregar, eliminar, renombrar), y campos adicionales según la acción. El método execute realiza la acción correspondiente sobre el entorno proporcionado, como agregar o eliminar una columna. La clase contribuye al árbol sintáctico abstracto (AST) mediante el método ast, generando nodos etiquetados según la acción de alteración de tabla.

La lógica de compilación para generar código de tres direcciones (C3D) aún no ha sido implementada en el método compile. Además, se proporciona un método estático getType que devuelve una cadena representando el tipo de dato en función del tipo proporcionado, lo cual podría ser utilizado en la generación del AST.

Cabe destacar que el manejo de errores en caso de acciones no reconocidas o faltantes no está implementado en el método execute.

```python
from statements.Abstracts.Instruction import Instruction
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnC3D, Type
from utils.TypeInst import TypeInst

class AlterTable(Instruction):
    def __init__(self, line: int, column: int, id: str, action: str, field1: str, field2: str, type: Type):
        super().__init__(line, column, TypeInst.ALTER_TABLE)
        self.id = id
        self.action = action
        self.field1 = field1
        self.field2 = field2
        self.type = type

    def execute(self, env: Env) -> any:
        if self.action.lower() == 'add':
            env.addColumn(self.id, self.field1, self.type, self.line, self.column)
            return
        if self.action.lower() == 'drop':
            env.dropColumn(self.id, self.field1, self.line, self.column)
            return
        if self.action.lower() == 'renameto':
            env.renameTo(self.id, self.field1, self.line, self.column)
        if self.action.lower() == 'renamecolumn':
            env.renameColumn(self.id, self.field1, self.field2, self.line, self.column)
```

```python
    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="ALTER TABLE"];'
        match self.action.lower():
            case 'add':
                dot += f'node_{id}_action[label="ADD"];'
                dot += f'\nnode_{id}_table[label="{self.id}"];'
                dot += f'\nnode_{id}_field1[label="{self.field1}"];'
                dot += f'\nnode_{id}_type[label="{self.getType(self.type)}"];'
                dot += f'\nnode_{id}_action -> node_{id}_table;'
                dot += f'\nnode_{id}_action -> node_{id}_field1;'
                dot += f'\nnode_{id}_action -> node_{id}_type;'
            case 'drop':
                dot += f'node_{id}_action[label="DROP"];'
                dot += f'\nnode_{id}_table[label="{self.id}"];'
                dot += f'\nnode_{id}_field1[label="{self.field1}"];'
                dot += f'\nnode_{id}_action -> node_{id}_table;'
                dot += f'\nnode_{id}_action -> node_{id}_field1;'
            case 'renameto':
                dot += f'node_{id}_action[label="RENAME TO"];'
                dot += f'\nnode_{id}_table[label="{self.id}"];'
                dot += f'\nnode_{id}_field1[label="{self.field1}"];'
                dot += f'\nnode_{id}_action -> node_{id}_table;'
                dot += f'\nnode_{id}_action -> node_{id}_field1;'
            case 'renamecolumn':
                dot += f'node_{id}_action[label="RENAME COLUMN"];'
                dot += f'\nnode_{id}_table[label="{self.id}"];'
                dot += f'\nnode_{id}_field1[label="{self.field1}"];'
                dot += f'\nnode_{id}_field2[label="{self.field2}"];'
                dot += f'\nnode_{id}_action -> node_{id}_table;'
                dot += f'\nnode_{id}_action -> node_{id}_field1;'
                dot += f'\nnode_{id}_action -> node_{id}_field2;'
        dot += f'\nnode_{id} -> node_{id}_action;'
        return ReturnAST(dot, id)
```

```python
def getType(type: Type) -> str:
    match type:
        case Type.INT:
            return "INT"
        case Type.DOUBLE:
            return "DOUBLE"
        case Type.VARCHAR:
            return "VARCHAR"
        case Type.BOOLEAN:
            return "BOOLEAN"
        case Type.DATE:
            return "DATE"
        case Type.TABLE:
            return "TABLE"
        case _:
            return "NULL"
```

AsignID

La clase AsignID representa una instrucción de asignación a un identificador en un lenguaje de programación. El constructor toma la línea y la columna de la declaración, así como el identificador (id) y la expresión (value) que se asignará a ese identificador. El método execute realiza la asignación en el entorno proporcionado, reasignando el valor asociado al identificador.

En cuanto a la contribución al árbol sintáctico abstracto (AST), el método ast genera nodos etiquetados, representando la instrucción de asignación y el identificador involucrado. El valor asignado se incorpora al AST utilizando el AST de la expresión (value). La lógica de compilación para generar código de tres direcciones (C3D) aún no ha sido implementada en el método compile.

Cabe señalar que la implementación actual del método compile está pendiente, y se requeriría para completar la funcionalidad de generación de código C3D basado en esta instrucción.

```python
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.Abstracts.Instruction import Instruction
from statements.Abstracts.Expression import Expression
from utils.TypeInst import TypeInst
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnC3D, Type

class AsignID(Instruction):
    def __init__(self, line: int, column: int, id: str, value: Expression):
        super().__init__(line, column, TypeInst.ASIGN_ID)
        self.id = id
        self.value = value

    def execute(self, env: Env):
        value = self.value.execute(env)
        env.reasignID(self.id, value, self.line, self.column)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="SET"];'
        value1: ReturnAST = self.value.ast(ast)
        dot += f'\nnode_{id}_id[label="{self.id}"]'
        dot += f'\nnode_{id} -> node_{id}_id'
        dot += '\n' + value1.dot
        dot += f'\nnode_{id} -> node_{value1.id};'
        return ReturnAST(dot, id)
```

Block

La clase Block representa un bloque de instrucciones en un lenguaje de programación. Un bloque es un conjunto de instrucciones que se ejecutan en secuencia. En el constructor, se especifica la línea y la columna de la declaración, así como una lista de instrucciones (instructions) que forman parte del bloque.

El método execute crea un nuevo entorno (newEnv) basado en el entorno actual y ejecuta cada instrucción en el bloque en este nuevo entorno. Se utiliza un manejo básico de excepciones para continuar con la ejecución si una instrucción en el bloque produce un error.

En cuanto al método ast, este contribuye al árbol sintáctico abstracto (AST) representando el bloque como un nodo etiquetado con "BEGIN-END". Luego, agrega nodos para cada instrucción en el bloque al AST.

Cabe mencionar que la implementación actual del método compile está pendiente y sería necesario completarlo para generar código de tres direcciones (C3D) basado en las instrucciones del bloque.

```python
from statements.Abstracts.Instruction import Instruction
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from utils.TypeInst import TypeInst
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnC3D, Type

class Block(Instruction):
    def __init__(self, line: int, column: int, instructions: list[Instruction]):
        super().__init__(line, column, TypeInst.BLOCK_INST)
        self.instructions = instructions

    def execute(self, env: Env) -> any:
        newEnv: Env = Env(env, env.name)
        for instruction in self.instructions:
            try:
                ret = instruction.execute(newEnv)
                if ret:
                    return ret
            except: {}

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="BEGIN-END"];'
        value1: ReturnAST
        for i in range(len(self.instructions)):
            value1 = self.instructions[i].ast(ast)
            dot += '\n' + value1.dot
            dot += f'\nnode_{id} -> node_{value1.id};'
        return ReturnAST(dot, id)
```

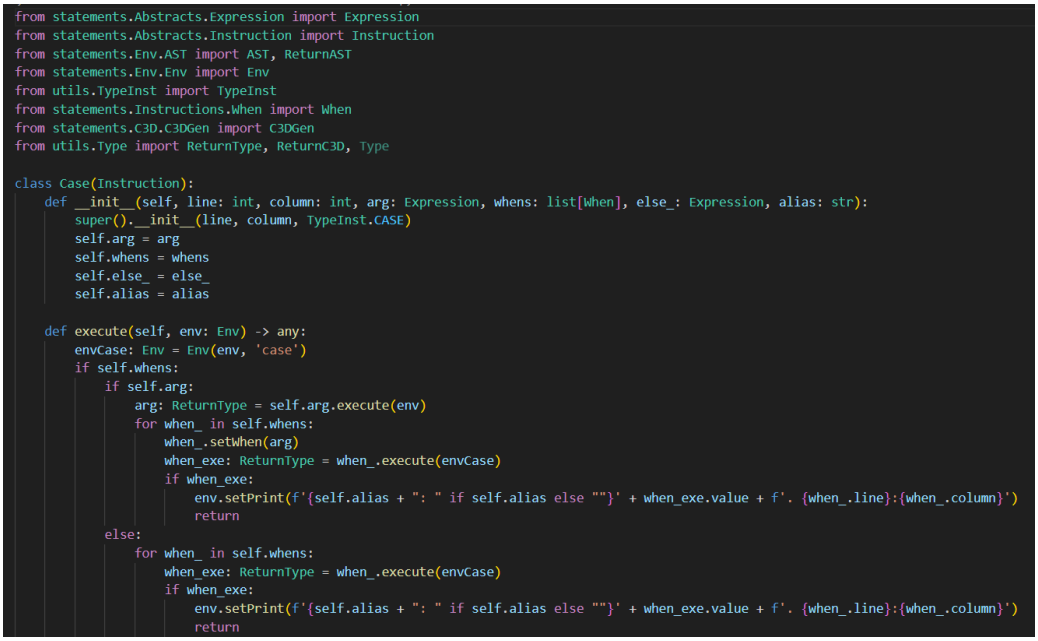
Case

La clase Case representa una estructura de control de casos (switch-case) en un lenguaje de programación. En el constructor, se especifica la línea y la columna de la declaración, así como la expresión de control (arg), una lista de casos (whens), una expresión por defecto (else_), y un alias opcional para el caso (alias).

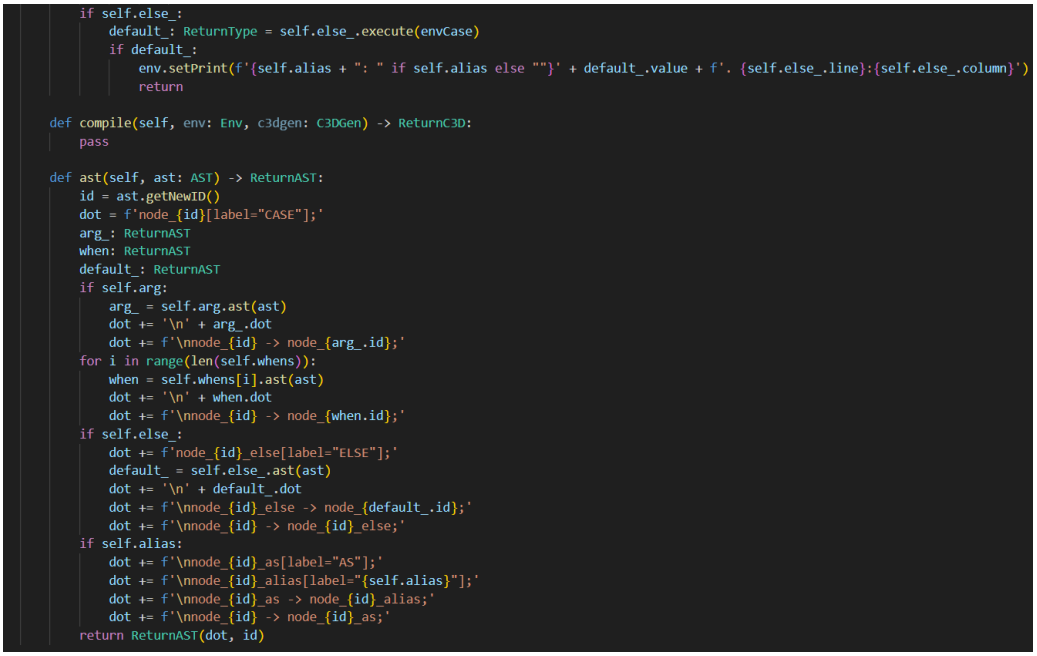
El método execute evalúa la expresión de control y ejecuta la rama correspondiente en función de los casos o la expresión por defecto. Cada caso está representado por la clase When. Se utiliza un entorno (envCase) para manejar las variables locales en el ámbito del caso.

El método ast contribuye al árbol sintáctico abstracto (AST) representando el caso como un nodo etiquetado con "CASE". Se añaden nodos para la expresión de control, cada caso y, opcionalmente, la expresión por defecto y el alias.

La implementación actual del método compile está pendiente y sería necesario completarlo para generar código de tres direcciones (C3D) basado en la evaluación del caso.



```python
from statements.Abstracts.Expression import Expression
from statements.Abstracts.Instruction import Instruction
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from utils.TypeInst import TypeInst
from statements.Instructions.When import When
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class Case(Instruction):
    def __init__(self, line: int, column: int, arg: Expression, whens: list[When], else_: Expression, alias: str):
        super().__init__(line, column, TypeInst.CASE)
        self.arg = arg
        self.whens = whens
        self.else_ = else_
        self.alias = alias

    def execute(self, env: Env) -> any:
        envCase: Env = Env(env, 'case')
        if self.whens:
            if self.arg:
                arg: ReturnType = self.arg.execute(env)
                for when_ in self.whens:
                    when_.setWhen(arg)
                    when_exe: ReturnType = when_.execute(envCase)
                    if when_exe:
                        env.setPrint(f'{self.alias + ": " if self.alias else ""}' + when_exe.value + f'. {when_.line}:{when_.column}')
                        return
            else:
                for when_ in self.whens:
                    when_exe: ReturnType = when_.execute(envCase)
                    if when_exe:
                        env.setPrint(f'{self.alias + ": " if self.alias else ""}' + when_exe.value + f'. {when_.line}:{when_.column}')
                        return
```



```python
        if self.else_:
            default_: ReturnType = self.else_.execute(envCase)
            if default_:
                env.setPrint(f'{self.alias + ": " if self.alias else ""}' + default_.value + f'. {self.else_.line}:{self.else_.column}')
                return

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="CASE"];'
        arg_: ReturnAST
        when: ReturnAST
        default_: ReturnAST
        if self.arg:
            arg_ = self.arg.ast(ast)
            dot += '\n' + arg_.dot
            dot += f'\nnode_{id} -> node_{arg_.id};'
        for i in range(len(self.whens)):
            when = self.whens[i].ast(ast)
            dot += '\n' + when.dot
            dot += f'\nnode_{id} -> node_{when.id};'
        if self.else_:
            dot += f'node_{id}_else[label="ELSE"];'
            default_ = self.else_.ast(ast)
            dot += '\n' + default_.dot
            dot += f'\nnode_{id}_else -> node_{default_.id};'
            dot += f'\nnode_{id} -> node_{id}_else;'
        if self.alias:
            dot += f'\nnode_{id}_as[label="AS"];'
            dot += f'\nnode_{id}_alias[label="{self.alias}"];'
            dot += f'\nnode_{id}_as -> node_{id}_alias;'
            dot += f'\nnode_{id} -> node_{id}_as;'
        return ReturnAST(dot, id)
```

CreateTable

La clase CreateTable representa la instrucción de creación de una tabla en un entorno de programación. El constructor toma la línea y la columna de la declaración, el nombre de la tabla (name), y una lista de atributos y claves foráneas (attribs). Los atributos pueden ser de tipo Attribute o ForeignKey.

El método execute crea una nueva instancia de la tabla con los atributos proporcionados y la guarda en el entorno. Luego, realiza llamadas a funciones relacionadas con la manipulación del árbol XML de la base de datos para reflejar la creación de la tabla y sus columnas. En caso de errores, imprime mensajes en la consola.

El método ast contribuye al árbol sintáctico abstracto (AST), representando la creación de la tabla como un nodo etiquetado con "TABLE". Se añaden nodos para el nombre de la tabla y los campos.

La implementación actual del método compile está pendiente y sería necesario completarlo para generar código de tres direcciones (C3D) basado en la creación de la tabla.

La función auxiliar getTypeOf devuelve el tipo de dato correspondiente a un tipo proporcionado, mapeando los tipos internos a sus equivalentes en la base de datos.

```python
from statements.Abstracts.Instruction import Instruction
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.Objects.Table import Table
from utils.TypeInst import TypeInst
from utils.Attribute import Attribute
from utils.ForeignKey import ForeignKey
from utils.Global import *
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnC3D, Type

class CreateTable(Instruction):
    def __init__(self, line: int, column: int, name: str, attribs: list[Attribute | ForeignKey]):
        super().__init__(line, column, TypeInst.CREATE_TABLE)
        self.name = name
        self.attribs = attribs

    def execute(self, env: Env) -> any:
        table = Table(self.name.lower(), self.attribs)
        env.saveTable(self.name, table, self.line, self.column)

        #-----------------XML-----------------
        xml.createTable(getUsedDatabase(), self.name.lower())
        for attrib in self.attribs:
            if type(attrib) == Attribute:
                xml.createColumn(getUsedDatabase(), self.name.lower(), attrib.id, self.getTypeOf(attrib.type).lower(), attrib.length, attrib.props['
```

```python
def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
    pass

def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="TABLE"];'
    dot += f'\nnode_{id}_name[label="{self.name}"]'
    dot += f'\nnode_{id}_fields[label="CAMPOS"]'
    for i in range(len(self.attribs)):
        if type(self.attribs[i]) == Attribute:
            dot += f'\nnode_{id}_field_{i}[label={self.attribs[i].id}]'
            dot += f'\nnode_{id}_fields -> node_{id}_field_{i};'
        elif type(self.attribs[i]) == ForeignKey:
            pass
    dot += f'\nnode_{id} -> node_{id}_name;'
    dot += f'\nnode_{id} -> node_{id}_fields;'
    return ReturnAST(dot, id)

def getTypeOf(self, type: Type) -> str:
    match type:
        case Type.INT:
            return "INT"
        case Type.DECIMAL:
            return "DECIMAL"
        case Type.NCHAR:
            return "NCHAR"
        case Type.NVARCHAR:
            return "NVARCHAR"
        case Type.BIT:
            return "BIT"
        case Type.DATE:
            return "DATE"
        case Type.TABLE:
            return "TABLE"
        case _:
            return "NULL"
```

DeleteTable

La clase DeleteTable representa la instrucción de eliminación de una tabla en un entorno de programación. El constructor toma la línea y la columna de la declaración, el nombre de la tabla a eliminar (id), y una condición opcional (condition) que especifica las filas a eliminar.

El método execute realiza la eliminación de la tabla invocando la función deleteTable del entorno, proporcionándole el nombre de la tabla y la condición de eliminación.

El método ast contribuye al árbol sintáctico abstracto (AST), representando la eliminación de la tabla como un nodo etiquetado con "DELETE". Se añade un nodo para el nombre de la tabla y otro para la condición, si está presente.

La implementación actual del método compile está pendiente y sería necesario completarlo para generar código de tres direcciones (C3D) basado en la eliminación de la tabla.

```python
from statements.Abstracts.Expression import Expression
from statements.Abstracts.Instruction import Instruction
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from utils.TypeInst import TypeInst
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnC3D, Type

class DeleteTable(Instruction):
    def __init__(self, line: int, column: int, id: str, condition: Expression):
        super().__init__(line, column, TypeInst.DELETE_TABLE)
        self.id = id
        self.condition = condition

    def execute(self, env: Env) -> any:
        if self.condition:
            env.deleteTable(self.id, self.condition, self.line, self.column)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="DELETE"];'
        dot += f'\nnode_{id}_tableName[label="{self.id}"];'
        dot += f'\nnode_{id} -> node_{id}_tableName;'
        condition = self.condition.ast(ast)
        dot += f'\n{condition.dot}'
        dot += f'\nnode_{id} -> node_{condition.id};'
        return ReturnAST(dot, id)
```

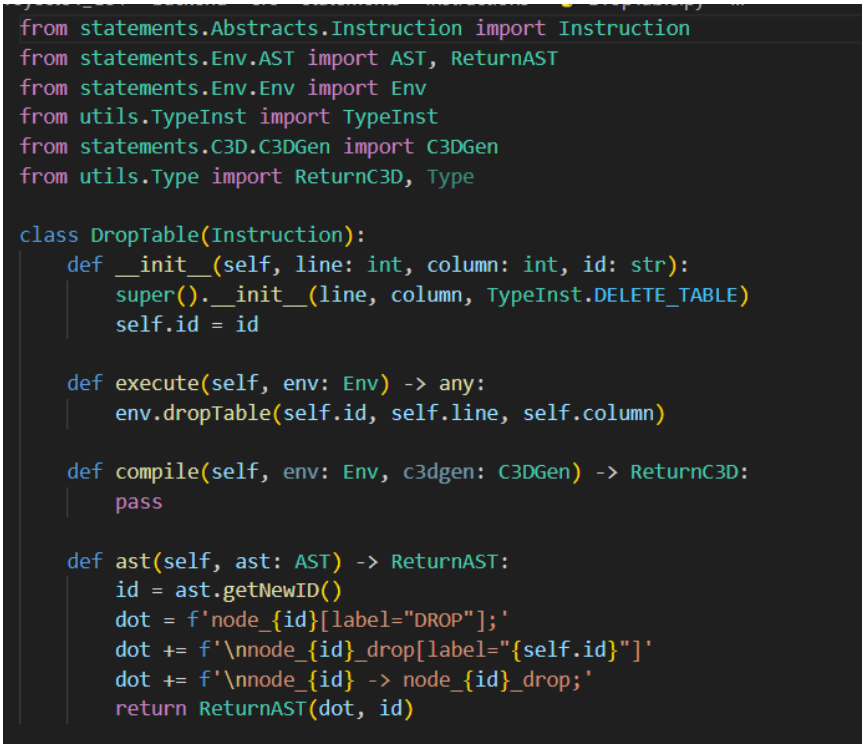
DropTable

La clase DropTable representa una instrucción para eliminar una tabla en un lenguaje de programación o lenguaje específico del dominio. Al heredar de la clase Instruction, se espera que esta instrucción pueda ejecutarse y generar representaciones en el árbol de sintaxis abstracta (AST) y en código intermedio C3D. La instancia de la clase se inicializa con información sobre la línea y columna del código fuente donde se encuentra la instrucción, así como con el identificador de la tabla (id) que se va a eliminar.

El método execute de la clase se encarga de ejecutar la instrucción, invocando el método dropTable del entorno (env). Este método toma como argumento el identificador de la tabla y la ubicación en el código fuente donde se realiza la operación de eliminación.

La función ast genera una representación del nodo en el AST para esta instrucción de eliminación de tabla. Crea un nodo con la etiqueta "DROP" y un hijo etiquetado con el identificador de la tabla (self.id). Este AST se utiliza para estructurar y visualizar la jerarquía sintáctica del código fuente.

Cabe destacar que el método compile no tiene implementación, lo que sugiere que la generación de código intermedio C3D para esta instrucción aún no se ha desarrollado o no es necesaria en el contexto actual.



```python
from statements.Abstracts.Instruction import Instruction
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from utils.TypeInst import TypeInst
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnC3D, Type

class DropTable(Instruction):
    def __init__(self, line: int, column: int, id: str):
        super().__init__(line, column, TypeInst.DELETE_TABLE)
        self.id = id

    def execute(self, env: Env) -> any:
        env.dropTable(self.id, self.line, self.column)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="DROP"];'
        dot += f'\nnode_{id}_drop[label="{self.id}"]'
        dot += f'\nnode_{id} -> node_{id}_drop;'
        return ReturnAST(dot, id)
```

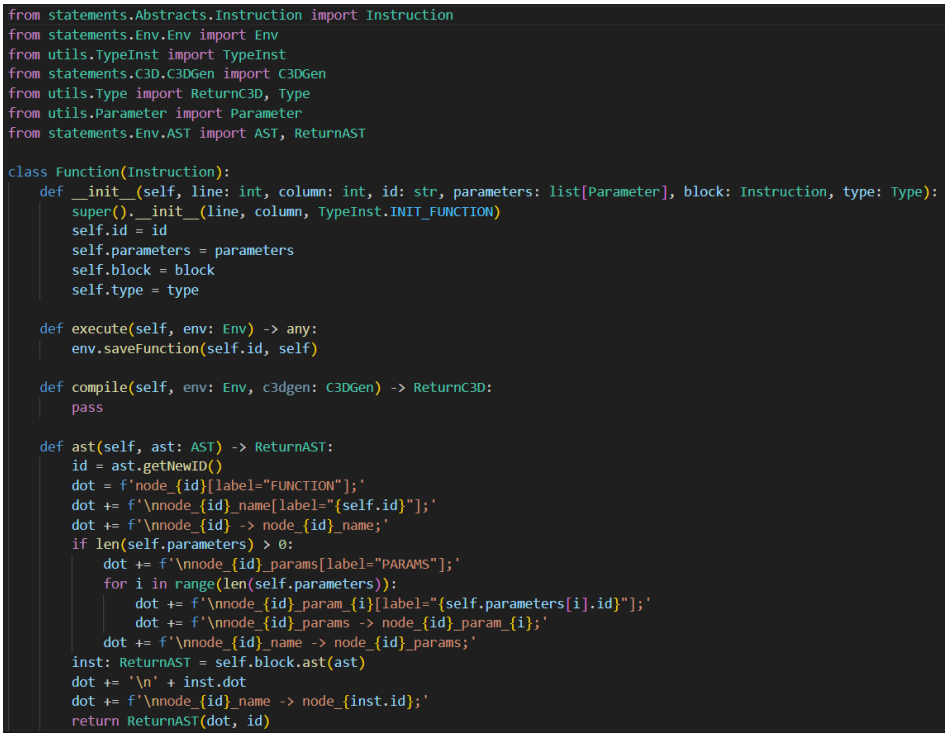
Function

La clase Function representa la definición de una función en un lenguaje de programación o un lenguaje específico del dominio. Al heredar de la clase Instruction, se espera que esta definición de función pueda ejecutarse y generar representaciones en el árbol de sintaxis abstracta (AST) y en código intermedio C3D. La instancia de la clase se inicializa con información sobre la línea y columna del código fuente donde se encuentra la definición, el identificador de la función (id), una lista de parámetros (parameters), un bloque de instrucciones (block) y el tipo de retorno de la función (type).

El método execute de la clase se encarga de ejecutar la instrucción, invocando el método saveFunction del entorno (env). Este método toma como argumento el identificador de la función y la propia instancia de la clase Function, lo que permite almacenar la definición de la función en el entorno para su posterior uso.

La función ast genera una representación del nodo en el AST para esta definición de función. Crea un nodo con la etiqueta "FUNCTION" y un hijo etiquetado con el identificador de la función (self.id). Si la función tiene parámetros, se crean nodos adicionales para representar la sección de parámetros. Luego, se genera la representación AST del bloque de instrucciones utilizando el método ast del bloque (self.block) y se conecta como un hijo del nodo de la función.

La función compile no tiene implementación, lo que sugiere que la generación de código intermedio C3D para esta definición de función aún no se ha desarrollado o no es necesaria en el contexto actual.



```python
from statements.Abstracts.Instruction import Instruction
from statements.Env.Env import Env
from utils.TypeInst import TypeInst
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnC3D, Type
from utils.Parameter import Parameter
from statements.Env.AST import AST, ReturnAST

class Function(Instruction):
    def __init__(self, line: int, column: int, id: str, parameters: list[Parameter], block: Instruction, type: Type):
        super().__init__(line, column, TypeInst.INIT_FUNCTION)
        self.id = id
        self.parameters = parameters
        self.block = block
        self.type = type

    def execute(self, env: Env) -> any:
        env.saveFunction(self.id, self)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="FUNCTION"];'
        dot += f'\nnode_{id}_name[label="{self.id}"];'
        dot += f'\nnode_{id} -> node_{id}_name;'
        if len(self.parameters) > 0:
            dot += f'\nnode_{id}_params[label="PARAMS"];'
            for i in range(len(self.parameters)):
                dot += f'\nnode_{id}_param_{i}[label="{self.parameters[i].id}"];'
                dot += f'\nnode_{id}_params -> node_{id}_param_{i};'
            dot += f'\nnode_{id}_name -> node_{id}_params;'
        inst: ReturnAST = self.block.ast(ast)
        dot += '\n' + inst.dot
        dot += f'\nnode_{id}_name -> node_{inst.id};'
        return ReturnAST(dot, id)
```

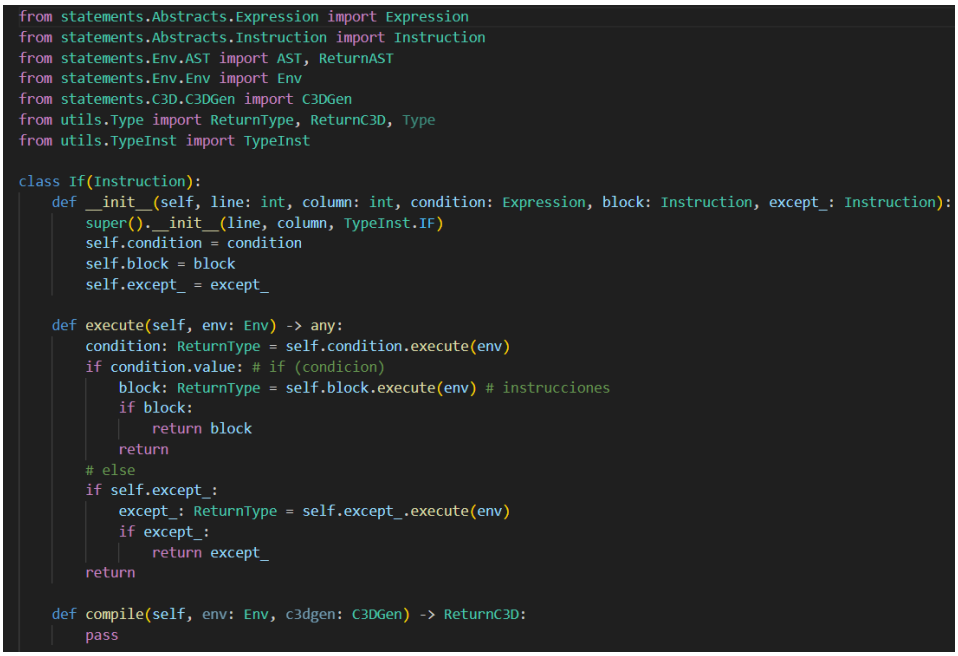
If

La clase If representa una estructura condicional "if" en un lenguaje de programación. La instancia de la clase se inicializa con información sobre la línea y columna del código fuente donde se encuentra la estructura, la condición (condition) que se evalúa para determinar la ejecución de un bloque de instrucciones (block) o un bloque alternativo (except_).

El método execute evalúa la condición y ejecuta el bloque de instrucciones si la condición es verdadera. Si no es verdadera y hay un bloque alternativo, ejecuta ese bloque. El método devuelve el resultado de la ejecución del bloque correspondiente o None si no se ejecuta ninguno.

La función ast genera una representación del nodo en el Árbol de Sintaxis Abstracta (AST) para esta estructura condicional "if". Crea un nodo con la etiqueta "IF" y agrega nodos adicionales para representar la condición (self.condition), el bloque de instrucciones (self.block), y el bloque alternativo (self.except_). Se conecta apropiadamente con arcos dirigidos.

La función compile no tiene implementación, lo que sugiere que la generación de código intermedio C3D para esta estructura condicional "if" aún no se ha desarrollado o no es necesaria en el contexto actual.



```python
from statements.Abstracts.Expression import Expression
from statements.Abstracts.Instruction import Instruction
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type
from utils.TypeInst import TypeInst

class If(Instruction):
    def __init__(self, line: int, column: int, condition: Expression, block: Instruction, except_: Instruction):
        super().__init__(line, column, TypeInst.IF)
        self.condition = condition
        self.block = block
        self.except_ = except_

    def execute(self, env: Env) -> any:
        condition: ReturnType = self.condition.execute(env)
        if condition.value: # if (condicion)
            block: ReturnType = self.block.execute(env) # instrucciones
            if block:
                return block
            return
        # else
        if self.except_:
            except_: ReturnType = self.except_.execute(env)
            if except_:
                return except_
        return

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass
```

InitID

La clase InitID representa la declaración e inicialización de variables o identificadores en un programa. La instancia de la clase se inicializa con información sobre la línea y la columna en el código fuente donde se encuentra la declaración, el nombre del identificador (id), el tipo de datos del identificador (type), y opcionalmente un valor inicial (value). La declaración puede ser de un solo identificador o de una lista de identificadores con sus respectivos tipos.

El método execute se encarga de ejecutar la declaración en el entorno actual. Si la declaración es para un solo identificador (str y Type) y tiene un valor inicial, se evalúa y se guarda en el entorno (env). Si la declaración es para una lista de identificadores (list y list) sin un valor inicial, se guarda cada identificador en el entorno con un valor inicial de 'NULL'. Se manejan errores si los tipos no coinciden.

La función ast genera una representación del nodo en el Árbol de Sintaxis Abstracta (AST) para esta declaración. Se crea un nodo con la etiqueta "DECLARE" y se agregan nodos adicionales para representar el tipo (self.getType(self.type)), el identificador (self.id), y opcionalmente, el valor inicial (self.value). Se conectan los nodos apropiadamente con arcos dirigidos.

La función compile no tiene implementación, lo que sugiere que la generación de código intermedio C3D para esta declaración aún no se ha desarrollado o no es necesaria en el contexto actual. Además, la función getType parece tener un error tipográfico ya que repite el caso Type.DECIMAL. Corregirlo para reflejar correctamente Type.BOOLEAN.

```python
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.Abstracts.Instruction import Instruction
from statements.Abstracts.Expression import Expression
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type
from utils.TypeInst import TypeInst
from typing import Union, List

class InitID(Instruction):
    def __init__(self, line: int, column: int, id: Union[str, List[str]], type: Union[Type, List[Type]], value: Union[Expression, None]):
        super().__init__(line, column, TypeInst.INIT_ID)
        self.id = id
        self.type = type
        self.value = value

    def execute(self, env: Env) -> any:
        if type(self.id) == str and type(self.type) == Type and self.value:
            value: ReturnType = self.value.execute(env)
            if value.type == self.type or self.type == Type.DECIMAL and value.type == Type.INT or \
            self.type == Type.BIT and value.type == Type.INT and int(value.value) in [0, 1] or \
            self.type == Type.NCHAR and value.type == Type.NVARCHAR:
                env.saveID(self.id, value.value, self.type, self.line, self.column)
            else:
                env.setError('Los tipos no coinciden en la declaración', self.line, self.column)
        elif type(self.id) == list and type(self.type) == list and not self.value:
            for i in range(len(self.id)):
                env.saveID(self.id[i], 'NULL', self.type[i], self.line, self.column)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass
```

```python
    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="DECLARE"];'
        if type(self.id) == str and type(self.type) == Type and self.value:
            dot += f'\nnode_{id}_type[label="{self.getType(self.type)}"];'
            dot += f'\nnode_{id} -> node_{id}_type;'
            dot += f'\nnode_{id}_id[label="{self.id}"];'
            dot += f'\nnode_{id}_type -> node_{id}_id;'
            value : ReturnAST = self.value.ast(ast)
            dot += '\n'+value.dot
            dot += f'\nnode_{id}_type -> node_{value.id};'
        elif type(self.id) == list and type(self.type) == list and not self.value:
            for i in range(len(self.id)):
                dot += f'\nnode_{id}_type_{i}[label="{self.getType(self.type[i])}"];'
                dot += f'\nnode_{id} -> node_{id}_type_{i};'
                dot += f'\nnode_{id}_id_{i}[label="${self.id[i]}"];'
                dot += f'\nnode_{id}_type_{i} -> node_{id}_id_{i};'
        return ReturnAST(dot, id)

    def getType(self, type: Type) -> str:
        match type:
            case Type.INT:
                return "INT"
            case Type.DECIMAL:
                return "DECIMAL"
            case Type.NCHAR:
                return "NCHAR"
            case Type.NVARCHAR:
                return "NVARCHAR"
            case Type.DECIMAL:
                return "BOOLEAN"
            case Type.DATE:
                return "DATE"
            case Type.TABLE:
                return "TABLE"
            case _:
                return "NULL"
```

InsertTable

La clase InsertTable representa la instrucción de inserción de datos en una tabla. Se inicializa con información sobre la línea y columna en el código fuente donde se encuentra la instrucción, el nombre de la tabla (name), los campos (fields) y los valores (values) que se desean insertar.

El método execute se encarga de ejecutar la instrucción en el entorno actual. Verifica si la cantidad de campos es igual a la cantidad de valores a insertar. Si son iguales, se llama a la función insertTable en el entorno (env) para realizar la inserción. En caso contrario, se genera un error indicando si hay más o menos valores de los esperados.

La función ast genera una representación del nodo en el Árbol de Sintaxis Abstracta (AST) para esta instrucción de inserción. Se crea un nodo con la etiqueta "INSERT" y se agregan nodos adicionales para representar el nombre de la tabla (self.name), los campos (self.fields), y los valores (self.values). Se conectan los nodos apropiadamente con arcos dirigidos.

La función compile no tiene implementación, lo que sugiere que la generación de código intermedio C3D para esta instrucción aún no se ha desarrollado o no es necesaria en el contexto actual.

```python
from statements.Abstracts.Instruction import Instruction
from statements.Abstracts.Expression import Expression
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from utils.TypeInst import TypeInst
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnC3D, Type

class InsertTable(Instruction):
    def __init__(self, line: int, column: int, name: str, fields: list[str], values: list[Expression]):
        super().__init__(line, column, TypeInst.INSERT_TABLE)
        self.name = name
        self.fields = fields
        self.values = values

    def execute(self, env: Env) -> any:
        if len(self.fields) == len(self.values):
            env.insertTable(self.name, self.fields, self.values, self.line, self.column)
            return
        if len(self.fields) < len(self.values):
            env.setError('Inserta más valores de los esperados', self.line, self.column)
            return
        env.setError('Inserta menos valores de los esperados', self.line, self.column)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass
```

```python
    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="INSERT"];'
        dot += f'\nnode_{id}_table[label="{self.name}"];'
        dot += f'\nnode_{id}_fields[label="FIELDS"];'
        for i in range(len(self.fields)):
            dot += f'\nnode_{id}_field_{i}[label="{self.fields[i]}"];'
            dot += f'\nnode_{id}_fields -> \nnode_{id}_field_{i};'
        dot += f'\nnode_{id}_values[label="VALORES"];'
        value: ReturnAST
        for i in range(len(self.values)):
            value = self.values[i].ast(ast)
            dot += '\n' + value.dot
            dot += f'\nnode_{id}_values -> node_{value.id};'
        dot += f'\nnode_{id}_table -> node_{id}_fields;'
        dot += f'\nnode_{id}_table -> node_{id}_values;'
        dot += f'\nnode_{id} -> node_{id}_table'
        return ReturnAST(dot, id)
```

Select_prt

La clase Select_prt representa una instrucción de impresión (SELECT) en un lenguaje de programación. Esta instrucción se utiliza para imprimir valores en la salida. La instrucción se inicializa con información sobre la línea y columna en el código fuente donde se encuentra, y una lista de expresiones a imprimir (expression). Cada expresión en la lista está representada como una lista donde el primer elemento es la expresión y el segundo elemento (opcional) es un alias o etiqueta para la impresión.

El método execute se encarga de ejecutar la instrucción en el entorno actual. Itera sobre las expresiones, ejecuta cada expresión y realiza la impresión correspondiente utilizando el entorno. Si se proporciona un alias, se utiliza en la impresión; de lo contrario, se imprime el valor directamente.

El método compile genera código intermedio C3D para la instrucción de impresión. Utiliza un generador de código C3D (C3DGen) para añadir comentarios y generar instrucciones C3D según el tipo de las expresiones a imprimir.

La función ast genera una representación del nodo en el Árbol de Sintaxis Abstracta (AST) para esta instrucción de impresión. Crea un nodo con la etiqueta "SELECT" y agrega nodos adicionales para representar cada expresión en la lista. Si se proporciona un alias, se agrega un nodo "AS" y un nodo para el alias correspondiente.

```python
from statements.Abstracts.Instruction import Instruction
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from utils.TypeInst import TypeInst
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class Select_prt(Instruction):
    def __init__(self, line: int, column: int, expression: list[list[any]]):
        super().__init__(line, column, TypeInst.SELECT)
        self.expression = expression

    def execute(self, env: Env) -> any:
        value: ReturnType
        for i in range(len(self.expression)):
            value = self.expression[i][0].execute(env) if self.expression[i] else None
            if value:
                if self.expression[i][1] != '':
                    env.setPrint(self.expression[i][1] + ': ' + str(value.value))
                else:
                    env.setPrint(value.value)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        c3dgen.addComment('---------- Print ----------')
        if len(self.expression) > 0:
            for exp in self.expression:
                value: ReturnC3D = exp[0].compile(env, c3dgen)
                if value.type == Type.INT:
                    c3dgen.addPrintf('d', '(int) ' + value.strValue)
                elif value.type == Type.DECIMAL:
                    c3dgen.addPrintf('f', '(float) ' + value.strValue)
            c3dgen.addPrint("\n")
        c3dgen.addComment("-------- Fin Print --------")
```

```python
    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="SELECT"];'
        value: ReturnAST
        for i in range(len(self.expression)):
            value = self.expression[i][0].ast(ast)
            if self.expression[i][1] != '':
                dot += f'\nnode_{id}_AS{i}[label="AS"];'
                dot += f'\nnode_{id} -> node_{id}_AS{i};'
                dot += f'\n{value.dot}'
                dot += f'\nnode_{id}_AS{i} -> node_{value.id};'
                dot += f'\nnode_{id}_ASTXT{i}[label="{self.expression[i][1]}"];'
                dot += f'\nnode_{id}_AS{i} -> node_{id}_ASTXT{i};'
            else:
                dot += f'\n{value.dot}'
                dot += f'\nnode_{id} -> node_{value.id};'
        return ReturnAST(dot, id)
```

Select

La clase Select representa una instrucción de selección (SELECT) en un lenguaje de programación. Esta instrucción se utiliza para seleccionar datos de una tabla. La instrucción se inicializa con información sobre la línea y columna en el código fuente donde se encuentra, el nombre de la tabla (id), los campos a seleccionar (fields), y una condición opcional (condition).

El método execute se encarga de ejecutar la instrucción de selección en el entorno actual. Si no se proporciona una condición, se utiliza una condición predeterminada (Primitive(self.line, self.column, 'true', Type.BOOLEAN)). Luego, se llama al método selectTable del entorno para realizar la selección de datos.

El método ast genera una representación del nodo en el Árbol de Sintaxis Abstracta (AST) para esta instrucción de selección. Crea un nodo con la etiqueta "SELECT" y agrega nodos adicionales para representar el nombre de la tabla, los campos a seleccionar y la condición. Si se proporciona una condición, se agrega un nodo "CONDITION" y un nodo adicional para representar la condición en el AST.

En resumen, esta clase proporciona funcionalidad para ejecutar instrucciones de selección en el entorno de ejecución y genera representaciones en AST de estas instrucciones.

```python
from statements.Abstracts.Instruction import Instruction
from statements.Abstracts.Expression import Expression
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.Expressions.Primitive import Primitive
from utils.TypeInst import TypeInst
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnC3D, Type

class Select(Instruction):
    def __init__(self, line: int, column: int, id: str, fields: list[list[any]] or str, condition: Expression):
        super().__init__(line, column, TypeInst.SELECT)
        self.id = id
        self.fields = fields
        self.condition = condition

    def execute(self, env: Env) -> any:
        self.condition = self.condition if self.condition else Primitive(self.line, self.column, 'true', Type.BOOLEAN)
        env.selectTable(self.id, self.fields, self.condition, self.line, self.column)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass
```

```python
def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="SELECT"];'
    dot += f'\nnode_{id}_id[label="{self.id}"];'
    dot += f'\nnode_{id} -> node_{id}_id;'
    dot += f'\nnode_{id}_fields[label="FIELDS"];'
    dot += f'\nnode_{id}_id -> node_{id}_fields;'
    dot += f'\nnode_{id}_condition[label="CONDITION"];'
    dot += f'\nnode_{id}_id -> node_{id}_condition;'
    if type(self.fields) == str:
        dot += f'\nnode_{id}_star[label="*"];'
        dot += f'\nnode_{id}_fields -> node_{id}_star;'
    else:
        value: ReturnAST
        for i in range(len(self.fields)):
            value = self.fields[i][0].ast(ast)
            if self.fields[i][1] != '':
                dot += f'\nnode_{id}_AS${i}[label="AS"];'
                dot += f'\nnode_{id}_fields -> node_{id}_AS{i};'
                dot += f'\n{value.dot};'
                dot += f'\nnode_{id}_AS{i} -> node_{value.id};'
                dot += f'\nnode_{id}_ASTXT{i}[label="{self.fields[i][1]}"];'
                dot += f'\nnode_{id}_AS{i} -> node_{id}_ASTXT{i};'
            else:
                dot += f'\n{value.dot}'
                dot += f'\nnode_{id}_fields -> node_{value.id};'
    if self.condition:
        condition = self.condition.ast(ast)
        dot += f'\n{condition.dot}'
        dot += f'\nnode_{id}_condition -> node_{condition.id};'
    return ReturnAST(dot, id)
```

TruncateTable

La clase TruncateTable representa una instrucción de truncado (TRUNCATE) en un lenguaje de programación. Esta instrucción se utiliza para eliminar todos los registros de una tabla, pero mantiene la estructura de la tabla para futuras operaciones. La instrucción se inicializa con información sobre la línea y columna en el código fuente donde se encuentra y el nombre de la tabla a truncar (id).

El método execute se encarga de ejecutar la instrucción de truncado en el entorno actual. Llama al método truncateTable del entorno, pasando el nombre de la tabla y la posición en el código fuente.

El método ast genera una representación del nodo en el Árbol de Sintaxis Abstracta (AST) para esta instrucción de truncado. Crea un nodo con la etiqueta "TRUNCATE" y agrega un nodo adicional para representar el nombre de la tabla en el AST.

```python
from statements.Abstracts.Instruction import Instruction
from utils.TypeInst import TypeInst
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnC3D, Type

class TruncateTable(Instruction):
    def __init__(self, line: int, column: int, id: str):
        super().__init__(line, column, TypeInst.TRUNCATE_TABLE)
        self.id = id

    def execute(self, env: Env) -> any:
        env.truncateTable(self.id, self.line, self.column)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="TRUNCATE"];'
        dot += f'\nnode_{id}_truncate[label="{self.id}"]'
        dot += f'\nnode_{id} -> node_{id}_truncate;'
        return ReturnAST(dot, id)
```

UpdateTable

La clase UpdateTable representa una instrucción de actualización (UPDATE) en un lenguaje de programación. Esta instrucción se utiliza para modificar los valores de los campos en registros específicos de una tabla. La clase se inicializa con información sobre la línea y columna en el código fuente donde se encuentra, el nombre de la tabla (id), los campos a actualizar (fields), los nuevos valores (values) y la condición que determina qué registros se deben actualizar (condition).

El método execute se encarga de ejecutar la instrucción de actualización en el entorno actual. Llama al método updateTable del entorno, pasando el nombre de la tabla, los campos, los nuevos valores y la condición, junto con la posición en el código fuente.

El método ast genera una representación del nodo en el Árbol de Sintaxis Abstracta (AST) para esta instrucción de actualización. Crea un nodo con la etiqueta "UPDATE" y agrega nodos adicionales para representar los campos, valores y la condición en el AST.

```python
from statements.Abstracts.Instruction import Instruction
from statements.Abstracts.Expression import Expression
from utils.TypeInst import TypeInst
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnC3D, Type

class UpdateTable(Instruction):
    def __init__(self, line: int, column: int, id: str, fields: list[str], values: list[Expression], condition: Expression):
        super().__init__(line, column, TypeInst.UPDATE_TABLE)
        self.id = id
        self.fields = fields
        self.values = values
        self.condition = condition

    def execute(self, env: Env) -> any:
        env.updateTable(self.id, self.fields, self.values, self.condition, self.line, self.column)
        return

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass
```

```python
def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="UPDATE"];'
    dot += f'\nnode_{id}_set[label="SET"];'
    dot += f'\nnode_{id} -> node_{id}_set;'
    dot += f'\nnode_{id}_condition[label="CONDITION"];'
    dot += f'\nnode_{id} -> node_{id}_condition;'
    value: ReturnAST
    for i in range(len(self.fields)):
        dot += f'\nnode_{id}_field{i}[label="{self.fields[i]}"]'
        dot += f'\nnode_{id}_set -> node_{id}_field{i};'
        value = self.values[i].ast(ast)
        dot += f'\n{value.dot}'
        dot += f'\nnode_{id}_field{i} -> node_{value.id};'
    condition = self.condition.ast(ast)
    dot += f'\n{condition.dot}'
    dot += f'\nnode_{id}_condition -> node_{condition.id};'
    return ReturnAST(dot, id)
```

Truncate

La clase TruncateTable representa una instrucción de truncado (TRUNCATE) en un lenguaje de programación. Esta instrucción se utiliza para eliminar todos los registros de una tabla, pero no elimina la definición de la tabla en sí. La clase se inicializa con información sobre la línea y columna en el código fuente donde se encuentra, y el nombre de la tabla (id) que se va a truncar.

El método execute se encarga de ejecutar la instrucción de truncado en el entorno actual. Llama al método truncateTable del entorno, pasando el nombre de la tabla y la posición en el código fuente.

El método ast genera una representación del nodo en el Árbol de Sintaxis Abstracta (AST) para esta instrucción de truncado. Crea un nodo con la etiqueta "TRUNCATE" y agrega un nodo adicional para representar el nombre de la tabla en el AST.

```python
from statements.Abstracts.Instruction import Instruction
from statements.Abstracts.Expression import Expression
from utils.TypeInst import TypeInst
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class When(Instruction):
    def __init__(self, line: int, column: int, when_: Expression, result: Expression):
        super().__init__(line, column, TypeInst.WHEN)
        self.when_ = when_
        self.result = result
        self.whenEvaluate = None

    def setWhen(self, whenEvaluate: ReturnType):
        self.whenEvaluate = whenEvaluate

    def execute(self, env: Env) -> ReturnType:
        envWhen: Env = Env(env, f'{env.name} when')
        when_: ReturnType = self.when_.execute(envWhen)
        if self.whenEvaluate:
            whenE: ReturnType = self.whenEvaluate
            envWhen.name = f'{envWhen.name} {when_.value}'
            if when_.value == whenE.value:
                result: ReturnType = self.result.execute(envWhen)
                return result
        else:
            condition: ReturnType = self.when_.execute(env)
            if condition.value:
                return self.result.execute(env)
```

```python
        else:
            condition: ReturnType = self.when_.execute(env)
            if condition.value:
                return self.result.execute(env)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass

    def ast(self, ast: AST) -> ReturnAST:
        id = ast.getNewID()
        dot = f'node_{id}[label="WHEN"];'
        dot += f'node_{id}_cond[label="CONDICION"];'
        dot += f'node_{id}_result[label="RESULT"];'
        cond: ReturnAST = self.when_.ast(ast)
        result: ReturnAST = self.result.ast(ast)
        dot += '\n' + cond.dot
        dot += '\n' + result.dot
        dot += f'\nnode_{id}_cond -> node_{cond.id};'
        dot += f'\nnode_{id}_result -> node_{result.id};'
        dot += f'\nnode_{id} -> node_{id}_cond;'
        dot += f'\nnode_{id} -> node_{id}_result;'
        return ReturnAST(dot, id)
```

While

La clase While representa una instrucción de bucle "while" en un lenguaje de programación. Esta instrucción ejecuta un bloque de código mientras una condición dada sea verdadera. La clase se inicializa con información sobre la línea y columna en el código fuente donde se encuentra, la condición del bucle (condition), y el bloque de código que se ejecutará mientras la condición sea verdadera (block).

El método execute se encarga de ejecutar la instrucción de bucle "while". Crea un entorno específico para el bucle (whileEnv) y ejecuta repetidamente el bloque de código mientras la condición sea verdadera. Si el bloque de código devuelve un valor de TypeInst.CONTINUE, se salta a la siguiente iteración del bucle. Si devuelve TypeInst.BREAK, se sale del bucle. Si devuelve algún otro valor, se devuelve ese valor.

El método ast genera una representación del nodo en el Árbol de Sintaxis Abstracta (AST) para esta instrucción de bucle "while". Crea un nodo con la etiqueta "WHILE" y agrega nodos adicionales para representar la condición y el bloque del bucle en el AST

```python
from statements.Abstracts.Instruction import Instruction
from statements.Abstracts.Expression import Expression
from utils.TypeInst import TypeInst
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env


class While(Instruction):
    def __init__(self, line: int, column: int, condition: Expression, block: Instruction):
        super().__init__(line, column, TypeInst.LOOP_WHILE)
        self.condition = condition
        self.block = block

    def execute(self, env: Env) -> any:
        whileEnv: Env = Env(env, f'{env.name} while')
        condition: ReturnType = self.condition.execute(whileEnv)
        while condition.value:
            block: ReturnType = self.block.execute(whileEnv)
            if block:
                if block.value == TypeInst.CONTINUE:
                    condition = self.condition.execute(whileEnv)
                    continue
                elif block.value == TypeInst.BREAK:
                    break
                return block
            condition = self.condition.execute(whileEnv)

    def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
        pass
```

```python
def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="WHILE"];'
    dot += f'\nnode_{id}_cond[label="CONDICION"]'
    cond: ReturnAST = self.condition.ast(ast)
    dot += '\n' + cond.dot
    dot += f'\nnode_{id}_cond -> node_{cond.id};'
    inst: ReturnAST = self.block.ast(ast)
    dot += '\n' + inst.dot
    dot += f'\nnode_{id} -> node_{inst.id};'
    dot += f'\nnode_{id} -> node_{id}_cond;'
    return ReturnAST(dot, id)
```

When

La clase When representa una instrucción de control de flujo "when" en un lenguaje de programación. La instrucción "when" se utiliza para comparar una expresión (when_) con un conjunto de condiciones y ejecutar el bloque de código asociado a la primera condición verdadera. La clase se inicializa con información sobre la línea y columna en el código fuente donde se encuentra, la expresión de comparación (when_), y el resultado asociado (result).

La función setWhen se utiliza para establecer el valor evaluado de la expresión de comparación. El método execute se encarga de ejecutar la instrucción "when". Crea un entorno específico para el "when" (envWhen) y evalúa la condición. Si la expresión de comparación tiene un valor evaluado previamente (whenEvaluate), compara este valor con el valor de la expresión de comparación. Si son iguales, ejecuta el bloque de código asociado y devuelve el resultado. Si no hay un valor previamente evaluado, evalúa la condición y, si es verdadera, ejecuta el bloque de código asociado y devuelve el resultado.

El método ast genera una representación del nodo en el Árbol de Sintaxis Abstracta (AST) para esta instrucción "when". Crea un nodo con la etiqueta "WHEN" y agrega nodos adicionales para representar la condición y el resultado en el AST.

```python
from statements.Abstracts.Instruction import Instruction
from statements.Abstracts.Expression import Expression
from utils.TypeInst import TypeInst
from statements.Env.AST import AST, ReturnAST
from statements.Env.Env import Env
from statements.C3D.C3DGen import C3DGen
from utils.Type import ReturnType, ReturnC3D, Type

class When(Instruction):
    def __init__(self, line: int, column: int, when_: Expression, result: Expression):
        super().__init__(line, column, TypeInst.WHEN)
        self.when_ = when_
        self.result = result
        self.whenEvaluate = None

    def setWhen(self, whenEvaluate: ReturnType):
        self.whenEvaluate = whenEvaluate

    def execute(self, env: Env) -> ReturnType:
        envWhen: Env = Env(env, f'{env.name} when')
        when_: ReturnType = self.when_.execute(envWhen)
        if self.whenEvaluate:
            whenE: ReturnType = self.whenEvaluate
            envWhen.name = f'{envWhen.name} {when_.value}'
            if when_.value == whenE.value:
                result: ReturnType = self.result.execute(envWhen)
                return result
        else:
            condition: ReturnType = self.when_.execute(env)
            if condition.value:
                return self.result.execute(env)
```

```python
def compile(self, env: Env, c3dgen: C3DGen) -> ReturnC3D:
    pass

def ast(self, ast: AST) -> ReturnAST:
    id = ast.getNewID()
    dot = f'node_{id}[label="WHEN"];'
    dot += f'node_{id}_cond[label="CONDICION"];'
    dot += f'node_{id}_result[label="RESULT"];'
    cond: ReturnAST = self.when_.ast(ast)
    result: ReturnAST = self.result.ast(ast)
    dot += '\n' + cond.dot
    dot += '\n' + result.dot
    dot += f'\nnode_{id}_cond -> node_{cond.id};'
    dot += f'\nnode_{id}_result -> node_{result.id};'
    dot += f'\nnode_{id} -> node_{id}_cond;'
    dot += f'\nnode_{id} -> node_{id}_result;'
    return ReturnAST(dot, id)
```

Este código implementa una clase llamada Env que representa un entorno de ejecución para un lenguaje de programación. Un entorno de ejecución es responsable de gestionar variables, funciones y tablas en el contexto de la ejecución del programa.

Aquí hay una explicación detallada de algunas de las funciones y características clave de esta clase:

Variables

saveID: Se utiliza para guardar una variable en el entorno. Verifica si la variable ya existe y, si no, la agrega al diccionario de identificadores (ids). También actualiza la tabla de símbolos (symTable).

getValue: Obtiene el valor de una variable buscando en el entorno actual y en los entornos anteriores.

reasignID: Reasigna un nuevo valor a una variable existente después de verificar la compatibilidad de tipos.

Funciones

saveFunction: Guarda una función en el entorno. También actualiza la tabla de símbolos.

getFunction: Obtiene una función del entorno actual o de los entornos anteriores.

Tablas

Métodos como saveTable, insertTable, truncateTable, dropTable, etc., gestionan las operaciones relacionadas con las tablas. Estos métodos interactúan con las tablas y también actualizan la tabla de símbolos.

Operaciones relacionadas con la tabla de símbolos:

Se utiliza la tabla de símbolos (symTable) para realizar un seguimiento de las variables, funciones y tablas definidas en el programa.

Impresión y Errores

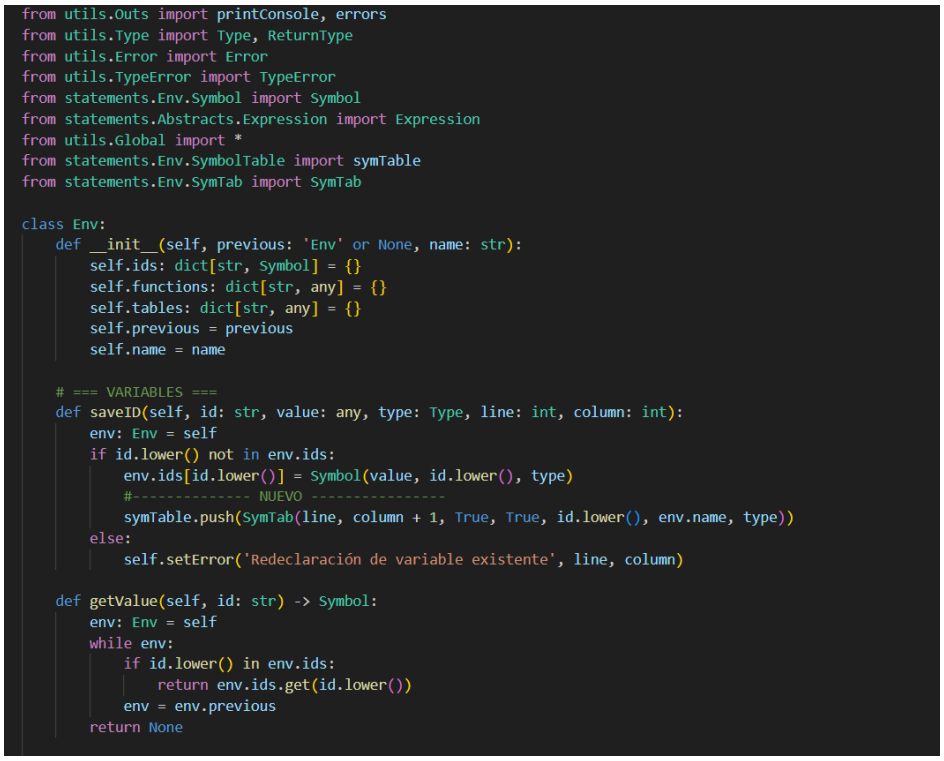
setPrint: Agrega un mensaje a la lista printConsole para imprimir más tarde.

selectPrint: Agrega los resultados de una selección a la lista printConsole.

setError: Maneja los errores semánticos, añadiéndolos a la lista errors.

Utilidades

getTypeOf: Obtiene el nombre de un tipo específico (INT, DECIMAL, NCHAR, etc.).



```python
from utils.Outs import printConsole, errors
from utils.Type import Type, ReturnType
from utils.Error import Error
from utils.TypeError import TypeError
from statements.Env.Symbol import Symbol
from statements.Abstracts.Expression import Expression
from utils.Global import *
from statements.Env.SymbolTable import symTable
from statements.Env.SymTab import SymTab

class Env:
    def __init__(self, previous: 'Env' or None, name: str):
        self.ids: dict[str, Symbol] = {}
        self.functions: dict[str, any] = {}
        self.tables: dict[str, any] = {}
        self.previous = previous
        self.name = name

    # === VARIABLES ===
    def saveID(self, id: str, value: any, type: Type, line: int, column: int):
        env: Env = self
        if id.lower() not in env.ids:
            env.ids[id.lower()] = Symbol(value, id.lower(), type)
            #-------------- NUEVO ----------------
            symTable.push(SymTab(line, column + 1, True, True, id.lower(), env.name, type))
        else:
            self.setError('Redeclaración de variable existente', line, column)

    def getValue(self, id: str) -> Symbol:
        env: Env = self
        while env:
            if id.lower() in env.ids:
                return env.ids.get(id.lower())
            env = env.previous
        return None
```

```
def reasignID(self, id: str, value: ReturnType, line: int, column: int) -> bool:
    env: Env = self
    while env:
        if id.lower() in env.ids:
            symbol: Symbol = env.ids.get(id.lower())
            if value.type == symbol.type or symbol.type == Type.DECIMAL and value.type == Type.INT or \
            symbol.type == Type.BIT and value.type == Type.INT and int(value.value) in [0, 1] or \
            symbol.type == Type.NCHAR and value.type == Type.NVARCHAR:
                symbol.value = value.value
                env.ids[id.lower()] = symbol
                return True
            env.setError(f'Los tipos no coinciden en la asignación. Intenta asignar un "{env.getTypeOf(value.type)}" a un "{env.getTypeOf(symbol
            return False
        env = env.previous
    self.setError('Resignación de valor a variable inexistente', line, column)
    return False

# === FUNCTIONS ===
def saveFunction(self, id: str, func: any):
    env: Env = self
    if not id.lower() in env.functions:
        env.functions[id.lower()] = func
        #------------ NUEVO ---------------
        symTable.push(SymTab(func.line, func.column + 1 , False, False, id.lower(), env.name, Type.NULL))
    else:
        self.setError('Redefinición de función existente', func.line, func.column)

def getFunction(self, id: str) -> any:
    env: Env = self
    while env:
        if id.lower() in env.functions:
            return env.functions.get(id.lower())
        env = env.previous
    return None
```

```
def saveTable(self, id: str, table: any, line: int, column: int):
    env: Env = self
    if not id.lower() in env.tables:
        env.tables[id.lower()] = table
        self.setPrint(f'Tabla \'{id.lower()}\' creada. {line}:{column + 1}')
        #------------ NUEVO ---------------
        symTable.push(SymTab(line, column + 1, False, False, id.lower(), env.name, Type.TABLE))
    else:
        self.setError('Redefinición de tabla existente', line, column)

def insertTable(self, id: str, fields: list[str], values: list[Expression], line: int, column: int) -> bool:
    env: Env = self
    while env:
        if id.lower() in env.tables:
            if env.tables.get(id.lower()).validateFields(fields):
                newRow: dict[str, list[any]] = env.tables.get(id.lower()).getFieldsRow()
                result: ReturnType
                dataXml = []
                for i in range(len(fields)):
                    result = values[i].execute(self)
                    newRow[fields[i].lower()] = [result.type, result.value]
                    dataXml.append({"value": result.value, "column": fields[i].lower()})
                if env.tables.get(id.lower()).insert(env, newRow, line, column):
                    #------------ NUEVO ---------------
                    res = xml.insert(getUsedDatabase(), id.lower(), dataXml)
                    if not res[0]:
                        self.setPrint(res[1])
                        return False
                    self.setPrint(f'Registro insertado exitosamente en Tabla \'{id.lower()}\'. {line}:{column + 1}')
                    return True
                return False
            self.setError(f'Inserta dato en columna inexistente en Tabla \'{id.lower()}\'', line, column)
            return False
        env = env.previous
    self.setError('Insertar en tabla inexistente', line, column)
    return False
```

```python
def truncateTable(self, id: str, line: int, column: int) -> bool:
    env: Env = self
    while env:
        if id.lower() in env.tables:
            env.tables.get(id.lower()).truncate()
            self.setPrint(f'Registros eliminados de Tabla \'{id.lower()}\'. {line}:{column + 1}')
            return True
        env = env.previous
    self.setError('Truncar tabla inexistente', line, column)
    return False

def dropTable(self, id: str, line: int, column: int) -> bool:
    env: Env = self
    while env:
        if id.lower() in env.tables:
            del env.tables[id.lower()]
            self.setPrint(f'Tabla \'{id.lower()}\' eliminada. {line}:{column + 1}')
            return True
        env = env.previous
    self.setError('Eliminación de tabla inexistente', line, column)
    return False

def deleteTable(self, id: str, condition: Expression, line: int, column: int):
    env: Env = self
    while env:
        if id.lower() in env.tables:
            env.tables.get(id.lower()).deleteWhere(condition, self)
            self.setPrint(f'Eliminación de Tabla \'{id.lower()}\'. {line}:{column + 1}')
            return
        env = env.previous
    self.setError('Eliminar registro en tabla inexistente', line, column)
    return False
```

```python
def deleteTable(self, id: str, condition: Expression, line: int, column: int):
    env: Env = self
    while env:
        if id.lower() in env.tables:
            env.tables.get(id.lower()).deleteWhere(condition, self)
            self.setPrint(f'Eliminación de Tabla \'{id.lower()}\'. {line}:{column + 1}')
            return
        env = env.previous
    self.setError('Eliminar registro en tabla inexistente', line, column)
    return False

def updateTable(self, id: str, fields: list[str], values: list[Expression], condition: Expression, line: int, column: int):
    env: Env = self
    while env:
        if id.lower() in env.tables:
            env.tables.get(id.lower()).updateWhere(condition, fields, values, self)
            self.setPrint(f'Tabla \'{id.lower()}\' actualizada. {line}:{column + 1}')
            return True
        env = env.previous
    self.setError('Actualizar registro en tabla inexistente', line, column)
    return False

def selectTable(self, id: str, fields: list[list[any]] or str, condition: Expression, line: int, column: int):
    env: Env = self
    while env:
        if id.lower() in env.tables:
            table = env.tables.get(id.lower()).select(fields, condition, self)
            self.setPrint(f'Selección en Tabla \'{id.lower()}\'. {line}:{column + 1}')
            env.selectPrint(table if table else [])
            return True
        env = env.previous
    self.setError('Selección en tabla inexistente', line, column)
    return False
```

```
def addColumn(self, id: str, newColumn: str, type: Type, line: int, column: int) -> bool:
    env: Env = self
    while env:
        if id.lower() in env.tables:
            if not newColumn.lower() in env.tables.get(id.lower()).fields:
                env.tables.get(id.lower()).addColumn(newColumn, type)
                self.setPrint(f'Columna {newColumn.lower()} insertada exitosamente en Tabla \'{id.lower()}\'. {line}:{column + 1}')
                return True
            self.setError(f'Ya hay una columna {newColumn.lower()} en Tabla \'{id.lower()}\'', line, column)
            return False
        env = env.previous
    self.setError('Alterar tabla inexistente', line, column)
    return False

def dropColumn(self, id: str, dropColumn: str, line: int, column: int) -> bool:
    env: Env = self
    while env:
        if id.lower() in env.tables:
            if dropColumn.lower() in env.tables.get(id.lower()).fields:
                env.tables.get(id.lower()).dropColumn(dropColumn)
                self.setPrint(f'Columna {dropColumn.lower()} eliminada exitosamente de la Tabla \'{id.lower()}\'. {line}:{column + 1}')
                return True
            self.setError(f'La columna {dropColumn.lower()} no existe en Tabla \'{id.lower()}\'', line, column)
            return False
        env = env.previous
    self.setError('Alterar tabla inexistente', line, column)
    return False
```

```
def renameTo(self, id: str, newId: str, line: int, column: int) -> bool:
    env: Env = self
    while env:
        if id.lower() in env.tables:
            table = env.tables.get(id.lower())
            if table:
                table.renameTo(newId.lower())
                env.tables[newId.lower()] = table
                del env.tables[id.lower()]
                self.setPrint(f'Tabla \'{id.lower()}\' renombrada como {newId.lower()}. {line}:{column + 1}')
                return True
        env = env.previous
    self.setError(f'La tabla \'{id.lower()}\' no existe', line, column)
    return False

def renameColumn(self, id: str, currentColumn: str, newColumn: str, line: int, column: int) -> bool:
    env: Env = self
    while env:
        if id.lower() in env.tables:
            if currentColumn.lower() in env.tables.get(id.lower()).fields:
                env.tables.get(id.lower()).renameColumn(currentColumn.lower(), newColumn.lower())
                self.setPrint(f'Columna {currentColumn.lower()} actualizada exitosamente a {newColumn.lower()}. {line}:{column + 1}')
                return True
            self.setError(f'La columna {currentColumn.lower()} no existe en Tabla \'{id.lower()}\'', line, column)
            return False
        env = env.previous
    self.setError('Alterar tabla inexistente', line, column)
    return False
```

```python
# === UTILS ===
def setPrint(self, print_: str):
    printConsole.append([print_])

def selectPrint(self, select: list[list[any]]):
    printConsole.extend(select)

def setError(self, errorD: str, line: int, column: int):
    if not self.match(errorD, line, column + 1):
        errors.append(Error(line, column + 1, TypeError.SEMANTIC, errorD))

def match(self, err: str, line: int, column: int):
    for error in errors:
        if(error.__str__() == (Error(line, column, TypeError.SEMANTIC, err)).__str__()):
            return True
    return False

def getTypeOf(self, type: Type) -> str:
    match type:
        case Type.INT:
            return "INT"
        case Type.DECIMAL:
            return "DECIMAL"
        case Type.NCHAR:
            return "NCHAR"
        case Type.NVARCHAR:
            return "NVARCHAR"
        case Type.BIT:
            return "BIT"
        case Type.DATE:
            return "DATE"
        case Type.TABLE:
            return "TABLE"
        case _:
            return "NULL"
```

Abstract Sintact Tree (AST)

Este código define dos clases, AST y ReturnAST, que pueden ser parte de un sistema de construcción y representación de un Árbol de Sintaxis Abstracta (AST) en el contexto de un compilador o intérprete.

Clase AST (Árbol de Sintaxis Abstracta):

__init__ Método del Constructor:

Propósito: Inicializa una instancia de la clase AST.

Atributos

nodeID: Es un atributo entero que se inicializa en 0. Este atributo se utiliza para asignar identificadores únicos a los nodos del árbol.

getNewID Método:

Propósito: Devuelve un nuevo identificador único cada vez que es llamado.

Funcionamiento: Incrementa nodeID en 1 y devuelve el nuevo valor. Este método es útil para asignar identificadores únicos a los nodos del árbol durante su construcción.

Clase ReturnAST

__init__ Método del Constructor:

Propósito: Inicializa una instancia de la clase ReturnAST.

Parámetros

dot: Es un string que representa la información del nodo en formato DOT, que es utilizado para visualizar el árbol.

id: Es un entero que representa el identificador único del nodo en el árbol.

Atributos:

dot: Almacena la información del nodo en formato DOT.

id: Almacena el identificador único del nodo.

Estas clases se utilizan para mantener y gestionar información relacionada con la construcción y representación del AST. La clase AST se encarga de generar identificadores únicos para los nodos del árbol, y la clase ReturnAST almacena información específica de un nodo para su posterior uso, como en la visualización del árbol.

```python
class AST:
    def __init__(self):
        self.nodeID: int = 0

    def getNewID(self) -> int:
        self.nodeID += 1
        return self.nodeID

class ReturnAST:
    def __init__(self, dot: str, id: int):
        self.dot = dot
        self.id = id
```

Tabla de símbolos

```python
from statements.Env.SymTab import SymTab

class SymbolTable:
    def __init__(self):
        self.symbols = []

    def push(self, sym: SymTab):
        if self.validateSymbol(sym):
            self.symbols.append(sym)

    def pop(self, id):
        for i in range(len(self.symbols)):
            if self.symbols[i].id == id:
                self.symbols.pop(i)
                break

    def validateSymbol(self, sym: SymTab):
        for i in self.symbols:
            if i.hash() == sym.hash():
                return False
        return True

    def getDot(self):
        dot = 'digraph SymbolsTable {graph[fontname="Arial" labelloc="t" bgcolor="#252526" fontcolor="white"];node[shape=none fontname="Arial"];labe
        for i in range(len(self.symbols)):
            self.symbols[i].num = i + 1
            dot += self.symbols[i].getDot()
        dot += '</table>>];}'
        return dot

    def print(self):
        print('TABLA DE SÍMBOLOS')
        for sym in self.symbols:
            print(sym.toString())
```

```python
    def print(self):
        print('TABLA DE SÍMBOLOS')
        for sym in self.symbols:
            print(sym.toString())

    def splice(self):
        self.symbols.clear()

    def toString(self):
        table = '╔═' + '═'.repeat(69) + '═╗'
        table += '\n║ ' + ' '.repeat(26) + 'TABLA DE SÍMBOLOS' + ' '.repeat(26) + ' ║'
        table += '\n╠═' + '═'.repeat(20) + '═╦═' + '═'.repeat(10) + '═╦═' + '═'.repeat(15) + '═╦═' + '═'.repeat(5) + '═╦═' + '═'.repeat(7) + '═╣'
        table += '\n║ ' + 'ID'.padEnd(20) + ' ║ ' + 'TIPO'.padEnd(10) + ' ║ ' + 'ENTORNO'.padEnd(15) + ' ║ ' + 'LINEA'.padEnd(5) + ' ║ ' + 'COLUMNA'
        table += '\n╠═' + '═'.repeat(20) + '═╬═' + '═'.repeat(10) + '═╬═' + '═'.repeat(15) + '═╬═' + '═'.repeat(5) + '═╬═' + '═'.repeat(7) + '═╣'
        for sym in self.symbols:
            table += '\n' + sym.toString()
        table += '\n╚═' + '═'.repeat(20) + '═╩═' + '═'.repeat(10) + '═╩═' + '═'.repeat(15) + '═╩═' + '═'.repeat(5) + '═╩═' + '═'.repeat(7) + '═╝'
        return table

symTable = SymbolTable()
```

```python
from utils.Type import Type

class Symbol:
    def __init__(self, value: any, id: str, type: Type):
        self.value = value
        self.id = id.lower()
        self.type = type

    def __str__(self) -> str:
        return f'{self.id}: {self.type} = {self.value}'
```

```python
from utils.Type import Type


class SymTab:
    def __init__(self, line, column, isVariable, isPrimitive, id, nameEnv, type: Type):
        self.num = 0
        self.line = line
        self.column = column
        self.isVariable = isVariable
        self.isPrimitive = isPrimitive
        self.id = id
        self.nameEnv = nameEnv
        self.type = type

    def toString(self):
        return '| ' + f'{self.id}'.ljust(20) + ' | ' + f'{self.getType(self.type)}'.ljust(10) + ' | ' + f'{self.nameEnv}'.ljust(15) + ' | ' + f'{sel

    def hash(self):
        return f'{self.id}_{self.type}_{self.nameEnv}_{self.line}_{self.column}_{self.isVariable}_{self.isPrimitive}'

    def getDot(self):
        if self.isPrimitive or self.isVariable:
            if self.isPrimitive:
                if self.isVariable:
                    return f'<tr><td bgcolor="white">{self.num}</td><td bgcolor="white">{self.id}</td><td bgcolor="white">Variable</td><td bgcolor="
                return f'<tr><td bgcolor="white">{self.num}</td><td bgcolor="white">{self.id}</td><td bgcolor="white">Función</td><td bgcolor="white
        if self.type != Type.NULL:
            return f'<tr><td bgcolor="white">{self.num}</td><td bgcolor="white">{self.id}/td><td bgcolor="white">Función</td><td bgcolor="white">{se
        return f'<tr><td bgcolor="white">{self.num}</td><td bgcolor="white">{self.id}</td><td bgcolor="white">Método</td><td bgcolor="white">{self.g
```

```python
def getType(self, type: Type):
    switcher = {
        Type.BIT: "BIT",
        Type.INT: "INT",
        Type.DECIMAL: "DECIMAL",
        Type.NCHAR: "VARCHAR",
        Type.NVARCHAR: "VARCHAR",
        Type.DATETIME: "DATETIME",
        Type.BOOLEAN: "BOOLEAN",
        Type.DATE: "DATE",
        Type.TABLE: "TABLE",
        Type.NULL: "NULL"
    }
    return switcher.get(type, "UNKNOWN")
```

Table

Este código define una serie de clases (Data, Field, y Table) que representan estructuras de datos para manejar tablas y operaciones relacionadas, como inserción, actualización y selección en una tabla. Aquí está una explicación detallada:

Clase Data:

Propósito

Representa un dato con su tipo asociado.

Métodos

__init__(self, type: Type, value: any): Inicializa una instancia de la clase con un tipo y valor específicos.

update(self, value: any): Actualiza el valor del dato.

getData(self) -> ReturnType: Devuelve el valor y tipo del dato en un objeto ReturnType.

Clase Field

Propósito

Representa un campo en una tabla con su tipo, valores, longitud, y propiedades.

Métodos

__init__(self, type: Type, values: list[Data], length: int, notNull: bool, isPrimary: bool): Inicializa una instancia de la clase con un tipo, lista de valores, longitud y propiedades como notNull e isPrimary.

slice(self): Borra todos los valores actuales del campo.

updateLength(self, n: int): Actualiza la longitud del campo.

Clase Table

Propósito

Representa una tabla con sus campos, filas y operaciones relacionadas.

Métodos

__init__(self, name: str, attribs: list[Attribute | ForeignKey]): Inicializa una instancia de la clase con un nombre y una lista de atributos o claves foráneas.

insert(self, env: Env, fields: dict[str, list[any]], line: int, column: int) -> bool: Inserta una fila en la tabla.

validate(self, env: Env, fields: dict[str, list[any]], line: int, column: int) -> bool: Valida los tipos de datos para la inserción.

validateFields(self, names: list[str]) -> bool: Valida si los campos existen en la tabla.

truncate(self): Elimina todas las filas de la tabla.

addColumn(self, newColumn: str, type: Type): Agrega una columna a la tabla.

dropColumn(self, column: str): Elimina una columna de la tabla.

renameTo(self, newId: str): Cambia el nombre de la tabla.

renameColumn(self, currentColumn: str, newColumn: str): Cambia el nombre de una columna.

createTmpFields(self) -> dict[str, Field]: Crea un diccionario temporal de campos.

deleteWhere(self, condition: Expression, env: Env): Elimina filas basadas en una condición.

updateWhere(self, condition: Expression, fields: list[str], values: list[Expression], env: Env): Actualiza filas basadas en una condición.

getTable(self, fields: dict[str, Field], rows: int) -> str: Obtiene una representación en cadena de la tabla.

getFieldsRow(self) -> dict[str, list[any]]: Obtiene una fila de valores predeterminados para insertar.

Codigo De Tres Direcciones(C3D)

Este código parece ser una implementación de un generador de código en tres direcciones (C3D) en Python. El C3D es un lenguaje de bajo nivel utilizado en compiladores para representar código intermedio. Aquí hay una descripción general de las clases y funciones definidas:

Clases de Instrucciones

Instruction: Clase base para representar instrucciones C3D.

Type: Enumeración de tipos de instrucciones C3D (GOTO, IF, LABEL, etc.).

Otras clases como Label, If, Goto, Asign, Expression, Generic, Printf, Function, Return, End, CallFunction, SetHeap, GetHeap, SetStack, y GetStack representan diferentes tipos de instrucciones C3D específicas.

Clase C3DGen

Constructor __init__: Inicializa varias listas y diccionarios para almacenar instrucciones y gestionar temporales.

Métodos como enableMain, enableNatives, enableFunction, y enableGlobal se utilizan para habilitar diferentes modos de generación de código.

Métodos como addInstruction, addLabel, addIf, addGoto, addAsign, etc., se utilizan para agregar instrucciones específicas al código C3D.

Métodos como generatePrintString, generateConcatString, etc., generan funciones nativas en C3D si aún no se han generado.

Generación de Código

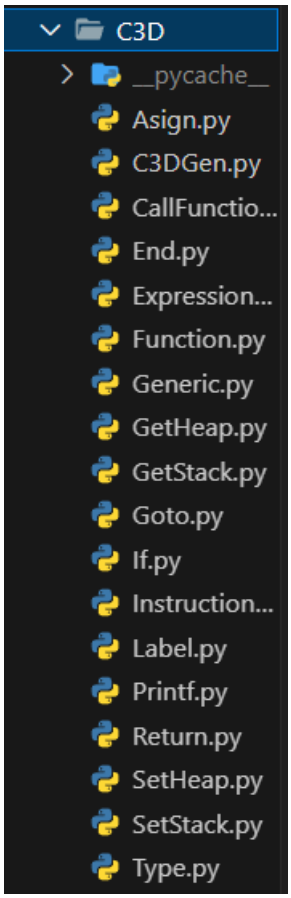
La función generateFinalCode construye el código final, incluyendo encabezados y definiciones de arreglos.

getFinalCode realiza algunas transformaciones en el código generado, como cambiar etiquetas y nombres de temporales.

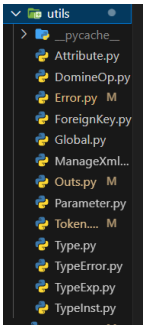
Manejo de Temporales y Etiquetas

El código utiliza contadores (labelCount y temporalCount) para generar nombres únicos para etiquetas y temporales.

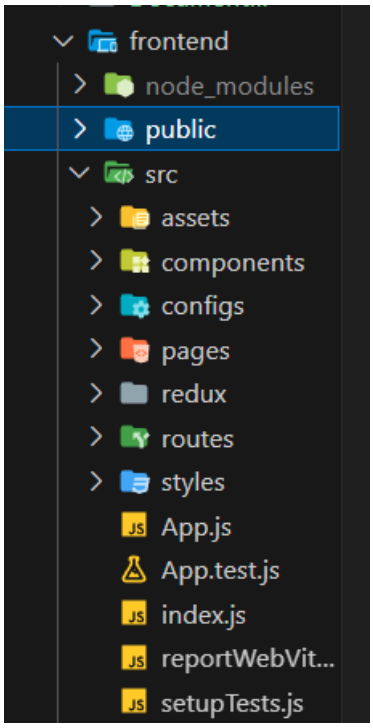
Se mantiene un seguimiento de las instrucciones generadas para evitar repeticiones innecesarias de GOTO.



Clases de apoyo



Frontend


frontend
node_modules
public
src
assets
components
configs
pages
redux
routes
styles
App.js
App.test.js
index.js
reportWebVit...
setupTests.js