Compilación

En este reporte se estarán analizando y desarrollando las diferentes partes del transpilador del lenguaje RoadToCivilization al lenguaje python, entre ellas están, el lexer, la gramática, el parser, el análisis semántico y el transpilado a python.

Lexer

Nuestro lenguaje está formado por conjunto de expresiones regulares cada una asociada a un tipo de token, la unión de estas expresiones regulares sería el conjunto de tokens que conforman nuestro lenguaje. El lexer se encarga de verificar que todos los tokens que nos entren pertenezcan al lenguaje.

Para la realización del lexer se crearon todas las expresiones regulares que conforman el lenguaje, seguido a esto se verifica,haciendo match, que cada token pertenezca al conjunto de expresiones regulares ya definidos. Cada expresión regular tiene un nombre para identificar que tipo de token es, luego se devuelven el token y su tipo si no hubo ningún problema al hacer match.

Aquí también se verificó que los strings sean correctos, por ejemplo, si empiezan a escribir un string y hacen un salto de línea sería incorrecto y el lexer devuelve error de sintaxis.

Nosotros implementamos dos lexer, uno usando la librería re y otro sin usar la misma, ambos funcionan correctamente, pero el que utilizamos es el que usa la librería re, puesto que es un poquito más rápido y el código es más legible.

In [ ]:
```
import os
from sys import path
path.append(os.path.abspath(os.path.join('',os.pardir)))

def tokenize(self,text):
    while len(text) > 0:

        match = self.regex.match(text)
        error_token = ''

        while not match:
            error_token += text[0]
            text = text[1:]
            if len(text) <= 0: break
            match = self.regex.match(text)

        if error_token:
            self.errors.append(f'Syntax error, unexpexted token "{error_token}" at line {self.line}')
            if len(text) <= 0: continue

        lexeme = match.group()

        if lexeme == '\n':
            self.line += 1

        # STRINGS
        elif lexeme == '"':
            text = text[1:]
            while len(text) > 0:
                c = text[0]
                text = text[1:]

                if c == '\\':
                    if text[0] == 'b':
                        lexeme += '\\b'

                    elif text[0] == 't':
                        lexeme += '\\t'

                    elif text[0] == 'n':
                        lexeme += '\\n'

                    elif text[0] == 'f':
                        lexeme += '\\f'

                    else:
                        lexeme += text[0]

                    text = text[1:]

                elif c == '\n':
                    self.errors.append(f'Syntax error at line {self.line} : Undefined string')
                    self.line += 1
                    break

                elif c == '\0':
                    self.errors.append(f'Syntax error at line {self.line} : String cannot contain the null char
```

```
                else:
                    lexeme += c
                    if c == '"':
                        break
            else:
                self.errors.append(f'Syntax error at line {self.line} : String cannot contain EOF')

        token_type = match.lastgroup if lexeme.lower() not in self.keywords and match.lastgroup is not None els

        yield lexeme, token_type, self.line

        text = text[match.end():] if lexeme[0] != '"' else text

    yield '$', self.eof, self.line
```

AST

Para la construcción del AST se hizo una jerarquía de clases que se compone de la siguiente forma.

Primeramente, tenemos los nodos DeclarationNode y ExpressionNode.

De DeclarationNode heredan:

- DeclarationEntity
- DeclarationVar
- FuncDeclaration

De ExpressionNode heredan:

- AsignationVar
- FactorNode
  - BooleanNode
  - StringNode
  - NumberNode
  - VariableNode
  - FunctionName
- ListNode
- IndexListNode
- IfElse
- Not
- InstanceFunction
- WhileNode
- BinaryNode
  - And
  - Or
  - LessThan
  - MoreThan
  - EqualEqual
  - PlusNode
  - MinusNode
  - StarNode
  - DivNode

In [ ]:
```
class ast_nodes:

    class Node:
        pass


    class ProgramNode(Node):
        def __init__(self, declarations):
            self.declarations = declarations


    class DeclarationNode(Node):
        pass
    class ExpressionNode(Node):
        pass
```

Gramática

El diseño de la gramática fue algo bastante trabajoso, fue una de las partes más difíciles, para modelarla se usó la clase Grammar que se encuentra en en la carpeta cmp.

Primeramente definimos los terminales y no terminales, a través de los métodos Terminal y Terminals, y NonTerminal y NonTerminals respectivamente. Con startSymbol accedemos al símbolo distinguido e indicamos cuál es. Luego definimos las producciones haciendo uso del operador %=, con G.Epsilon accederemos a epsilon como su nombre lo indica. El símbolo de fin de cadena se modelará con la clase EOF, que no debe ser instanciada directamente con el constructor, una instancia G de Grammar lo construirá automáticamente.

También, en la gramática, junto con las producciones se construirán los nodos del AST que denota a cada operación, haciendo uso de funciones lambdas que devuleven un nodo o una lista de nodos.

In [ ]:
```
from cmp.pycompiler import Grammar
from Compilacion.astree.AST_Nodes import ast_nodes

def build_grammar():
    Gram = Grammar()

    #No Terminales
    program = Gram.NonTerminal('<program>', startSymbol=True)
    simulation = Gram.NonTerminal('<simulation>')
    declarationnt, arith_expr, funct, term = Gram.NonTerminals('<declarationnt> <arith_expr> <funct> <term>')
    crudcharnt, ifnt, cond, whilent = Gram.NonTerminals('<crudcharnt> <ifnt> <cond> <whilent>')
    asignationnt, arg_list, type_var = Gram.NonTerminals('<asignationnt> <arg_list> <type_var>')
    arg_types_list, functnt, factor = Gram.NonTerminals('<arg_types_list> <functnt> <factor>')
    elseblock, listnt, block, listindexed = Gram.NonTerminals('<elseblock> <listnt> <block> <listindexed>')

    #Terminales
    open_parenthesis, closed_parenthesis, equal, colon, plus, minus, star, div = Gram.Terminals('( ) = , + - *
    semicolon, point, open_bracket, closed_bracket, open_square, closed_square = Gram.Terminals('; . { } [ ]')
    ift, elset, whilet, true, false, string, number = Gram.Terminals('if else while true false string number')
    entity, var, funct_name, typet = Gram.Terminals('entity var funct_name type')
    arg, lessthan, morethan, equalequal, andt, ort, nott = Gram.Terminals('arg < > == and or not')



    #Producciones
    program %=  simulation, lambda h,s: ast_nodes.ProgramNode(s[1])

    simulation %= declarationnt + semicolon, lambda h,s: [s[1]]
    simulation %= declarationnt + semicolon + simulation, lambda h,s: [s[1]] + s[3]
    simulation %= asignationnt + semicolon, lambda h,s: [s[1]]
    simulation %= asignationnt + semicolon + simulation, lambda h,s: [s[1]] + s[3]
    simulation %= ifnt, lambda h,s: [s[1]]
    simulation %= ifnt + simulation, lambda h,s: [s[1]] + s[2]
    simulation %= whilent, lambda h,s: [s[1]]
    simulation %= whilent + simulation, lambda h,s: [s[1]] + s[2]
    simulation %= functnt, lambda h,s: [s[1]]
    simulation %= functnt + simulation, lambda h,s: [s[1]] + s[2]
    simulation %= crudcharnt + semicolon, lambda h,s: [s[1]]
    simulation %= crudcharnt + semicolon + simulation, lambda h,s: [s[1]] + s[3]
    simulation %= funct + semicolon, lambda h,s: [s[1]]
    simulation %= funct + semicolon + simulation, lambda h,s: [s[1]] + s[3]

    block %= declarationnt + semicolon + block, lambda h,s: [s[1]] + s[3]
    block %= asignationnt + semicolon + block, lambda h,s: [s[1]] + s[3]
    block %= ifnt + block, lambda h,s: [s[1]] + s[2]
    block %= whilent + block, lambda h,s: [s[1]] + s[2]
    block %= crudcharnt + semicolon + block, lambda h,s: [s[1]] + s[3]
    block %= factor + semicolon + block, lambda h,s: [s[1]] + s[3]
    block %= Gram.Epsilon, lambda h,s: []

    arg_list %= cond, lambda h,s: [s[1]]
    arg_list %= arg_list + colon + cond, lambda h,s: s[1] + [s[3]]
    arg_list %= Gram.Epsilon, lambda h,s:  []


    declarationnt %= typet + var + equal + typet + open_parenthesis + arg_list + closed_parenthesis, lambda h,s
    declarationnt %= typet + var + equal + arith_expr, lambda h,s: ast_nodes.DeclarationVar(s[1],s[2],s[4])

    asignationnt %= var + equal + arith_expr, lambda h,s: ast_nodes.AsignationVar(s[1],s[3])

    arith_expr %= arith_expr + plus + term, lambda h,s: ast_nodes.PlusNode(s[1],s[3])
    arith_expr %= arith_expr + minus + term, lambda h,s: ast_nodes.MinusNode(s[1],s[3])
    arith_expr %= term, lambda h,s:s[1]

    term %= term + star + factor, lambda h,s: ast_nodes.StarNode(s[1],s[3])
    term %= term + div + factor, lambda h,s: ast_nodes.DivNode(s[1],s[3])
    term %= cond, lambda h,s: s[1]

    factor %= open_parenthesis + arith_expr + closed_parenthesis, lambda h,s: s[2]
    factor %= string, lambda h,s: ast_nodes.StringNode(s[1])
    factor %= number, lambda h,s: ast_nodes.NumberNode(s[1])
    factor %= true, lambda h,s: ast_nodes.BooleanNode(s[1])
    factor %= false, lambda h,s: ast_nodes.BooleanNode(s[1])
    factor %= var, lambda h,s: ast_nodes.VariableNode(s[1])
    factor %= funct, lambda h,s: s[1]
    factor %= listnt, lambda h,s: s[1]
    factor %= listindexed, lambda h,s: s[1]
    factor %= funct_name, lambda h,s: ast_nodes.FunctionName(s[1])
```

```
    listnt %= open_square + arg_list + closed_square, lambda h,s: ast_nodes.ListNode(s[2])
    listindexed %= factor + open_square + factor + closed_square, lambda h,s: ast_nodes.IndexListNode(s[1], s[3

    ifnt %= ift + open_parenthesis + cond +closed_parenthesis + open_bracket + block + closed_bracket + elseblo

    elseblock %= elset + open_bracket + block + closed_bracket, lambda h,s: s[3]
    elseblock %= Gram.Epsilon, lambda h,s: []

    cond %= factor, lambda h,s: s[1]
    cond %= nott + cond, lambda h,s: ast_nodes.Not(s[2])
    cond %= factor + andt + cond, lambda h,s: ast_nodes.And(s[1],s[3])
    cond %= factor + ort + cond, lambda h,s: ast_nodes.Or(s[1],s[3])
    cond %= factor + lessthan + factor, lambda h,s: ast_nodes.LessThan(s[1],s[3])
    cond %= factor + morethan + factor, lambda h,s: ast_nodes.MoreThan(s[1],s[3])
    cond %= factor + equalequal + factor, lambda h,s: ast_nodes.EqualEqual(s[1],s[3])

    funct %= funct_name + open_parenthesis + arg_list + closed_parenthesis, lambda h,s: ast_nodes.InstanceFunct
    funct %= factor + point + funct_name + open_parenthesis + arg_list + closed_parenthesis, lambda h,s: ast_no

    whilent %= whilet + open_parenthesis + cond +closed_parenthesis + open_bracket + block + closed_bracket, la

    functnt %= typet + funct_name + open_parenthesis + arg_types_list + closed_parenthesis + open_bracket + blo

    arg_types_list %= type_var, lambda h,s: [s[1]]
    arg_types_list %= arg_types_list + colon + type_var, lambda h,s: s[1] + [s[3]]
    arg_types_list %= Gram.Epsilon, lambda h,s: []

    type_var %= typet + var, lambda h,s: [s[1], s[2]]
    type_var %= entity + var, lambda h,s: [s[1], s[2]]

    return Gram
```

Parser

El parser construye las tablas action y goto, a partir, de la gramática, para después pasarle la lista de tokens y ver que se corresponda a lo definido en la gramática y no haya errores sintácticos.

Ahora analizaremos como se construyó el parser.

In [ ]:
```
from cmp.utils import ContainerSet
from cmp.pycompiler import Item
from cmp.automata import State, multiline_formatter
```

First

El conjunto first puede ser computado para los terminales, no terminales y producciones de la gramática, los first se inicializan vacíos y se van actualizando siguiendo las reglas vistas en conferencias:

- Si X $\rightarrow$ W1 | W2 | ... | Wn entonces por definición, First(X) = $\cup_i$ First(W_i).
- Si X $\rightarrow \epsilon$ entonces $\epsilon \in$ First(X).
- Si W = xZ donde x es un terminal, entonces trivialmente First(W) = { x }.
- Si W = YZ donde Y es un no-terminal y Z una forma oracional, entonces First(Y) $\subseteq$ First(W).
- Si W = YZ y $\epsilon \in$ First(Y) entonces First(Z) $\subseteq$ First(W).

El algoritmo se hizo en dos partes, primeramente se hizo el método compute_local_first, donde se calculará el First(alpha), siendo alpha una forma oracional, después de esto se hace el método compute_first para calcular todos los conjuntos first, donde se van actualizando los conjuntos iniciales según lo que devuelve compute_local_first.

In [ ]:
```
def compute_local_first(firsts, alpha):
    first_alpha = ContainerSet()

    try:
        alpha_is_epsilon = alpha.IsEpsilon
    except:
        alpha_is_epsilon = False

    # alpha == epsilon ? First(alpha) = { epsilon }
    if alpha_is_epsilon:
        first_alpha.set_epsilon()

    # alpha = X1 ... XN
    else:
        for item in alpha:
            first_symbol = firsts[item]
    # First(Xi) subconjunto First(alpha)
            first_alpha.update(first_symbol)
```

```
    # epsilon pertenece a First(X1)...First(Xi) ? First(Xi+1) subconjunto de First(X) y First(alpha)
            if not first_symbol.contains_epsilon:
                break
    # epsilon pertenece a First(X1)...First(XN) ? epsilon pertence a First(X) y al First(alpha)
        else:
            first_alpha.set_epsilon()

    # First(alpha)
    return first_alpha
```

In [ ]:
```
def compute_firsts(G):
    firsts = {}
    change = True

    # init First(Vt)
    for terminal in G.terminals:
        firsts[terminal] = ContainerSet(terminal)

    # init First(Vn)
    for nonterminal in G.nonTerminals:
        firsts[nonterminal] = ContainerSet()

    while change:
        change = False

        # P: X -> alpha
        for production in G.Productions:
            X = production.Left
            alpha = production.Right

            # get current First(X)
            first_X = firsts[X]

            # init First(alpha)
            try:
                first_alpha = firsts[alpha]
            except:
                first_alpha = firsts[alpha] = ContainerSet()

            # CurrentFirst(alpha)???
            local_first = compute_local_first(firsts, alpha)

            # update First(X) and First(alpha) from CurrentFirst(alpha)
            change |= first_alpha.hard_update(local_first)
            change |= first_X.hard_update(local_first)

    # First(Vt) + First(Vt) + First(RightSides)
    return firsts
```

Shift Reduce

La implementación de este parser es básicamente el concepto visto en conferencia puesto en código.

Si se reconoce la tupla (el estado y el tipo del token en que está), indexa en la tabla action esta tupla que nos da la próxima operación a hacer (shift o reduce) y el estado al que se mueve en la siguiente iteración.

Se crea una clase ShiftReduceParser donde redefinimos el método **call** a partir de lo antes dicho.

In [ ]:
```
class ShiftReduceParser:
    SHIFT = 'SHIFT'
    REDUCE = 'REDUCE'
    OK = 'OK'

    def __init__(self, G, verbose=False):
        self.G = G
        self.verbose = verbose
        self.action = {}
        self.goto = {}
        self._build_parsing_table()
        self.error = ''

    def _build_parsing_table(self):
        raise NotImplementedError()

    def __call__(self, w):
        stack = [ 0 ]
        cursor = 0
        output = []
        operations = []

        while True:
            state = stack[-1]
            lookahead = w[cursor]
            if self.verbose: print(stack, '<---||--->', w[cursor:])
```

```
            try:
                action, tag = self.action[(state, self.G[lookahead.token_type])]
                if action == self.SHIFT:
                    operations.append(self.SHIFT)
                    stack.append(tag)
                    cursor += 1

                elif action == self.REDUCE:
                    operations.append(self.REDUCE)
                    output.append(tag)
                    for _ in tag.Right: stack.pop()
                    a = self.goto[stack[-1],tag.Left]
                    stack.append(a)


                elif action == self.OK:
                    return output,operations
                else:
                    raise NameError
            except:
                self.error = f'Syntax error at line {lookahead.line}'
                return None, None
```

Clausura LR1

Para realizarla primeramente, nos apoyaremos en dos métodos que implementamos el expand y el compress.

- expand (Y $\rightarrow \alpha$. X δ, c) = { X $\rightarrow$. β, b | b $\in$ First(δ c) } este devuelve un conjunto de items que sugiere incluir.
- compress coge un conjunto de items LR1 y devuelve el mismo conjunto pero los items que tienen el mismo centro están unidos.

Conociendo el concepto de clausura LR1 visto en conferencia, básicamente se traduce a hacer expand por los items y después hacerle compress a el conjunto.

In [ ]:
```
from Compilacion.parse.auxiliar import expand, compress

def closure_lr1(items, firsts):
    closure = ContainerSet(*items)

    changed = True
    while changed:
        changed = False

        new_items = ContainerSet()

        for item in closure:
            new_items.extend(expand(item, firsts))

        changed = closure.update(new_items)

    return compress(closure)
```

Goto LR1

Goto(I,X) = CL({ Y $\rightarrow$ α X. β, c | Y $\rightarrow$ α .X β, c $\in$ I })

El método setea just_kernel = true para calcular solamente el conjunto de items kernel y no todo el conjunto, sino, se tiene el conjunto first para calcular la clausura lr1.

In [ ]:
```
def goto_lr1(items, symbol, firsts=None, just_kernel=False):
    assert just_kernel or firsts is not None, '`firsts` must be provided if `just_kernel=False`'
    items = frozenset(item.NextItem() for item in items if item.NextSymbol == symbol)
    return items if just_kernel else closure_lr1(items, firsts)
```

Autómata LR1

Para construir el autómata partimos del estado inicial Program' -> .Program,$, luego se van haciendo trancisiones con los terminales y no terminales, usando como función de trancisión goto_lr1.

In [ ]:
```
def build_LR1_automaton(G):
    assert len(G.startSymbol.productions) == 1, 'Grammar must be augmented'

    firsts = compute_firsts(G)
    firsts[G.EOF] = ContainerSet(G.EOF)

    start_production = G.startSymbol.productions[0]
    start_item = Item(start_production, 0, lookaheads=(G.EOF,))
    start = frozenset([start_item])

    closure = closure_lr1(start, firsts)
    automaton = State(frozenset(closure), True)
```

```
    pending = [ start ]
    visited = { start: automaton }

    while pending:
        current = pending.pop()
        current_state = visited[current]

        for symbol in G.terminals + G.nonTerminals:
            kernels = goto_lr1(current_state.state, symbol, just_kernel=True)

            if not kernels:
                continue

            try:
                next_state = visited[kernels]
            except KeyError:
                pending.append(kernels)
                visited[pending[-1]] = next_state = State(frozenset(goto_lr1(current_state.state, symbol, first

            current_state.add_transition(symbol.Name, next_state)

    automaton.set_formatter(multiline_formatter)
    return automaton
```

LR1 Canónico

Las tablas goto y action se llenan a partir de las reglas vistas en conferencias:

- Sea "$X \rightarrow \alpha . c\omega, s$" un item del estado I_i y $Goto(I_i, c) = I_j$.
  Entonces $ACTION[I_i, c] = `S_j`$.
- Sea "$X \rightarrow \alpha . , s$" un item del estado I_i.
  Entonces $ACTION[I_i, s] = `R_k`$ (producción `k` es $X \rightarrow \alpha$).
- Sea I_i el estado que contiene el item "$S' \rightarrow S. , \$$" (S' distinguido). Entonces $ACTION[I_i, \$] = `OK`$.
- Sea "$X \rightarrow \alpha . Y\omega, s$" item del estado I_i y $Goto(I_i, Y) = I_j$.
  Entonces $GOTO[I_i, Y] = j$.

El algoritmo es básicamente, un grupo de if else siguiendo las reglas antes mencionadas. El algoritmo se hace en un método llamado _build_parsing_table que está dentro de una clase LR1Parser que hereda de la clase ShiftReduceParser.

In [ ]:
```
class LR1Parser(ShiftReduceParser):

    def _build_parsing_table(self):
        G = self.G.AugmentedGrammar(True)

        automaton = build_LR1_automaton(G)
        for i, node in enumerate(automaton):
            if self.verbose: print(i, '\t', '\n\t '.join(str(x) for x in node.state), '\n')
            node.idx = i

        for node in automaton:
            idx = node.idx
            for item in node.state:
                if item.IsReduceItem:
                    prod = item.production
                    if prod.Left == G.startSymbol:
                        LR1Parser._register(self.action, (idx, G.EOF), (ShiftReduceParser.OK, None))
                    else:
                        for lookahead in item.lookaheads:
                            LR1Parser._register(self.action, (idx, lookahead), (ShiftReduceParser.REDUCE, prod)
                else:
                    next_symbol = item.NextSymbol
                    if next_symbol.IsTerminal:
                        LR1Parser._register(self.action, (idx, next_symbol), (ShiftReduceParser.SHIFT, node[nex
                    else:
                        LR1Parser._register(self.goto, (idx, next_symbol), node[next_symbol.Name][0].idx)
                pass

    @staticmethod
    def _register(table, key, value):
        assert key not in table or table[key] == value, 'Shift-Reduce or Reduce-Reduce conflict!!!'
        table[key] = value
```

Análisis semántico

En esta parte se verificarán los posibles errores semánticos que pueda tener el lenguaje, esto se realiza a través de type_collector, type_builder y type_checker. Los recorridos semánticos por el AST se realizan utilizando el visitor que se encuentra en la carpeta cmp, con este como decorador en los métodos visit que tenemos por cada nodo se van visitando todos los nodos del árbol. Cuando se

encuentra un error semántico lo devolvemos y paramos la ejecución.

Type_collector

Este módulo guardará en un contexto todos los tipos que pertenezcan al lenguaje, además de los métodos predefinidos que tiene el mismo.

Aquí dejamos algunos ejemplos de los tipos y funciones predefinidas.

In [ ]:
```
import cmp.visitor as visitor
from Compilacion.astree.AST_Nodes import ast_nodes as nodes

@visitor.when(nodes.ProgramNode)
def visit(self, node):

    species_type = self.context.create_type('Species')
    land_type = self.context.create_type('Land')
    society_type = self.context.create_type('Society')
    string_type = self.context.create_type('String')
    boolean_type = self.context.create_type('Boolean')
    number_type = self.context.create_type('Number')


    land_type.define_method('_deleteInfluence', ['entity_1_name', 'characteristic_1_name', 'entity_2_name', 'ch
    land_type.define_method('_changeCharacteristic', ['name', 'value', 'liminf', 'limsup', 'mutab', 'dist'], [s
    society_type.define_method('_changeCharacteristic', ['name', 'value', 'liminf', 'limsup', 'mutab', 'dist'],
    species_type.define_method('_changeCharacteristic', ['name', 'value', 'liminf', 'limsup', 'mutab', 'dist'],
    land_type.define_method('_deleteCharacteristic', ['name'], [string_type], boolean_type)
    society_type.define_method('_deleteCharacteristic', ['name'], [string_type], boolean_type)
    species_type.define_method('_deleteCharacteristic', ['name'], [string_type], boolean_type)


    return
```

Type_builder

En este recorrido tomaremos todas las declaraciones de variables, las declaraciones de entidades y las declaraciones de funciones, que se irán guardando en el contexto.

Dejamos de ejemplo el recorrido al nodo DeclarationEntity.

In [ ]:
```
import cmp.visitor as visitor
from Compilacion.astree.AST_Nodes import ast_nodes as nodes
from cmp.semantic import SemanticError

@visitor.when(nodes.DeclarationEntity)
def visit(self,node):
    if(self.error):
        return
    try:
        attrType = self.context.get_type(node.entity)
        self.current_type.define_attribute(node.var, attrType)
    except SemanticError as se:
        self.errors.append(se.text)
        self.error = True
        return se

    return
```

Type_checker

En esta parte se verifica que las variables, funciones y entidades instanciadas ya se encuentren declaradas en su scope o en alguno superior, también se verifica que los tipos sean correctos, o sea, que estén bien declarados o instanciados, por ejemplo, Boolean a = 5 da error semántico, pues el tipo declarado no coincide con el asignado, este y otros errores son los que revisa el type_checker.

El siguiente ejemplo muestra el chequeo del nodo AsignationVar.

In [ ]:
```
import cmp.visitor as visitor
from Compilacion.astree.AST_Nodes import ast_nodes as nodes
from cmp.semantic import SemanticError, TypeCompatible


WRONG_SIGNATURE = 'Method "%s" already defined in "%s" with a different signature.'
SELF_IS_READONLY = 'Variable "self" is read-only.'
LOCAL_ALREADY_DEFINED = 'Variable "%s" is already defined in method "%s".'
INCOMPATIBLE_TYPES = 'Cannot convert "%s" into "%s".'
VARIABLE_NOT_DEFINED = 'Variable "%s" is not defined in "%s".'
INVALID_OPERATION = 'Operation "%s" is not defined between "%s" and "%s".'
INVALID_RETURN = 'Return value is not an asignation".'
INVALID_PARAMS = 'Invalid params'
```

```
INVALID_NAME = "Inavlid name %s"

@visitor.when(nodes.AsignationVar)
def visit(self, node, scope):
    if(self.error):
        return
    var = scope.find_variable(node.var)

    type_expr = self.visit(node.arith_expr, scope.create_child())
    if(self.error):
        return

    if var is None:
        self.errors.append(VARIABLE_NOT_DEFINED % (node.var, self.current_method.name))
        self.error = True
        return TypeCompatible()

    elif not type_expr.conforms_to(var.type):
        self.errors.append(INCOMPATIBLE_TYPES % (type_expr.name, var.type.name))
        self.error = True
        return TypeCompatible()


    return type_expr
```

Transpilado

Para transpilar nuestro lenguaje a python hacemos otro recorrido por el AST, en este caso iremos formando un string, donde cada nodo del AST aportará un pedacito a este, cada nodo sabe como transpilarse a python (sabe convertirse en un string que será el código en python de ese nodo), por tanto, al finalizar este recorrido tendremos en un string todo el código que nos entran pero transfomado a python.

Aquí dejamos un ejemplo de cómo se transpila a python, mostramos el nodo de declarar una variable.

In [ ]:
```
@visitor.when(nodes.DeclarationVar)
def visit(self, node, scope, ident):
    self.declared_var.append(node.var)

    var_type = self.context.get_type(node.type)

    decl = "z" + node.var + " = " + self.visit(node.arith_expr, scope.create_child(), ident+1)

    scope.define_variable(node.var, var_type)

    return decl
```