*~~#-\*- coding: utf-8 -\*-  
#-------------------------------------------------------------------------  
# lexer.py  
# Analisador léxico para a linguagem Tinnny++  
# Autores: Sávio Camacam  
#-------------------------------------------------------------------------~~***~~from~~** ~~locale~~ **~~import~~** ~~str~~**~~from~~** ~~ply~~ **~~import~~** ~~yacc~~**~~from~~** ~~lexer~~ **~~import~~** ~~Lexer~~**~~class~~** ~~Scope:~~ **~~def~~** ~~\_\_init\_\_(self, name):  
 self.name = name  
 self.subscopes = []  
 self.entries = []~~ **~~def~~** ~~get\_entries(self):~~ **~~return~~** ~~self.entries~~**~~class~~** ~~Entry:~~ **~~def~~** ~~\_\_init\_\_(self, name, valor):  
 self.name = name  
 self.valor = valor~~**~~class~~** ~~Tree:~~ **~~def~~** ~~\_\_init\_\_(self, type\_node, child=[], value=~~**~~''~~**~~):  
 self.type = type\_node  
 self.child = child  
 self.value = value~~ **~~def~~** ~~\_\_str\_\_(self):~~ **~~return~~** ~~self.type~~**~~class~~** ~~Parser:~~ **~~def~~** ~~\_\_init\_\_(self, code):  
 lex = Lexer()  
 self.scopes = []  
 s = Scope(~~**~~'global'~~**~~)  
 self.scopes.append(s)  
 self.tokens = lex.tokens  
 self.precendence = (  
 (~~**~~'left'~~**~~,~~ **~~'SENAO'~~**~~),  
 (~~**~~'left'~~**~~,~~ **~~'EQU'~~**~~,~~ **~~'NEQ'~~**~~,~~ **~~'GEQ'~~**~~,~~ **~~'GRT'~~**~~,~~ **~~'LEQ'~~**~~,~~ **~~'LET'~~**~~),  
 (~~**~~'left'~~**~~,~~ **~~'ADD'~~**~~,~~ **~~'SUB'~~**~~),  
 (~~**~~'left'~~**~~,~~ **~~'TIMES'~~**~~,~~ **~~'DIV'~~**~~),  
 )  
 parser = yacc.yacc(debug=~~**~~False~~**~~, module=self, optimize=~~**~~False~~**~~)  
 self.ast = parser.parse(code)  
 print(~~**~~""~~**~~)~~  
  
**~~def~~** ~~p\_programa(self, p):~~ *~~'programa : lista\_declaracoes'~~* ~~p[0] = Tree(~~**~~'program'~~**~~, [p[1]])~~  
  
**~~def~~** ~~p\_lista\_declaracoes(self, p):~~ *~~'''  
 lista\_declaracoes : lista\_declaracoes declaracao  
 | declaracao  
 | error  
 '''~~* **~~if~~** ~~p.slice[1].type ==~~ **~~"error"~~**~~:  
 print(~~**~~"Erro de declaracao na linha "~~** ~~+ str(p.slice[1].lineno))~~ **~~elif~~** ~~len(p) == 3:  
 p[0] = Tree(~~**~~'lista-declaracoes'~~**~~, [p[1],p[2]])~~ **~~elif~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'lista-declaracoes'~~**~~, [p[1]])~~  
  
**~~def~~** ~~p\_declaracao(self, p):~~ *~~'''  
 declaracao : declaracao\_variaveis  
 | inicializacao\_variaveis  
 | declaracao\_funcao  
 | error  
 '''~~* **~~if~~** ~~p.slice[1].type ==~~ **~~"error"~~**~~:  
 print(~~**~~"Erro de declaracao na linha "~~** ~~+ str(p.slice[1].lineno))~~ **~~else~~**~~:  
 p[0] = Tree(~~**~~'declaracao'~~**~~, [p[1]])~~  
  
**~~def~~** ~~p\_declaracao\_variaveis(self, p):~~ *~~'declaracao\_variaveis : tipo COLON lista\_variaveis'~~* ~~p[0] = Tree(~~**~~'declaracao-variaveis'~~**~~, [p[1], p[3]])~~ **~~def~~** ~~p\_declaracao\_variaveis\_error(self, p):~~ *~~'declaracao\_variaveis : tipo COLON error'~~* ~~print(~~**~~"Erro na declaração de variáveis na linha "~~** ~~+ str(p.slice[3].lineno))~~  
  
**~~def~~** ~~p\_inicializacao\_variaveis(self, p):~~ *~~'inicializacao\_variaveis : atribuicao'~~* ~~p[0] = Tree(~~**~~'inicializacao-variaveis'~~**~~, [p[1]])~~  
  
**~~def~~** ~~p\_declaracao\_funcao(self, p):~~ *~~'''  
 declaracao\_funcao : tipo cabecalho  
 | cabecalho  
 '''~~* **~~if~~** ~~len(p) == 3:  
 p[0] = Tree(~~**~~'declaracao-funcao'~~**~~, [p[1], p[2]])~~ **~~elif~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'declaracao-funcao'~~**~~, [p[1]])~~ **~~def~~** ~~p\_declaracao\_funcao\_error(self, p):~~ *~~'''  
 declaracao\_funcao : tipo cabecalho error  
 | cabecalho error  
 '''~~* ~~print(~~**~~"Erro de declaracao na linha "~~** ~~+ str(p.slice[1].lineno))~~  
**~~def~~** ~~p\_tipo(self, p):~~ *~~'''tipo : INTEIRO   
 | FLUTUANTE'''~~* ~~p[0] = Tree(~~**~~'tipo'~~**~~, [], p[1])~~  
  
**~~def~~** ~~p\_lista\_variaveis(self, p):~~ *~~'''  
 lista\_variaveis : lista\_variaveis COM var  
 | var  
 '''~~* **~~if~~** ~~len(p) == 4:  
 p[0] = Tree(~~**~~'lista-variaveis'~~**~~, [p[1], p[3]])~~ **~~elif~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'lista-variaveis'~~**~~, [p[1]])~~ **~~def~~** ~~p\_lista\_variaveis\_error(self, p):~~ *~~'lista\_variaveis : lista\_variaveis COM error'~~* ~~print(~~**~~"Erro na declaração de variaveis na linha "~~** ~~+ str(p.slice[2].lineno))~~  
  
**~~def~~** ~~p\_atribuicao(self, p):~~ *~~'''  
 atribuicao : var simbolo\_atribuicao expressao  
 | condicional  
 | NOT condicional  
 '''~~* **~~if~~** ~~len(p) == 4:  
 p[0] = Tree(~~**~~'atribuicao'~~**~~, [p[1], p[2], p[3]])~~ **~~elif~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'atribuicao'~~**~~, [p[1]])~~ **~~elif~~** ~~len(p) == 3:  
 p[0] = Tree(~~**~~'atribuicao'~~**~~, [p[2]], p[1])~~  
  
  
**~~def~~** ~~p\_condicional(self, p):~~ *~~'''  
 condicional : expressao\_simples operador\_relacional expressao\_aditiva  
 | LPAR condicional RPAR  
 | condicional simbolo\_condicional condicional  
 | condicional simbolo\_condicional error  
 | LPAR error RPAR  
 | error simbolo\_condicional condicional  
 '''~~* **~~if~~** ~~p.slice[1].type ==~~ **~~"expressao\_simples"~~**~~:  
 p[0] = Tree(~~**~~'operador\_relacional'~~**~~, [p[1], p[2], p[3]])~~ **~~elif~~** ~~p.slice[2].type ==~~ **~~"condicional"~~**~~:  
 p[0] = Tree(~~**~~'condicional'~~**~~, [p[2]])~~ **~~elif~~** ~~p.slice[1].type ==~~ **~~"condicional"~~**~~:  
 p[0] = Tree(~~**~~'simbolo\_condicional'~~**~~, [p[1], p[2], p[3]])~~  
  
**~~def~~** ~~p\_simbolo\_condicional(self, p):~~ *~~'''  
 simbolo\_condicional : OR  
 | AND  
 '''~~* ~~p[0] = Tree(~~**~~'simbolo-condicional'~~**~~, [], p[1])~~ **~~def~~** ~~p\_simbolo\_atribuicao(self, p):~~ *~~'simbolo\_atribuicao : ASS'~~* ~~p[0] = Tree(~~**~~'simbolo-atribuicao'~~**~~, [], p[1])~~  
  
**~~def~~** ~~p\_atribuicao\_error(self, p):~~ *~~'atribuicao : var simbolo\_atribuicao error'~~* ~~print(~~**~~"Erro de atribuição na linha "~~** ~~+ str(p.slice[2].lineno))~~  
**~~def~~** ~~p\_cabecalho(self, p):~~ *~~'cabecalho : ID LPAR lista\_parametros RPAR corpo FIM'~~* ~~p[0] = Tree(~~**~~'cabecalho'~~**~~, [p[3], p[5]], p[1])~~ **~~def~~** ~~p\_cabecalho\_error(self, p):~~ *~~'cabecalho : ID LPAR lista\_parametros RPAR corpo error'~~* ~~print(~~**~~"Erro sintático. Função está faltando FIM na linha "~~** ~~+ str(p.slice[6].lineno))~~  
  
**~~def~~** ~~p\_corpo(self, p):~~ *~~'''  
 corpo : corpo acao  
 | empty  
 '''~~* **~~if~~** ~~len(p) == 3:  
 p[0] = Tree(~~**~~'corpo'~~**~~, [p[1], p[2]])~~ **~~'''elif len(p) == 2:  
 p[0] = Tree('corpo', [p[1]])'''  
  
 def~~** ~~p\_acao(self, p):~~ *~~'''  
 acao : expressao  
 | declaracao\_variaveis  
 | se  
 | repita  
 | leia  
 | escreve  
 | retorna  
 '''~~* ~~p[0] = Tree(~~**~~'acao'~~**~~, [p[1]])~~  
  
**~~def~~** ~~p\_se(self, p):~~ *~~'''  
 se : SE expressao ENTAO corpo FIM  
 | SE expressao ENTAO corpo SENAO corpo FIM  
 '''~~* **~~if~~** ~~len(p) == 6:  
 p[0] = Tree(~~**~~'se'~~**~~, [p[2], p[4]])~~ **~~elif~~** ~~len(p) == 8:  
 p[0] = Tree(~~**~~'se'~~**~~, [p[2], p[4], p[6]])~~ **~~def~~** ~~p\_se\_error(self, p):~~ *~~'''  
 se : SE expressao error corpo FIM  
 | error SENAO corpo FIM  
 '''~~* **~~if~~** ~~len(p) == 6:  
 print(~~**~~"Erro sintático na linha "~~** ~~+ str(p.slice[3].lineno) +~~ **~~". Esperado um ENTÃO"~~**~~)~~ **~~elif~~** ~~len(p) == 5:  
 print(~~**~~"Erro sintático na linha "~~** ~~+ str(p.slice[1].lineno) +~~ **~~". Condicional SENÃO mal-formado"~~**~~)~~  
  
**~~def~~** ~~p\_repita(self, p):~~ *~~'repita : REPITA corpo ATE expressao'~~* ~~p[0] = Tree(~~**~~'repita'~~**~~, [p[2], p[4]])~~ **~~def~~** ~~p\_repita\_error(self, p):~~ *~~'repita : REPITA corpo error'~~* ~~print(~~**~~"Erro. Esperado expressão de termino"~~**~~)~~  
  
**~~def~~** ~~p\_leia(self, p):~~ *~~'leia : LEIA LPAR expressao RPAR'~~* ~~p[0] = Tree(~~**~~'leia'~~**~~, [p[3]], p[1])~~ **~~def~~** ~~p\_escreve(self, p):~~ *~~'escreve : ESCREVE LPAR expressao RPAR'~~* ~~p[0] = Tree(~~**~~'escreve'~~**~~, [p[3]], p[1])~~ **~~def~~** ~~p\_retorna(self, p):~~ *~~'retorna : RETORNA LPAR expressao RPAR'~~* ~~p[0] = Tree(~~**~~'retorna'~~**~~, [p[3]], p[1])~~  
  
**~~def~~** ~~p\_var(self, p):~~ *~~'''  
 var : ID  
 | ID indice  
 | ID lista\_dimensions  
 '''~~* **~~if~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'var'~~**~~, [], p[1])~~ **~~elif~~** ~~len(p) == 3:  
 p[0] = Tree(~~**~~'var'~~**~~, [p[2]], p[1])~~  
  
**~~def~~** ~~p\_expressao(self, p):~~ *~~'''  
 expressao : expressao\_simples  
 | atribuicao  
 '''~~* ~~p[0] = Tree(~~**~~'expressao'~~**~~, [p[1]])~~  
**~~def~~** ~~p\_lista\_parametros(self, p):~~ *~~'''  
 lista\_parametros : lista\_parametros COM parametro  
 | parametro  
 | empty  
 '''~~* **~~if~~** ~~len(p) == 4:  
 p[0] = Tree(~~**~~'lista-parametros'~~**~~, [p[1], p[3]])~~ **~~elif~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'lista-parametros'~~**~~, [p[1]])~~ **~~def~~** ~~p\_indice(self, p):~~ *~~'''  
 indice : indice LBR expressao RBR  
 | LBR expressao RBR  
 '''~~* **~~if~~** ~~len(p) == 5:  
 p[0] = Tree(~~**~~'indice'~~**~~, [p[1], p[3]])~~ **~~elif~~** ~~len(p) == 4:  
 p[0] = Tree(~~**~~'indice'~~**~~, [p[2]])~~ **~~def~~** ~~p\_indice\_error(self, p):~~ *~~'''  
 indice : indice LBR error RBR  
 | LBR error RBR  
 | error RBR  
 | LBR error  
 '''~~* ~~print(~~**~~"Erro sintático no índice na linha "~~** ~~+ str(p.slice[3].lineno))~~  
**~~def~~** ~~p\_expressao\_simples(self, p):~~ *~~'''  
 expressao\_simples : expressao\_aditiva  
 | expressao\_simples operador\_relacional expressao\_aditiva  
 '''~~* **~~if~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'expressao-simples'~~**~~, [p[1]])~~ **~~elif~~** ~~len(p) == 4:  
 p[0] = Tree(~~**~~'expressao-simples'~~**~~, [p[1], p[2], p[3]])~~  
**~~def~~** ~~p\_parametro(self, p):~~ *~~'''  
 parametro : tipo COLON var  
 | parametro  
 '''~~* **~~if~~** ~~p.slice[1].type ==~~ **~~"tipo"~~**~~:  
 p[0] = Tree(~~**~~'parametro'~~**~~, [p[1], p[3]])~~ **~~elif~~** ~~p.slice[1].type ==~~ **~~'parametro'~~**~~:  
 p[0] = Tree(~~**~~'parametro'~~**~~, [p[1]])~~  
  
**~~def~~** ~~p\_lista\_dimensions(self, p):~~ *~~'''  
 lista\_dimensions : dimension  
 | lista\_dimensions dimension  
 '''~~* **~~if~~** ~~len(p) == 3:  
 p[0] = Tree(~~**~~'lista-dimensions'~~**~~, [p[1], p[2]])~~ **~~elif~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'lista-dimensions'~~**~~, [p[1]])~~  
  
**~~def~~** ~~p\_dimension(self, p):~~ *~~'dimension : LBR RBR'~~* ~~p[0] = Tree(~~**~~'dimension'~~**~~, [],~~ **~~'dim'~~**~~)~~  
  
  
**~~def~~** ~~p\_operador\_relacional(self, p):~~ *~~'''  
 operador\_relacional : LET  
 | GRT  
 | EQU  
 | NEQ  
 | LEQ  
 | GEQ  
 '''~~* ~~p[0] = Tree(~~**~~'operador-relacional'~~**~~, [], p[1])~~  
  
**~~def~~** ~~p\_expressao\_aditiva(self, p):~~ *~~'''  
 expressao\_aditiva : expressao\_multiplicativa  
 | expressao\_aditiva operador\_soma expressao\_multiplicativa  
 '''~~* **~~if~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'expressao-aditiva'~~**~~, [p[1]])~~ **~~elif~~** ~~len(p) == 4:  
 p[0] = Tree(~~**~~'expressao-aditiva'~~**~~, [p[1], p[2], p[3]])~~  
  
**~~def~~** ~~p\_expressao\_multiplicativa(self, p):~~ *~~'''  
 expressao\_multiplicativa : expressao\_unaria  
 | expressao\_multiplicativa operador\_multiplicacao expressao\_unaria  
 '''~~* **~~if~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'expressao-multiplicativa'~~**~~, [p[1]])~~ **~~elif~~** ~~len(p) == 4:  
 p[0] = Tree(~~**~~'expressao-multiplicativa'~~**~~, [p[1], p[2], p[3]])~~  
  
**~~def~~** ~~p\_operador\_soma(self, p):~~ *~~'''  
 operador\_soma : ADD  
 | SUB  
 '''~~* ~~p[0] = Tree(~~**~~'operador-soma'~~**~~, [], p[1])~~  
  
**~~def~~** ~~p\_operador\_multiplicacao(self, p):~~ *~~'''  
 operador\_multiplicacao : TIMES  
 | DIV  
 '''~~* ~~p[0] = Tree(~~**~~'operador-multiplicacao'~~**~~, [], p[1])~~  
  
**~~def~~** ~~p\_expressao\_unaria(self, p):~~ *~~'''  
 expressao\_unaria : fator  
 | operador\_soma fator  
 '''~~* **~~if~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'expressao-unaria'~~**~~, [p[1]])~~ **~~elif~~** ~~len(p) == 3:  
 p[0] = Tree(~~**~~'expressao-unaria'~~**~~, [p[1], p[2]])~~  
  
**~~def~~** ~~p\_fator(self, p):~~ *~~'''  
 fator : LPAR expressao RPAR  
 | var  
 | chamada\_funcao  
 | numero  
 '''~~* **~~if~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'fator'~~**~~, [p[1]])~~ **~~elif~~** ~~len(p) == 4:  
 p[0] = Tree(~~**~~'fator'~~**~~, [p[2]])~~  
**~~def~~** ~~p\_numero(self, p):~~ *~~'''  
 numero : INTEIRO  
 | FLUTUANTE  
 '''~~* ~~p[0] = Tree(~~**~~'numero'~~**~~, [], p[1])~~  
  
**~~def~~** ~~p\_chamada\_funcao(self, p):~~ *~~'chamada\_funcao : ID LPAR lista\_argumentos RPAR'~~* ~~p[0] = Tree(~~**~~'chamada-funcao'~~**~~, [p[3]], p[1])~~  
**~~def~~** ~~p\_lista\_argumentos(self, p):~~ *~~'''  
 lista\_argumentos : lista\_argumentos COM expressao  
 | expressao  
 | empty  
 '''~~* **~~if~~** ~~len(p) == 2:  
 p[0] = Tree(~~**~~'lista-argumentos'~~**~~, [p[1]])~~ **~~elif~~** ~~len(p) == 4:  
 p[0] = Tree(~~**~~'lista-argumentos'~~**~~, [p[1], p[3]])~~  
  
 **def** p\_empty(self, p):  
 *'empty :'* **pass  
  
 def** p\_error(self, p):  
 **if** p:  
 print(**"'%s', linha %d"** % (p.value, p.lineno))  
 **else**:  
 yacc.\_restart  
 print(**'Erro sintático: definições incompletas!'**)  
 exit(1)  
  
  
**def** generateTree(t):  
 **if** t **is not None**:  
 print(**'['**+ t.type + **' '** + t.value)  
  
 **for** node **in** t.child:  
 i = t.child.index(node)  
 generateTree(t.child[i])  
 print(**']'**)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 **from** sys **import** argv, exit  
 config = 1  
 **if** config:  
 f = open(argv[1], encoding=**'utf-8'**)  
 p = Parser(f.read())  
 generateTree(p.ast)  
  
 **else**:  
 **import** glob, os  
 path = **"C:/Users/savio/git/compiladores-march/testes"** os.chdir(path)  
  
 **for** file **in** glob.glob(**"\*.tpp"**):  
 print(file.title())  
 f = open(file, encoding=**'utf-8'**)  
 p = Parser(f.read())  
 generateTree(p.ast)