

Bruno Duarte Barreto Borges
Fabio Oliveira de Abreu
Erik Kazuo Sugawara

Relatório

Florianópolis, SC

2021/2022

Bruno Duarte Barreto Borges
Fabio Oliveira de Abreu
Erik Kazuo Sugawara

Relatório

Relatório sobre a Construção de um Compilador, feito pelos alunos: Bruno Duarte Barreto Borges, Fabio Oliveira de Abreu e Erik Kazuo Sugawara. Para obtenção da aprovação na disciplina Construção de Compiladores, ministrada pelo Prof. Dr. Alvaro Junio Pereira Franco.

Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
INE5426 - Construção de Compiladores

Orientador: Alvaro Junio Pereira Franco

Florianópolis, SC
2021/2022

Lista de abreviaturas e siglas

PLY	Python Lexx-Yacc
BNF	Bakus-Naur Form
Yacc	Yet Another Compiler-Compiler

Sumário

	Introdução	7
I	FERRAMENTA	9
1	PYTHON LEX-YACC	11
1.1	Sobre a ferramenta	11
1.2	Lex	11
II	ANALISADOR LÉXICO	15
2	METODOLOGIA	17
2.1	Extração dos tokens	17
2.2	Expressões Regulares	18
2.3	Analizador Léxico	21
3	DIAGRAMA DE TRANSIÇÃO	23
III	ANALISADOR SINTÁTICO	33
4	FERRAMENTA	35
4.1	Yacc	35
5	GRAMÁTICA	39
5.1	Gramática Original	42
5.2	Modificações na Gramática	43
5.3	Forma Convencional	45
5.3.1	Transformação para definição de gramática convencional	45
5.3.2	Fatoração da Gramática	47
5.4	Código em Python	51
IV	ANALISADOR SEMÂNTICO	67
5.5	Introdução	69
5.6	Árvore de Expressão (EXPA)	69
5.7	Inserção do tipo na tabela de símbolos	73

5.8	Verificação de Tipo	74
5.9	Verificação de Identificadores por Escopo	75
5.10	Comandos dentro do Escopo	76
5.11	Geração de Código Intermediário	77
5.12	Lazy Check	80
5.13	Exceções	81
5.14	Saída	81
5.15	Saída com Erro de Compilação	82
5.16	Árvore de Expressão (EXPA)	83
5.17	Tabela de Símbolos, Tipo e Escopo	85
	Conclusão	89

APÊNDICES 91

APÊNDICE A – SDD DE CONVCC-2021-1	93
---	----

APÊNDICE B – SDT DE CONVCC-2021-1	113
---	-----

Introdução

Um compilador é um programa de sistema que traduz uma linguagem de alto nível para um programa equivalente em código de máquina para um processador. Sendo assim, um compilador não produz diretamente o código de máquina, mas sim um programa semanticamente equivalente na linguagem simbólica (assembly), que é traduzido para o programa em linguagem de máquina através de montadores¹.

Este trabalho, em sua primeira fase, tem como objetivo criar um compilador através de um analisador léxico e sintático para uma linguagem (AL). Para facilitar os estudos na criação do compilador, foi disponibilizada a linguagem “LCC-2021-2” derivada da gramática “CC-2021-2” na forma BNF(Backus-Naur Form), inspirada na gramática X++ de Delamaro². Na sua segunda parte, iremos modificar a gramática de forma que ela seja coerente para o analisador sintático e o código analisado deve ser aceito por ele.

As atividades contidas neste relatório foram feitas com a linguagem Python em conjunto da ferramenta PLY (Python Lex-Yacc)³, uma implementação dos geradores de analisadores léxico e sintático (Lex-Yacc) para Python. Dessa forma, com o auxílio de ferramentas que permitem gerar analisadores, mostraremos o funcionamento do analisador léxico através de seu algoritmo e da sua aplicação com exemplos de programas em alto nível baseadas na gramática “CC-2021-2”.

¹ <<https://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node37.html>>

² <<http://conteudo.icmc.usp.br/pessoas/delamaro/SlidesCompiladores/CompiladoresFinal.pdf>>

³ <<https://www.dabeaz.com/ply/>>

Parte I

Ferramenta

1 Python Lex-Yacc

1.1 Sobre a ferramenta

O PLY é uma implementação pura em Python baseada nas ferramentas de construção de compiladores lex e yacc. Possui suporte para LARL(1) e mecanismos de validação, verificação de erros e diagnósticos. Sua primeira versão foi desenvolvida na Universidade de Chicago, para dar suporte à disciplina Introdução a Compiladores. A sua versão mais recente é o PLY-4.0, que requer o Python na versão 3.6 ou mais nova.

Ele consiste em dois módulos separados: o lex.py e o yacc.py. O módulo lex.py é utilizado para separar as entradas em texto do código em uma coleção de tokens baseadas em suas respectivas expressões regulares. O yacc.py é utilizado para reconhecer a sintaxe de uma linguagem baseada em uma gramática livre de contexto. Além disso, possui utilidades como verificação de erros, validação de gramática, suporte para produções vazias, tokens de erros e regras de precedência para resolver ambiguidades.

1.2 Lex

O Lex é utilizado para realizar a atribuição dos tokens com as entradas (*strings*). Os nomes dos tokens podem ser declarados de forma simples através de uma lista. Um token que é identificado se torna um objeto da classe LexToken.

```
import ply.lex as lex
import ply.yacc as yacc

# RESERVED WORDS
reserved = [
    'DEF',          # def
    'BREAK',        # break
    'FOR',           # for
    'IF',            # if
    'ELSE',          # else
    'INT',           # int
    'FLOAT',         # float
    'NEW',           # new
    'PRINT',         # print
    'READ',          # read
    'RETURN',        # return
    'STRING',        # string
]
```

Dada uma lista de tokens, podemos definir suas expressões regulares, que podem ser de forma simples ou complexa. Note que por definição da ferramenta, as expressões regulares são declaradas como uma variável que se inicia com “t_”, seguida pelo nome do token definido na lista. Segue um exemplo das expressões regulares para tokens simples, como: *Assign*, *Greater Than*, *Less Than*, *Equals*, *Less Equal*, *Greater Equal*, *Not Equal*, *Plus*, *Minus*, *Multiply*, *Divide* e *Rem*.

```
t_ASSIGN = r'\='  
t_GT = r'\>'  
t_LT = r'\<'  
t_EQ = r'\=='  
t_LE = r'\<='  
t_GE = r'\>='  
t_NEQ = r'\!='  
t_PLUS = r'\+ '  
t_MINUS = r'\- '  
t_MULTIPLY = r'\* '  
t_DIVIDE = r'\/'  
t_REM = r'\%'
```

Para tokens que requerem maior complexidade para serem identificados, podemos declará-los através de funções que também se iniciam pelos caracteres “t_” seguidos pelo nome do token definido. Note que também podemos realizar o tratamento dos valores obtidos acessando o atributo *value* do objeto *LexToken*.

```
def t_float_constant(t):  
    r'[+-]?\d+\.\d+([eE][+-]?\d+)?'  
    t.value = float(t.value)  
    return t  
  
def t_int_constant(t):  
    r'[+-]?\d+ '  
    t.value = int(t.value)  
    return t  
  
def t_string_constant(t):  
    r'"[^\\n\\r]*"'  
    return t  
  
def t_IDENT(t):  
    r'[a-zA-Z_][a-zA-Z_0-9]* '  
    return t
```

Com essa ferramenta também podemos fazer tratativas de leitura e identificação de erros. Ela também segue as definições de nomenclatura para tokens. Seguem alguns exemplos de tratamento de leitura para ignorar caracteres em branco, fim de linhas e caracteres ilegais.

```
t_ignore = r' ' # Ignore spaces between char.

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

def t_error(t):
    errors.append("Illegal char %s in line %d, column %d" % (t.value[0],
                                                             t.lexer.lineno, find_column(t))
                 )
    t.lexer.skip(1)
```

Para demonstrar o funcionamento do analisador léxico, executamos o seguinte código. A saída é um LexToken, onde os atributos correspondem respectivamente ao identificador do token, símbolo, linha e coluna.

```
# Test it out
data = '''
3 + 4 * 10
'''

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
    print(tok)
```

Saída :

```
LexToken(NUMBER,3,2,1)
LexToken(PLUS,'+',2,3)
LexToken(NUMBER,4,2,5)
LexToken(MULTIPLY,'*',2,7)
LexToken(NUMBER,10,2,10)
```


Parte II

Analizador Léxico

2 Metodologia

2.1 Extração dos tokens

Para obter os tokens, utilizamos a gramática “CC-2021-2” como base e dividimos cada um deles em um dos seguintes grupos: palavras reservadas, operadores, símbolos especiais, constantes e identificadores. Além disso, alteramos a gramática de modo que seja possível retornar valores com o token “*return*”.

```
# RESERVED WORDS
reserved = [
    'DEF',          # def
    'BREAK',        # break
    'FOR',           # for
    'IF',            # if
    'ELSE',          # else
    'INT',           # int
    'FLOAT',         # float
    'NEW',           # new
    'PRINT',         # print
    'READ',          # read
    'RETURN',        # return
    'STRING',        # string
]
```

```
# OPERATORS
operators = [
    'ASSIGN',        # =
    'GT',            # >
    'LT',            # <
    'EQ',            # ==
    'LE',            # <=
    'GE',            # >=
    'NEQ',           # !=
    'PLUS',          # +
    'MINUS',         # -
    'MULTIPLY',      # *
    'DIVIDE',        # /
    'REM',           # %
]
```

```
# SPECIAL SYMBOLS
```

```
special = [  
    'LPAREN',      # (  
    'RPAREN',      # )  
    'LBRACE',      # {  
    'RBRACE',      # }  
    'LBRACKET',    # [  
    'RBRACKET',    # ]  
    'SEMICOLON',   # ;  
    'COMMA',       # ,  
]
```

```
# CONSTANTS
```

```
constant = [  
    'int_constant',  
    'string_constant',  
    'float_constant',  
    'null_constant'  
]
```

```
# IDENTIFIERS
```

```
identifiers = [  
    'IDENT'  
]
```

```
tokens = reserved + operators + special + constant + identifiers
```

2.2 Expressões Regulares

Após a identificação dos tokens, criamos a expressão regular de cada um deles.

```
t_ignore = r' ' # Ignore spaces between char.
```

```
def t_DEF(t):  
    r'def'  
    return t
```

```
def t_BREAK(t):  
    r'break'  
    return t
```

```
def t_FOR(t):  
    r'for'  
    return t
```

```
def t_IF(t):
    r'if'
    return t

def t_ELSE(t):
    r'else'
    return t

def t_NEW(t):
    r'new'
    return t

def t_PRINT(t):
    r'print'
    return t

def t_READ(t):
    r'read'
    return t

def t_RETURN(t):
    r'return'
    return t

def t_STRING(t):
    r'string'
    return t

def t_INT(t):
    r'int'
    return t

def t_FLOAT(t):
    r'float'
    return t


def t_null_constant(t):
    r'null'
    return t

def t_float_constant(t):
    r'[+-]?[d+\.]\d+([eE][+-]?\d+)?'
    t.value = float(t.value)
    return t
```

```
def t_int_constant(t):
    r'[+-]?\d+'
    t.value = int(t.value)
    return t

def t_string_constant(t):
    r'"[^"\n\r]*"'
    return t

def t_IDENT(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    return t

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

errors = []

def t_error(t):
    errors.append("Illegal char %s in line %d, column %d" % (t.value[0],
                                                             t.lexer.lineno, find_column(t)))

    t.lexer.skip(1)
```

Além da criação das expressões regulares, também foi criado uma função que imprime a tabela de símbolos de forma elegante no terminal.

```
def print_table(lexer):
    pattern = "{:~25} | {:~60} | {:~7} | {:~7}"
    print("\033[4m" + pattern.format("TOKEN", "VALUE", "LINE", "COLUMN")
          + "\033[0m")

    while True:
        tok = lexer.token()
        if not tok:
            break
        print(pattern.format(tok.type, tok.value, tok.lineno,
                             find_column(tok)))

    for e in errors:
        print(e)
```

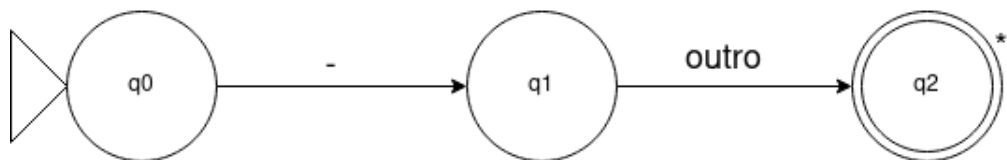
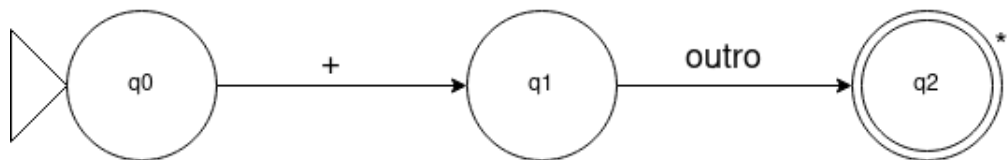
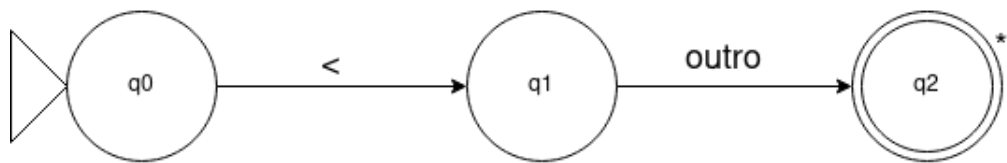
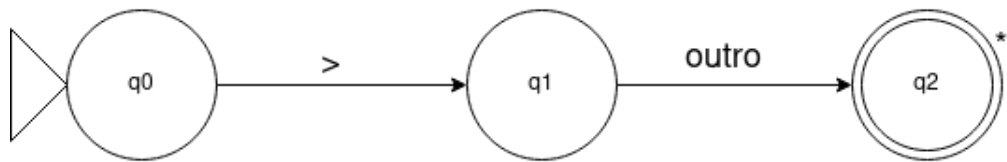
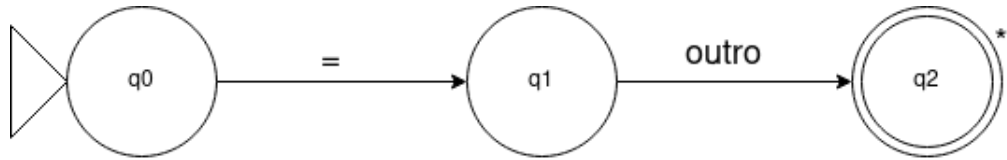
2.3 Analisador Léxico

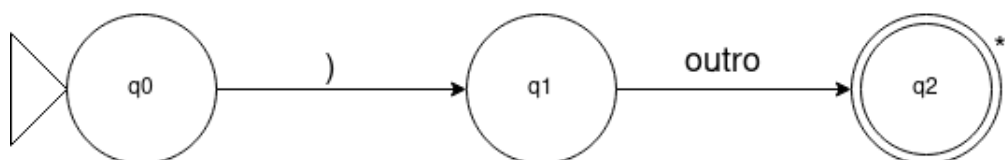
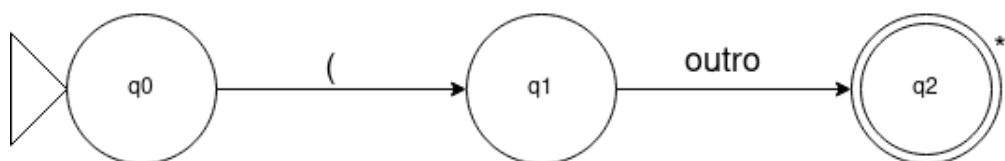
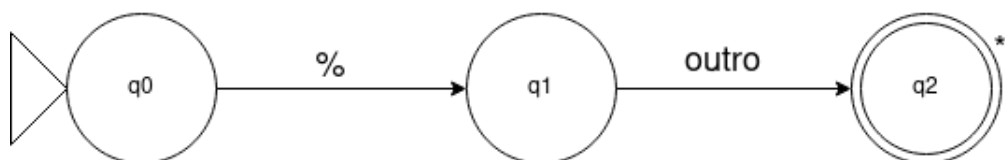
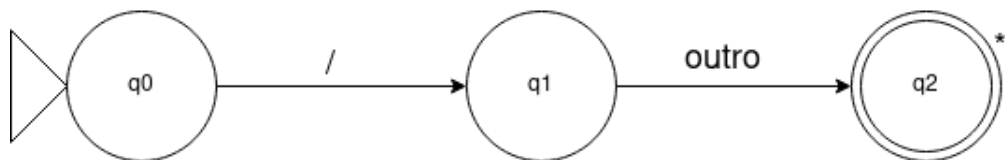
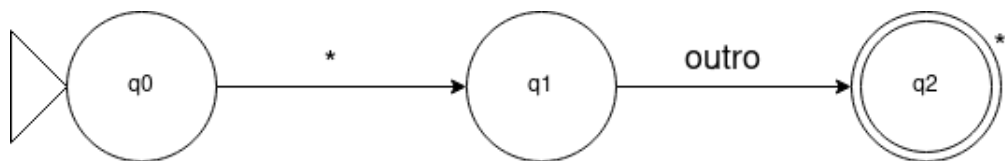
Com os tokens e expressões regulares definidos, executamos o analisador léxico e obtemos a sua tabela de símbolos formatada através da função “print_table”.

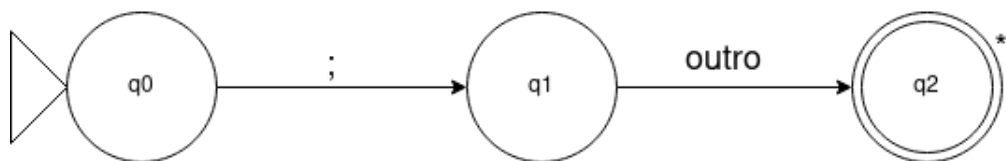
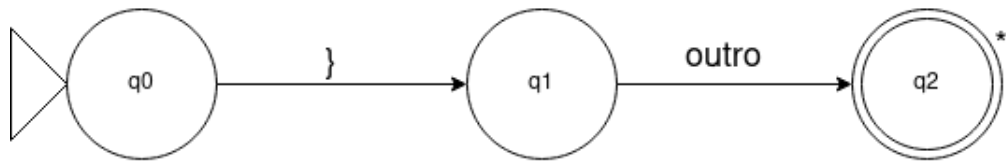
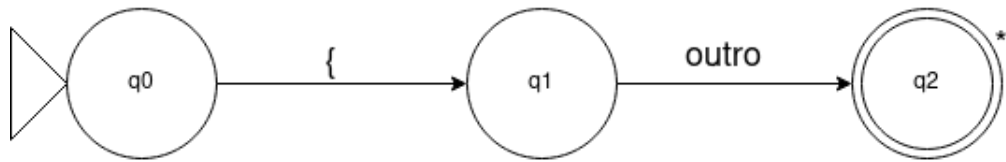
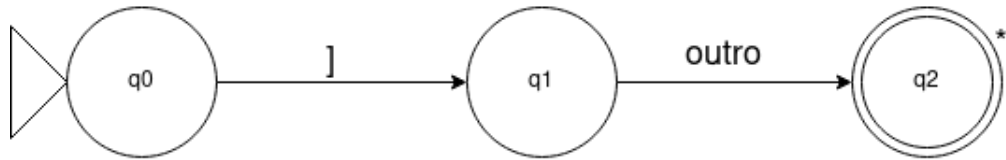
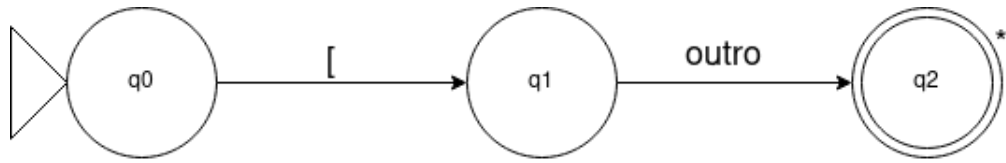
```
$ make run file='tmp/lil_example.lcc'
```

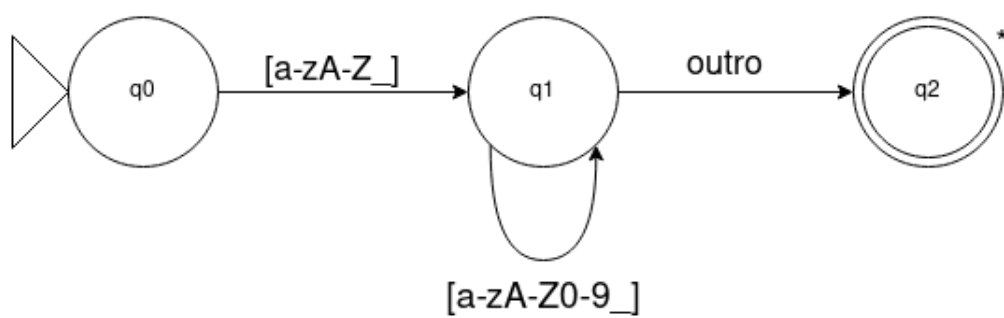
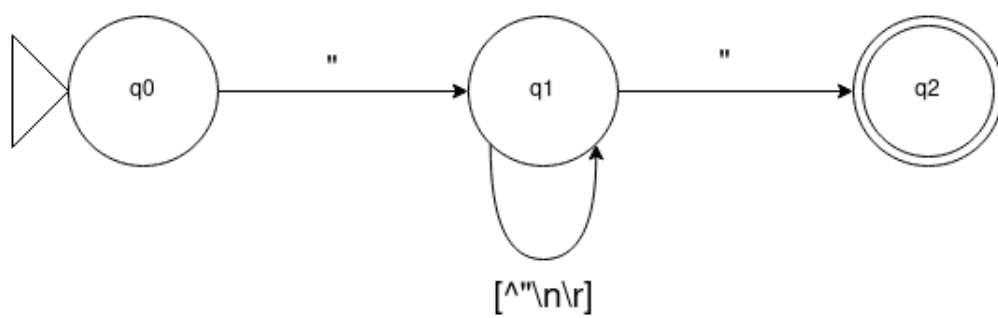
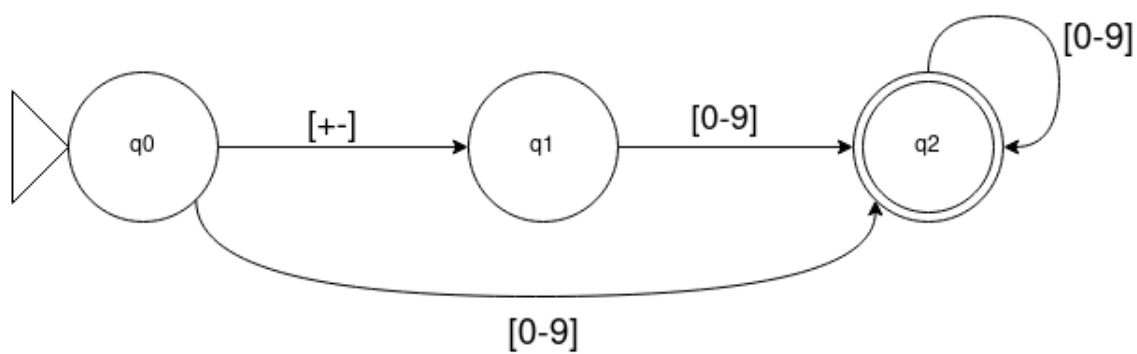
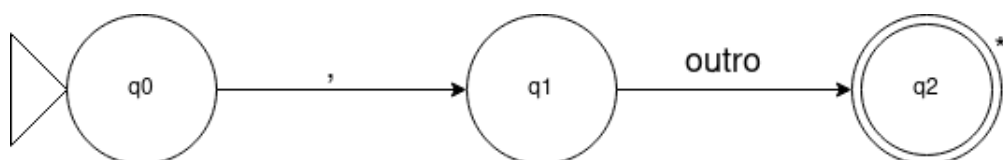
TOKEN	VALUE	LINE	COLUMN
DEF	def	1	1
IDENT	hello_world	1	5
LPAREN	(1	16
RPAREN)	1	17
LBRACE	{	1	19
PRINT	print	2	5
string_constant	"hello_world"	2	11
SEMICOLON	;	2	24
RBRACE	}	3	1
INT	int	5	1
IDENT	x	5	5
SEMICOLON	;	5	6
IDENT	x	6	1
ASSIGN	=	6	3
int_constant	10	6	5
SEMICOLON	;	6	7
IF	if	8	1
LPAREN	(8	4
IDENT	x	8	5
GT	>	8	7
int_constant	30	8	9
RPAREN)	8	11
LBRACE	{	8	13
IDENT	hello_world	9	5
LPAREN	(9	16
RPAREN)	9	17
SEMICOLON	;	9	18
RBRACE	}	10	1
ELSE	else	10	3
LBRACE	{	10	8
PRINT	print	11	5
string_constant	"erro"	11	11
SEMICOLON	;	11	17
RBRACE	}	12	1

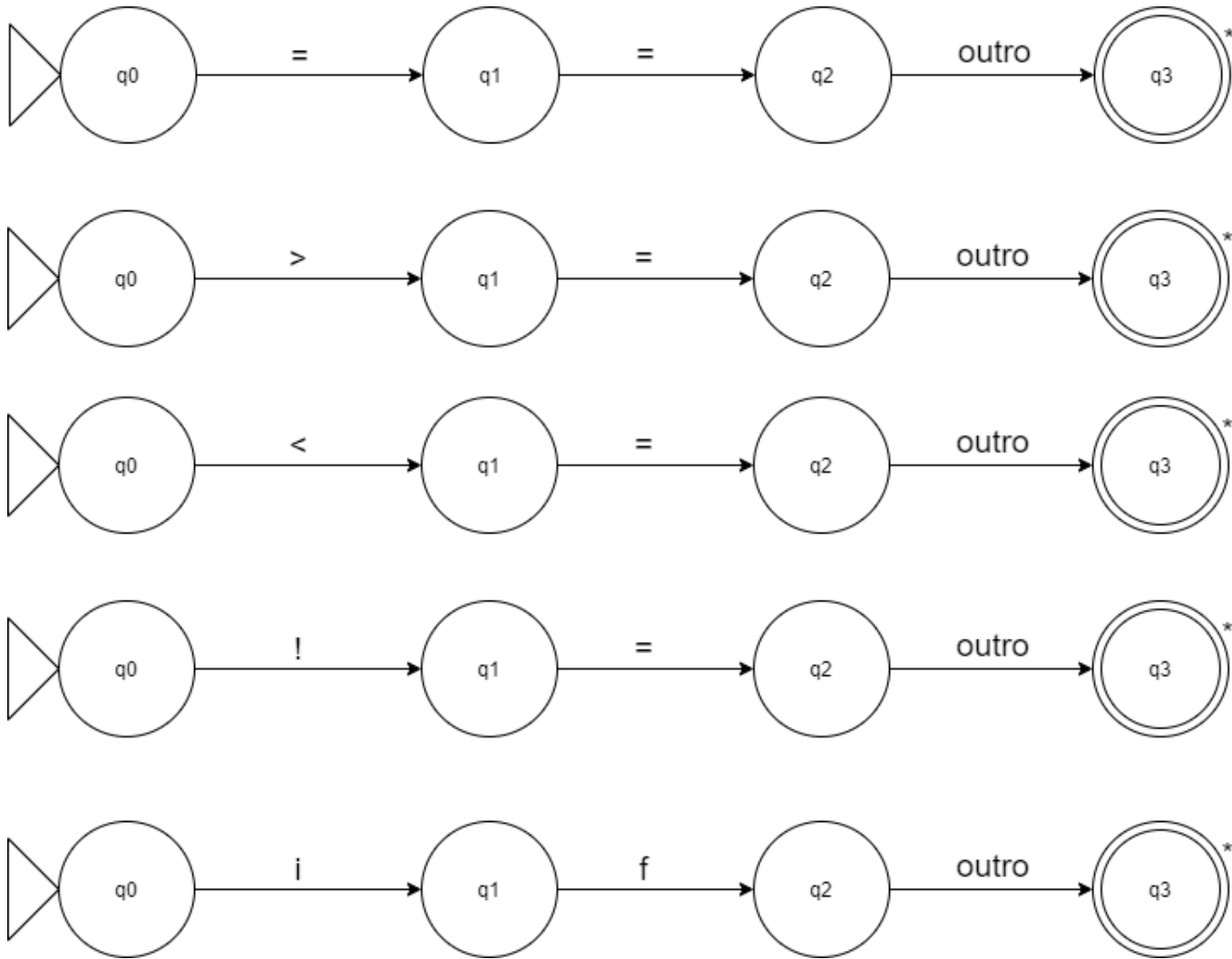
3 Diagrama de Transição

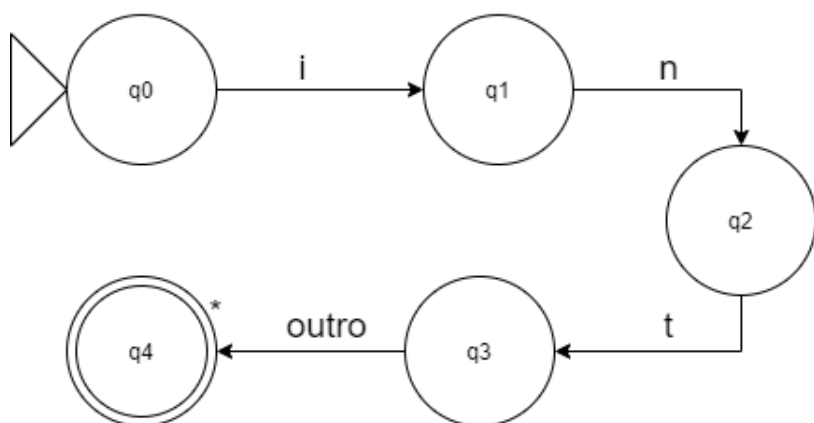
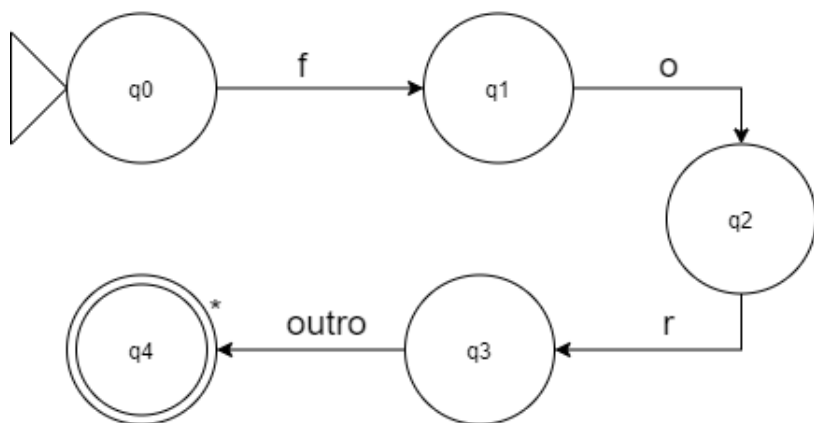
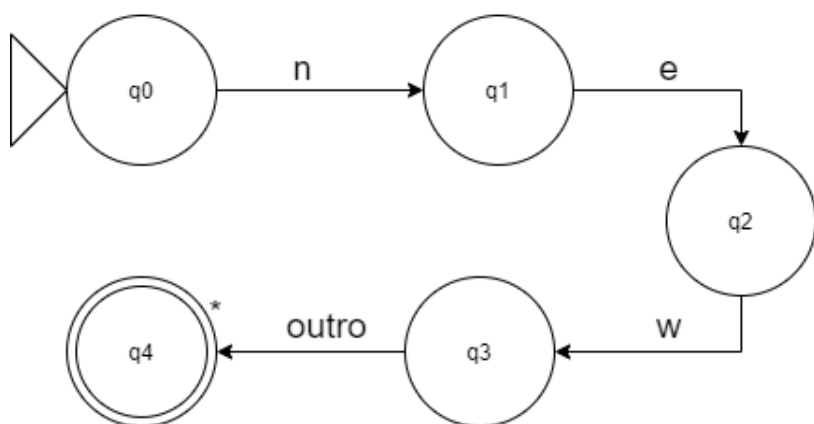
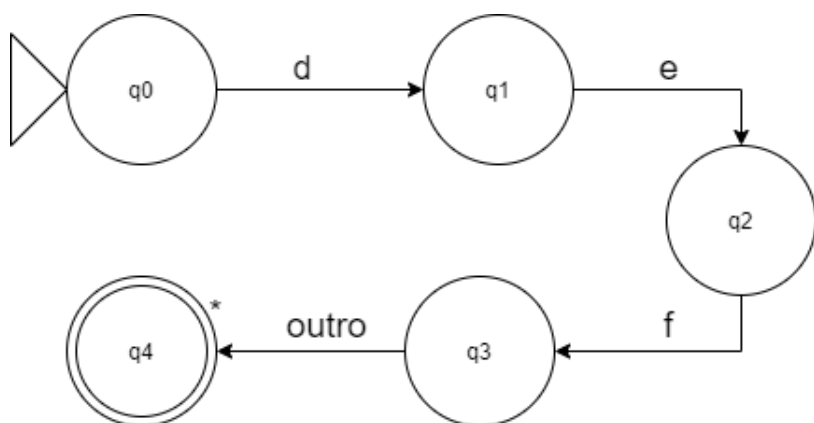


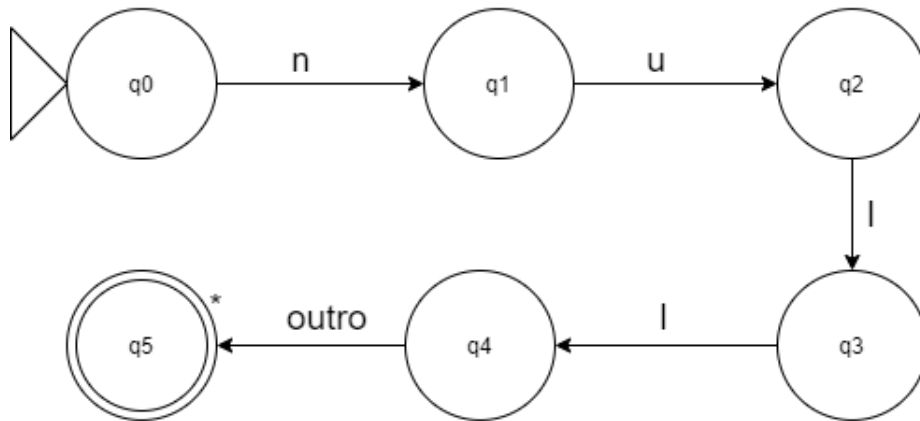
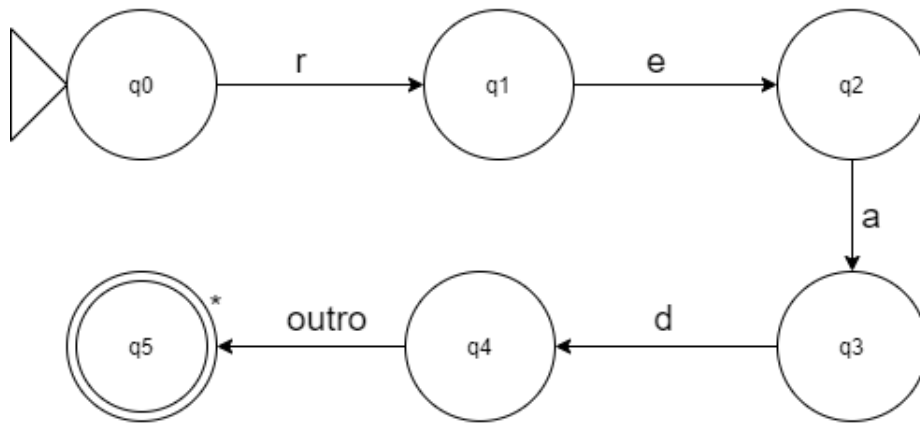
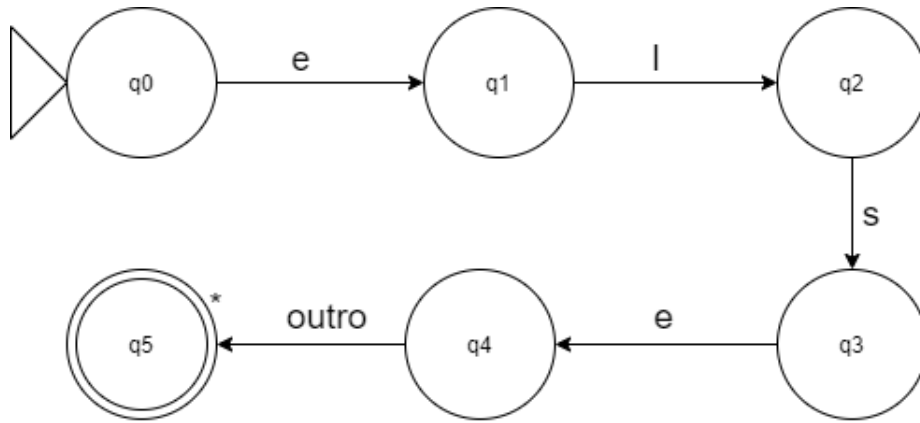


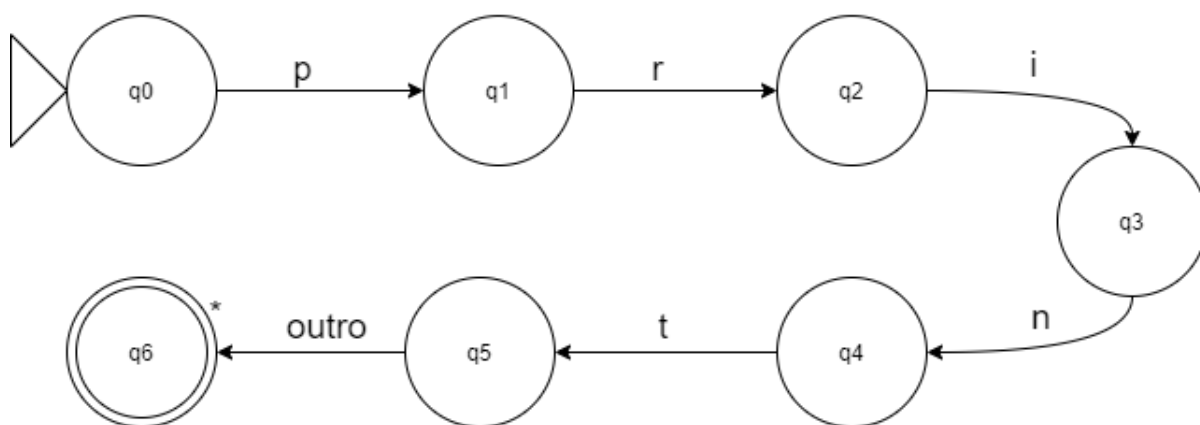
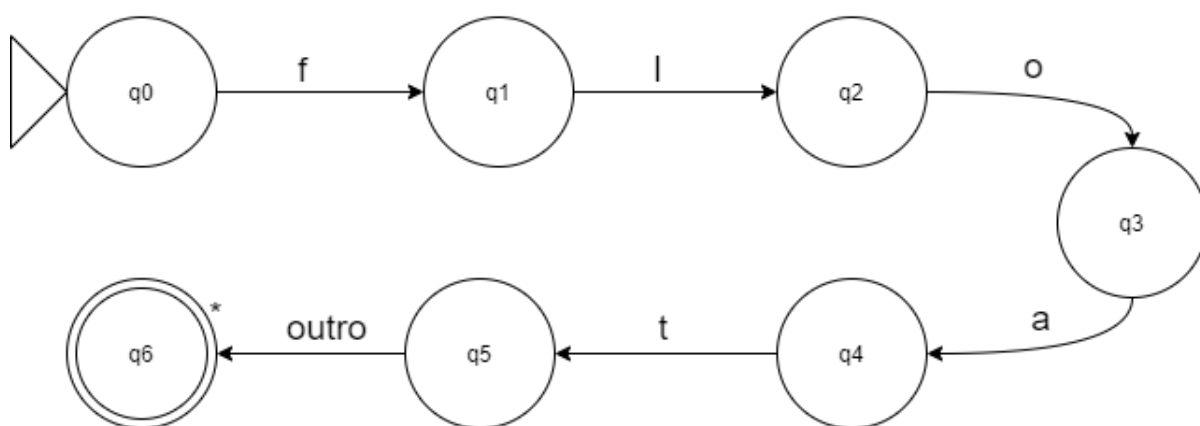
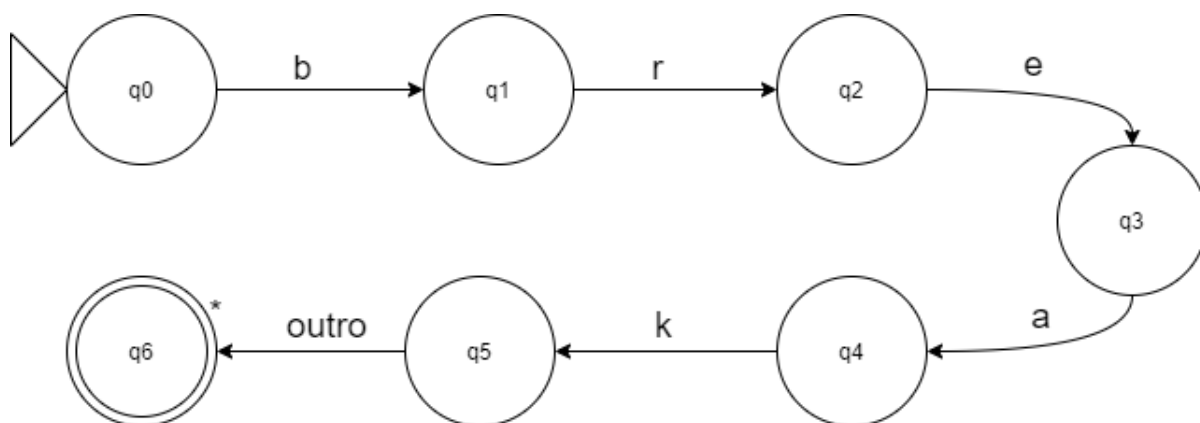


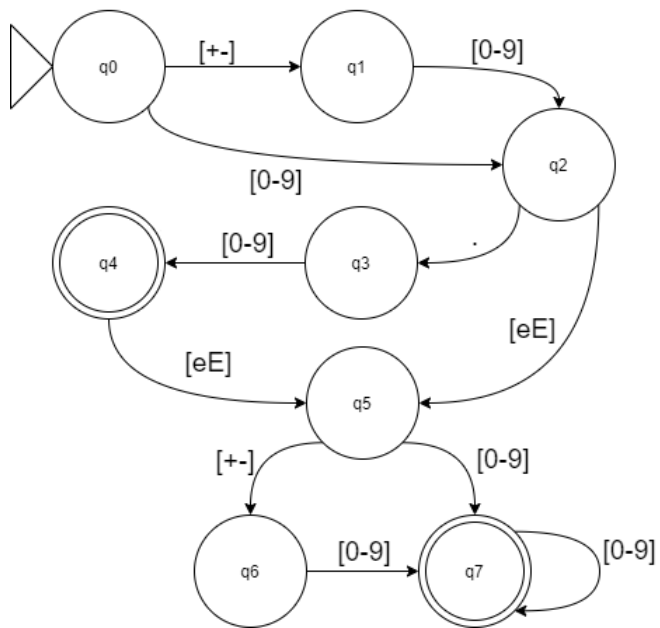
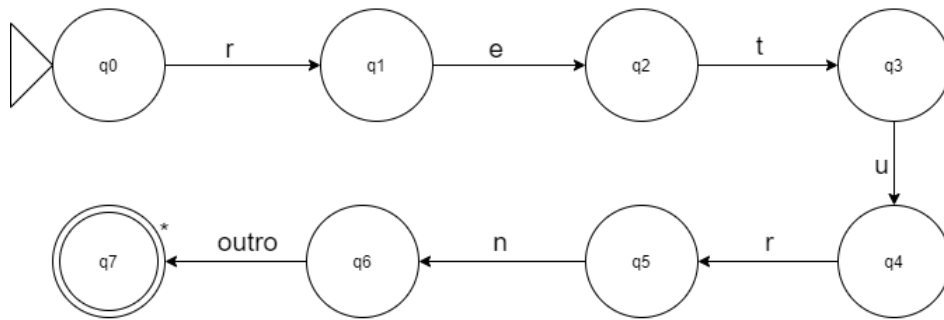
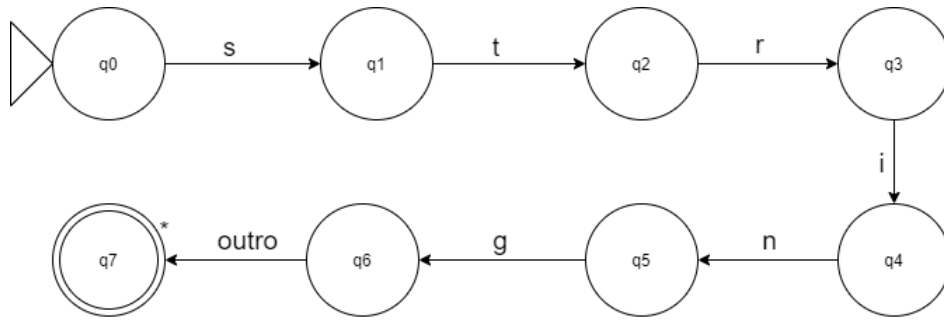












Parte III

Analizador Sintático

4 Ferramenta

4.1 Yacc

O Yacc é uma abreviação do termo em inglês "*Yet Another Compiler-Compiler*", semelhante a ferramenta de mesmo nome para Unix. Ele é o componente do PLY que realiza a análise da gramática.

Cada regra gramatical pode ser definida através de uma função Python, onde cada *docstring* possui a especificação apropriada da gramática livre de contexto. Cada função aceita um único argumento 'p', que é a sequência contendo os valores de cada símbolo da gramática da respectiva regra. Segue um exemplo, de como especificamos a regra para a expressão **FUNCDEF**.

```
def p_funcdef(p):
    '''
    funcdef : DEF IDENT LPAREN paramlist RPAREN LBRACE statelist RBRACE
    '''
```

Normalmente, a primeira regra especificada no yacc determina o começo da gramática. Entretanto, podemos escolher qual regra da gramática o analisador deve ser iniciado passando o parâmetro 'start=rule_expression' para o yacc. No nosso trabalho, ela pode começar pelas funções 'p_program_1', 'p_func_list' ou produção vazia. Note que optamos por utilizar a regra inicial '**program**'.

```
def p_program_1(p):
    '''
    program : statement
    '''

def p_funclist(p):
    '''
    funclist : funcdef funclist1
    '''

...
parser = yacc.yacc(start='program') #build the parser
```

Para verificar quais são os problemas que ocorrem durante a análise sintática, podemos criar uma regra para detectar erros sintáticos que são encontradas durante a análise. Além disso, se algum erro é encontrado na especificação da gramática, o yacc irá produzir mensagens de diagnósticos ou exceções. Os problemas que podem ser encontrados são: funções com nomes duplicados, conflitos gerados por gramáticas ambíguas, regras

gramaticais mal especificadas, recursões infinitas, regras e tokens que não são utilizados ou definidos.

```
def p_error(p):
    if not p:
        print("End of File!")
        return

    print("Erro:", p)
    print(text[p.lexpos - 40:p.lexpos + 40])
```

Fazer o debug de um compilador na maior parte das vezes não é uma tarefa fácil. Para realizar essa tarefa, podemos utilizar o modo de debug do yacc, que gera um arquivo chamado ‘parser.out’, que contém informações das regras gramaticais, símbolos terminais e não-terminais e suas aparições nas regras, os estados gerados pelo método LALR e avisos de conflitos que foram resolvidos automaticamente pelo compilador.

```
yacc.yacc(debug=True)
```

Segue uma amostra resumida do arquivo parser.out

Grammar

```
Rule 0      S' # $\rightarrow$# program
Rule 1      program # $\rightarrow$# statement
Rule 2      program # $\rightarrow$# funclist
Rule 3      program # $\rightarrow$# <empty>
```

...

Terminals, with rules where they appear

```
ASSIGN      : 32 33 60 61 94 95
BREAK       : 30 74 88
COMMA       : 15 43
```

...

Nonterminals, with rules where they appear

```
addsub      : 107 117
allocexpression : 38
atribstat1  : 32 33 60 61 94 95
```

...

Parsing method: LALR

state 0

```
(0) S' # $\rightarrow$# . program
```

```

(1) program #$\rightarrow$# . statement
(2) program #$\rightarrow$# . funclist
...
WARNING: Conflicts:
WARNING:
WARNING: shift/reduce conflict for LBRACKET in state 64 resolved as shift
WARNING: shift/reduce conflict for LBRACKET in state 239 resolved as shift

```

Para verificar a pilha de execução, podemos utilizar o módulo ‘logging’ e passar o seu objeto como parâmetro para o yacc, gerando o arquivo ‘parselog.txt’

```

import logging
logging.basicConfig(
    level = logging.DEBUG,
    filename = "parselog.txt",
    filemode = "w",
    format = "%(filename)10s:%(lineno)4d:%(message)s"
)
log = logging.getLogger()

yacc.yacc(debug=True, debuglog=log)

```

Amostra do arquivo `parselog.txt`

```

yacc.py: 362:PLY: PARSE DEBUG START
yacc.py: 410:
yacc.py: 411:State   : 0
yacc.py: 434:Stack   : . LexToken(FOR,'for',3,0)
yacc.py: 445:Action  : Shift and goto state 22
yacc.py: 410:
yacc.py: 411:State   : 22
yacc.py: 434:Stack   : FOR . LexToken(LPAREN,'(',3,3)
yacc.py: 445:Action  : Shift and goto state 70
yacc.py: 410:
yacc.py: 411:State   : 70
yacc.py: 434:Stack   : FOR LPAREN . LexToken(IDENT,'i',3,4)
yacc.py: 445:Action  : Shift and goto state 140
yacc.py: 410:
...
yacc.py: 411:State   : 2
yacc.py: 430:Defaulted state 2: Reduce using 1
yacc.py: 434:Stack   : statement . $end
yacc.py: 469:Action  : Reduce rule [program #$\rightarrow$# statement] wit
yacc.py: 506:Result  : <NoneType @ 0x90ba10> (None)
yacc.py: 410:
yacc.py: 411:State   : 1

```

```
yacc.py: 434:Stack    : program . $end
yacc.py: 571:Done    : Returning <NoneType @ 0x90ba10> (None)
```

No geral, para se retirar a ambiguidade de gramáticas ambíguas é um processo complicado e demorado, já que não existe algoritmo para resolver esse problema. Entretanto, o yacc permite associar um grau de precedência e associatividade para os tokens individuais, resolvendo a ambiguidade. Em nosso trabalho, realizamos esse procedimento para os casos **IF** e **ELSE**.

```
precedence = (
    ('nonassoc', 'IF'),
    ('left', 'ELSE'),
    ('left', 'LBRACKET')
)
```

Com o parser pronto, podemos realizar a análise sintática do código através da função ‘analyze’, que recebe o código em *string* como parâmetro e verifica se está de acordo com as regras gramaticais.

```
def analyze(input_value):
    global text
    text = input_value
    result = parser.parse(input=input_value, debug=log)
    print(result)
```

Por fim, ao executar o analisador sintático no código ‘lil_example.ccc’ para verificar sua saída, temos:

```
$ make run file="./tmp/lil_example.ccc"

IDENT | (LINE,COLUMN)
i    | [(1, 5), (1, 12), (1, 20), (1, 24), (2, 11)]
Generating LALR tables
WARNING: 2 shift/reduce conflicts
None
```

A saída ‘None’ é única resposta do yacc e indica que o analisador sintático terminou sua execução e os problemas de ambiguidade foram resolvidos através do ‘shift’. Note que as informações adicionais são enviadas para os arquivos parselog.txt ou parser.out.

5 Gramática

Com relação a nossa gramática, transformamos a original e a modificamos pensando em evitar recursão à esquerda. Sendo assim, realizamos apenas a fatoração à esquerda. Segue um exemplo da regra **PARAMLIST** que realizamos uma fatoração:

Antes da Fatoração

```
TYPES → int | float | string
PARAMLIST → TYPES ident , PARAMLIST | TYPES ident | ε
```

Após a Fatoração

```
TYPES → int
      | float
      | string

PARAMLIST → int ident , PARAMLIST
          | float ident , PARAMLIST
          | string ident , PARAMLIST
          | int ident
          | float ident
          | string ident
          | ε
```

Entretanto, a gramática que utilizamos não é LL(1), porque a regra **IFSTAT** consiste de uma produção que pode ser anulável. E o **ELSE** está presente tanto no *First* quanto no *Follow*. Sendo assim, a nossa gramática está em LR(1), porém, através da precedência de tokens, conseguimos adequar nossa gramática ao compilador.

```
precedence = (
('nonassoc', 'IF'),
('left', 'ELSE'),
('left', 'LBRACKET'))
```

```
PROGRAM → STATEMENT
        | FUNCLIST
        | ε
FUNCLIST → FUNCDEF FUNCLIST'
FUNCLIST' → def ident ( PARAMLIST ) { STATELIST } FUNCLIST'
          | ε
FUNCDEF → def ident ( PARAMLIST ) { STATELIST }
TYPES → int
```

```

        | float
        | string
PARAMLIST → string LISTDCL ident PARAMLIST'
        | float LISTDCL ident PARAMLIST'
        | int LISTDCL ident PARAMLIST'
        | ε
PARAMLIST' → , PARAMLIST
        | ε
LISTDCL → [] LISTDCL | ε
STATEMENT → int ident STATEMENT'' ;
        | float ident STATEMENT'' ;
        | string ident STATEMENT'' ;
        | ident STATEMENT' ;
        | PRINTSTAT ;
        | READSTAT ;
        | FUNCCALL ;
        | RETURNSTAT ;
        | IFSTAT
        | FORSTAT
        | WHILESTAT
        | { STATELIST }
        | break ;
        | ;
STATEMENT' → [ NUMEXPRESSION ] LVALUE' = ATRIBSTAT' ;
        | = ATRIBSTAT' ;
        | ( PARAMLISTCALL )
STATEMENT'' → [ NUMEXPRESSION ] LVALUE' ;
        | ;
ATRIBSTAT' → EXPRESSION
        | ALLOCEXPRESSION
        | FUNCCALL
FUNCCALL → ident ( PARAMLISTCALL )
PARAMLISTCALL → FACTOR PARAMLISTCALL''
        | ε
PARAMLISTCALL' → , PARAMLISTCALL
        | ε
PARAMLISTCALL'' → [ NUMEXPRESSION ] LVALUE' PARAMLISTCALL'
        | PARAMLISTCALL'
PRINTSTAT → print EXPRESSION
READSTAT → read EXPRESSION
RETURNSTAT → return RETURNSTAT'
RETURNSTAT' → EXPRESSION
        | ε
IFSTAT → if ( EXPRESSION ) STATEMENT IFSTAT'

```

```

IFSTAT' → else STATEMENT
        | ε
FORSTAT → for ( FORSTAT' ; FORSTAT'' ; FORSTAT' ) STATEMENT
FORSTAT' → ident FORSTAT''
        | ε
FORSTAT'' → EXPRESSION
        | ε
FORSTAT''' → [ NUMEXPRESSION ] LVALUE' = ATRIBSTAT'
        | = ATRIBSTAT'
WHILESTAT → while ( EXPRESSION ) STATEMENT
STATELIST → int LISTDCL ident STATELIST''
        | float LISTDCL ident STATELIST''
        | string LISTDCL ident STATELIST''
        | ident STATELIST'''
        | print EXPRESSION ; STATELIST'
        | read ident STATELIST''
        | ident ( PARAMLISTCALL ) ; STATELIST'
        | return RETURNSTAT' ; STATELIST'
        | if ( EXPRESSION ) STATEMENT IFSTAT' STATELIST'
        | for ( FORSTAT' ; FORSTAT'' ; FORSTAT' ) STATEMENT STATELIST'
        | while ( EXPRESSION ) STATEMENT STATELIST'
        | { STATELIST } STATELIST'
        | break ; STATELIST'
        | ; STATELIST'
STATELIST' → int LISTDCL ident STATELIST''
        | float LISTDCL ident STATELIST''
        | string LISTDCL ident STATELIST''
        | ident STATELIST'''
        | print EXPRESSION ; STATELIST'
        | read ident STATELIST''
        | ident ( PARAMLISTCALL ) ; STATELIST'
        | return RETURNSTAT' ; STATELIST'
        | if ( EXPRESSION ) STATEMENT IFSTAT' STATELIST'
        | for ( FORSTAT' ; FORSTAT'' ; FORSTAT' ) STATEMENT STATELIST'
        | while ( EXPRESSION ) STATEMENT STATELIST'
        | { STATELIST } STATELIST'
        | break ; STATELIST'
        | ; STATELIST'
        | ε
STATELIST'' → [ NUMEXPRESSION ] LVALUE' ; STATELIST'
        | ; STATELIST'
STATELIST''' → [ NUMEXPRESSION ] LVALUE''' = ATRIBSTAT' ; STATELIST'
        | = ATRIBSTAT' ; STATELIST'
ALLOCEXPRESSION → new TYPES [ NUMEXPRESSION ] LVALUE'

```

```

EXPRESSION → NUMEXPRESSION EXPRESSION '
EXPRESSION ' → COMPOPERATOR NUMEXPRESSION
                | ε
COMPOPERATOR → <
                | >
                | <=
                | >=
                | ==
                | !=
NUMEXPRESSION → TERM NUMEXPRESSION '
NUMEXPRESSION ' → ADDSUB TERM
                | ε
ADDSUB → +
        | -
TERM → UNARYEXPR TERM '
TERM ' → MULTDIV UNARYEXPR TERM '
        | ε
MULTDIV → *
        | /
        | %
UNARYEXPR → ADDSUB FACTOR
           | FACTOR
FACTOR → int_constant
        | float_constant
        | string_constant
        | null
        | ident LVALUE '
        | ( NUMEXPRESSION )
LVALUE ' → [ NUM_EXPRESSION ] LVALUE '
          s | ε

```

5.1 Gramática Original

```

PROGRAM → (STATEMENT | FUNCLIST)?
FUNCLIST → FUNCDEF FUNCLIST | FUNCDEF
FUNCDEF → def ident(PARAMLIST){STATELIST}
PARAMLIST → ((int | float | string) ident, PARAMLIST |
              (int | float | string) ident)?
STATEMENT → (VARDECL; | ATRIBSTAT; | PRINTSTAT; |

```

```

        READSTAT; | RETURNSTAT; | IFSTAT | FORSTAT | {STATELIST} |
        break; | ;)

```

```
VARDECL → (int | float | string) ident ([int constant])*
```

```
ATRIBSTAT → LVALUE = (EXPRESSION | ALLOCEXPRESSION | FUNCCALL)
```

```
FUNCCALL → ident(PARAMLISTCALL)
```

```
PARAMLISTCALL → (ident, PARAMLISTCALL | ident)?
```

```
PRINTSTAT → print EXPRESSION
```

```
READSTAT → read LVALUE
```

```
RETURNSTAT → return
```

```
IFSTAT → if(EXPRESSION ) STATEMENT (else STATEMENT)?
```

```
FORSTAT → for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT
```

```
STATELIST → STATEMENT (STATELIST)?
```

```
ALLOCEXPRESSION → new (int | float | string) ([ NUMEXPRESSION ]) +
```

```
EXPRESSION → NUMEXPRESSION(( < | > | <= | >= | == | != ) NUMEXPRESSION)?
```

```
NUMEXPRESSION → TERM ((+ | -) TERM)*
```

```
TERM → UNARYEXPR(( * | / | %) UNARYEXPR)*
```

```
UNARYEXPR → ((+ | -))? FACTOR
```

```
FACTOR → (int_constant | float_constant | string_constant |
        null | LVALUE |(NUMEXPRESSION))
```

```
LVALUE → ident([NUMEXPRESSION])*
```

5.2 Modificações na Gramática

```
PROGRAM → (STATEMENT | FUNCLIST)?
```

```
FUNCLIST → FUNCDEF FUNCLIST | FUNCDEF
```

`FUNCDEF` \rightarrow `def ident(PARAMLIST){STATELIST}`
`PARAMLIST` \rightarrow `((int | float | string) ident, PARAMLIST | (int | float | string) ident)?`
`STATEMENT` \rightarrow `(VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; | FUNCCALL; | RETURNSTAT; | IFSTAT | FORSTAT | WHILESTAT | {STATELIST} | break; | ;)`
`VARDECL` \rightarrow `(int | float | string) ident ([int constant])*`
`ATRIBSTAT` \rightarrow `LVALUE([NUMEXPRESSION])? = (EXPRESSION | ALLOCEXPRESSION | FUNCCALL)`
`FUNCCALL` \rightarrow `ident(PARAMLISTCALL)`
`PARAMLISTCALL` \rightarrow `(ident([NUMEXPRESSION])*, PARAMLISTCALL | ident)?`
`PRINTSTAT` \rightarrow `print EXPRESSION`
`READSTAT` \rightarrow `read LVALUE`
`RETURNSTAT` \rightarrow `return (ident | EXPRESSION)?`
`IFSTAT` \rightarrow `if(EXPRESSION) STATEMENT (else STATEMENT)?`
`FORSTAT` \rightarrow `for(ATRIBSTAT?; EXPRESSION?; ATRIBSTAT?) STATEMENT`
`WHILESTAT` \rightarrow `while(EXPRESSION) STATEMENT`
`STATELIST` \rightarrow `STATEMENT (STATELIST)?`
`ALLOCEXPRESSION` \rightarrow `new (int | float | string) ([NUMEXPRESSION]) +`
`EXPRESSION` \rightarrow `NUMEXPRESSION((< | > | <= | >= | == | !=) NUMEXPRESSION)?`
`NUMEXPRESSION` \rightarrow `TERM ((+ | -) TERM)*`
`TERM` \rightarrow `UNARYEXPR((* | / | %) UNARYEXPR)*`
`UNARYEXPR` \rightarrow `((+ | -))? FACTOR`
`FACTOR` \rightarrow `(int_constant | float_constant | string_constant | null | LVALUE | (NUMEXPRESSION))`

LVALUE \rightarrow ident([NUMEXP RESSION])*

5.3 Forma Convencional

5.3.1 Transformação para definição de gramática convencional

PROGRAM \rightarrow STATEMENT | FUNCLIST | ϵ

FUNCLIST \rightarrow FUNCDEF FUNCLIST | FUNCDEF

FUNCDEF \rightarrow def ident (PARAMLIST) { STATELIST }

TYPES \rightarrow int | float | string

PARAMLIST \rightarrow TYPES LISTDCL_ident , PARAMLIST | TYPES LISTDCL ident | ϵ

LISTDCL \rightarrow []LISTDCL | ϵ

STATEMENT \rightarrow VARDECL ; | ATRIBSTAT ; | PRINTSTAT ; |
 READSTAT ; | RETURNSTAT ; | IFSTAT | FORSTAT |
 WHILESTAT | { STATELIST } | break; | ;

VARDECL \rightarrow TYPES ident VARDECL'

VARDECL' \rightarrow [int_constant] VARDECL' | ϵ

ATRIBSTAT \rightarrow LVALUE ATRIBSTAT' = ATRIBSTAT''

ATRIBSTAT' \rightarrow [NUMEXPRESSION] | ϵ

ATRIBSTAT'' \rightarrow EXPRESSION | ALLOCEXPRESSION | FUNCCALL

FUNCCALL \rightarrow ident (PARAMLISTCALL)

PARAMLISTCALL \rightarrow ident PARAMLISTCALL', PARAMLISTCALL | ident | ϵ

PARAMLISTCALL' \rightarrow [NUMEXPRESSION] | ϵ

PRINTSTAT \rightarrow print EXPRESSION

READSTAT \rightarrow read LVALUE

RETURNSTAT \rightarrow return RETURNSTAT'

RETURNSTAT' \rightarrow ident | EXPRESSION | ϵ

IFSTAT \rightarrow if (EXPRESSION) STATEMENT IFSTAT'

IFSTAT' \rightarrow else STATEMENT | ϵ

FORSTAT \rightarrow for (FORSTAT' ; FORSTAT'' ; FORSTAT') STATEMENT

FORSTAT' \rightarrow ATRIBSTAT | ϵ

FORSTAT'' \rightarrow EXPRESSION | ϵ

WHILESTAT \rightarrow while (EXPRESSION) STATEMENT

STATELIST \rightarrow STATEMENT STATELIST'

STATELIST' \rightarrow STATELIST | ϵ

ALLOCEXPRESSION \rightarrow new TYPES [NUMEXPRESSION] ALLOCEXPRESSION'

ALLOCEXPRESSION' \rightarrow [NUMEXPRESSION] ALLOCEXPRESSION' | ϵ

EXPRESSION \rightarrow NUMEXPRESSION EXPRESSION'

EXPRESSION' \rightarrow COMPOPERATOR NUMEXPRESSION | ϵ

COMPOPERATOR \rightarrow < | > | < = | > = | = = | ! =

NUMEXPRESSION \rightarrow TERM NUMEXPRESSION'

NUMEXPRESSION' \rightarrow ADDSUB TERM | ϵ

ADDSUB \rightarrow + | -

TERM \rightarrow UNARYEXPR TERM'

TERM' \rightarrow MULTDIV UNARYEXPR TERM' | ϵ

MULTDIV \rightarrow * | / | %

UNARYEXPR \rightarrow UNARYEXPR' FACTOR

UNARYEXPR' \rightarrow ADDSUB | ϵ

FACTOR \rightarrow int_constant | float_constant | string_constant | null |
LVALUE | (NUMEXPRESSION)

LVALUE \rightarrow ident LVALUE'

LVALUE' \rightarrow [NUM_EXPRESSION] LVALUE' | ϵ

5.3.2 Fatoração da Gramática

PROGRAM \rightarrow STATEMENT
| FUNCLIST
| ϵ

FUNCLIST \rightarrow FUNCDEF FUNCLIST'

FUNCLIST' \rightarrow def ident (PARAMLIST) { STATELIST } FUNCLIST'
| ϵ

FUNCDEF \rightarrow def ident (PARAMLIST) { STATELIST }

TYPES \rightarrow int
| float
| string

PARAMLIST \rightarrow string LISTDCL ident PARAMLIST'
| float LISTDCL ident PARAMLIST'
| int LISTDCL ident PARAMLIST'
| ϵ

PARAMLIST' \rightarrow , PARAMLIST
| ϵ

LISTDCL \rightarrow []LISTDCL | ϵ

STATEMENT \rightarrow VARDECL ;
| ATRIBSTAT ;
| PRINTSTAT ;
| READSTAT ;
| FUNCCALL ;
| RETURNSTAT ;
| IFSTAT
| FORSTAT

```

    | WHILESTAT
    | { STATELIST }
    | break ;
    | ;

```

```

VARDECL → int ident VARDECL'
        | float ident VARDECL'
        | string ident VARDECL'

```

```

VARDECL' → [ int_constant ] VARDECL'
          | ε

```

```

ATRIBSTAT → LVALUE ATRIBSTAT' = ATRIBSTAT''

```

```

ATRIBSTAT' → [ NUMEXPRESSION ]
            | ε

```

```

ATRIBSTAT'' → EXPRESSION
            | ALLOCEXPRESSION
            | FUNCCALL

```

```

FUNCCALL → ident ( PARAMLISTCALL )

```

```

PARAMLISTCALL → ident PARAMLISTCALL' PARAMLISTCALL''
              | ε

```

```

PARAMLISTCALL' → [NUMEXPRESSION] | ε

```

```

PARAMLISTCALL'' → , PARAMLISTCALL
                | ε

```

```

PRINTSTAT → print EXPRESSION

```

```

READSTAT → read LVALUE

```

```

RETURNSTAT → return RETURNSTAT'

```

```

RETURNSTAT' → ident
            | EXPRESSION
            | ε

```

```

IFSTAT → if ( EXPRESSION ) STATEMENT IFSTAT'

```

```

IFSTAT' → else STATEMENT

```

| ϵ

FORSTAT \rightarrow for (FORSTAT' ; FORSTAT'' ; FORSTAT') STATEMENT

FORSTAT' \rightarrow LVALUE ATRIBSTAT' = ATRIBSTAT''
| ϵ

FORSTAT'' \rightarrow EXPRESSION
| ϵ

WHILESTAT \rightarrow while (EXPRESSION) STATEMENT

STATELIST \rightarrow int ident VARDECL' ; STATELIST'
| float ident VARDECL' ; STATELIST'
| string ident VARDECL' ; STATELIST'
| LVALUE ATRIBSTAT' = ATRIBSTAT'' ; STATELIST'
| print EXPRESSION ; STATELIST'
| read LVALUE ; STATELIST'
| ident (PARAMLISTCALL) ; STATELIST'
| return RETURNSTAT' ; STATELIST'
| if (EXPRESSION) STATEMENT IFSTAT' STATELIST'
| for (FORSTAT' ; FORSTAT'' ; FORSTAT') STATEMENT STATELIST'
| while (EXPRESSION) STATEMENT STATELIST'
| { STATELIST } STATELIST'
| break ; STATELIST'
| ; STATELIST'

STATELIST' \rightarrow int ident VARDECL' ; STATELIST'
| float ident VARDECL' ; STATELIST'
| string ident VARDECL' ; STATELIST'
| LVALUE ATRIBSTAT' = ATRIBSTAT'' ; STATELIST'
| print EXPRESSION ; STATELIST'
| read LVALUE ; STATELIST'
| ident (PARAMLISTCALL) ; STATELIST'
| return RETURNSTAT' ; STATELIST'
| if (EXPRESSION) STATEMENT IFSTAT' STATELIST'
| for (FORSTAT' ; FORSTAT'' ; FORSTAT') STATEMENT STATELIST'
| while (EXPRESSION) STATEMENT STATELIST'
| { STATELIST } STATELIST'
| break ; STATELIST'
| ; STATELIST'
| ϵ

ALLOCEXPRESSION \rightarrow new TYPES [NUMEXPRESSION] ALLOCEXPRESSION'

ALLOCEXPRESSION' \rightarrow [NUMEXPRESSION] ALLOCEXPRESSION'
| ϵ

EXPRESSION \rightarrow NUMEXPRESSION EXPRESSION'

EXPRESSION' \rightarrow COMPOPERATOR NUMEXPRESSION
| ϵ

COMPOPERATOR \rightarrow <
| >
| <=
| >=
| ==
| !=

NUMEXPRESSION \rightarrow TERM NUMEXPRESSION'

NUMEXPRESSION' \rightarrow ADDSUB TERM
| ϵ

ADDSUB \rightarrow +
| -

TERM \rightarrow UNARYEXPR TERM'

TERM' \rightarrow MULTDIV UNARYEXPR TERM'
| ϵ

MULTDIV \rightarrow *
| /
| %

UNARYEXPR \rightarrow UNARYEXPR' FACTOR

UNARYEXPR' \rightarrow +
| -
| ϵ

FACTOR \rightarrow int_constant
| float_constant
| string_constant
| null
| LVALUE

$$| (\text{ NUMEXPRESSION })$$

$$\text{LVALUE} \rightarrow \text{ident LVALUE}'$$

$$\begin{aligned} \text{LVALUE}' &\rightarrow [\text{ NUM_EXPRESSION }] \text{ LVALUE}' \\ &| \epsilon \end{aligned}$$

5.4 Código em Python

```

from lex import tokens
import ply.yacc as yacc

def p_program_1(p):
    '''
    program : statement
    '''

def p_program_2(p):
    '''
    program : funclist
    '''

def p_program_3(p):
    '''
    program :
    '''
    p[0]=None

def p_funclist(p):
    '''
    funclist : funcdef funclist1
    '''

def p_funclist1_1(p):
    '''
    funclist1 : DEF IDENT LPAREN paramlist RPAREN LBRACE statelist
                                     RBRACE funclist1
    '''

def p_funclist1_2(p):
    '''
    funclist1 :
    '''
    p[0]=None

def p_funcdef(p):

```

```
    '''
    funcdef : DEF IDENT LPAREN paramlist RPAREN LBRACE statelist RBRACE
    '''

def p_types_1(p):
    '''
    types : INT
    '''

def p_types_2(p):
    '''
    types : FLOAT
    '''

def p_types_3(p):
    '''
    types : STRING
    '''

def p_paramlist_1(p):
    '''
    paramlist : STRING listdcl IDENT paramlist1
    '''

def p_paramlist_2(p):
    '''
    paramlist : FLOAT listdcl IDENT paramlist1
    '''

def p_paramlist_3(p):
    '''
    paramlist : INT listdcl IDENT paramlist1
    '''

def p_paramlist_4(p):
    '''
    paramlist :
    '''
    p[0]=None

def p_paramlist1_1(p):
    '''
    paramlist1 : COMMA paramlist
    '''

def p_paramlist1_2(p):
    '''
```

```
    paramlist1 :  
        '''  
    p[0]=None  
  
def p_listdcl_1(p):  
    '''  
    listdcl : LBRACKET RBRACKET listdcl  
    '''  
  
def p_listdcl_2(p):  
    '''  
    listdcl :  
    '''  
    p[0]=None  
  
# def p_statement_1(p):  
#     '''  
#     statement : vardecl SEMICOLON  
#     '''  
  
def p_statement_1_1(p):  
    '''  
    statement : INT IDENT statement2  
    '''  
  
def p_statement_1_2(p):  
    '''  
    statement : FLOAT IDENT statement2  
    '''  
  
def p_statement_1_3(p):  
    '''  
    statement : STRING IDENT statement2  
    '''  
  
def p_statement_2(p):  
    '''  
    statement : IDENT statement1  
    '''  
  
def p_statement_3(p):  
    '''  
    statement : printstat SEMICOLON  
    '''  
  
def p_statement_4(p):  
    '''
```

```
    statement : readstat SEMICOLON
    '''

def p_statement_5(p):
    '''
    statement : returnstat SEMICOLON
    '''

def p_statement_6(p):
    '''
    statement : ifstat
    '''

def p_statement_7(p):
    '''
    statement : forstat
    '''

def p_statement_8(p):
    '''
    statement : whilestat
    '''

def p_statement_9(p):
    '''
    statement : LBRACE statelist RBRACE
    '''

def p_statement_10(p):
    '''
    statement : BREAK SEMICOLON
    '''

def p_statement_11(p):
    '''
    statement : SEMICOLON
    '''

def p_statement1_1(p):
    '''
    statement1 : LBRACKET numexpression RBRACKET lvalue1 ASSIGN
                                     atribstat1 SEMICOLON
    '''

def p_statement1_2(p):
    '''
    statement1 : ASSIGN atribstat1 SEMICOLON
```



```
'''

def p_statement1_3(p):
    '''
    statement1 : LPAREN paramlistcall RPAREN SEMICOLON
    '''

def p_statement2_1(p):
    '''
    statement2 : LBRACKET numexpression RBRACKET lvalue1 SEMICOLON
    '''

def p_statement2_2(p):
    '''
    statement2 : SEMICOLON
    '''

def p_atribstat1_1(p):
    '''
    atribstat1 : expression
    '''

def p_atribstat1_2(p):
    '''
    atribstat1 : allocexpression
    '''

def p_atribstat1_3(p):
    '''
    atribstat1 : funccall
    '''

def p_funccall(p):
    '''
    funccall : IDENT LPAREN paramlistcall RPAREN
    '''

def p_paramlistcall_1(p):
    '''
    paramlistcall : factor paramlistcall2
    '''

def p_paramlistcall_2(p):
    '''
    paramlistcall :
    '''
    p[0]=None
```

```
def p_paramlistcall1_1(p):
    '''
    paramlistcall1 : COMMA paramlistcall
    '''

def p_paramlistcall1_2(p):
    '''
    paramlistcall1 :
    '''
    p[0]=None

def p_paramlistcall2_1(p):
    '''
    paramlistcall2 : LBRACKET numexpression RBRACKET lvalue1
                    paramlistcall1
    '''

def p_paramlistcall2_2(p):
    '''
    paramlistcall2 : paramlistcall1
    '''

def p_printstat(p):
    '''
    printstat : PRINT expression
    '''

def p_readstat(p):
    '''
    readstat : READ expression
    '''

def p_returnstat(p):
    '''
    returnstat : RETURN returnstat1
    '''

def p_returnstat1_2(p):
    '''
    returnstat1 : expression
    '''

def p_returnstat1_3(p):
    '''
    returnstat1 :
```

```
    '''
    p[0]=None

def p_ifstat(p):
    '''
    ifstat : IF LPAREN expression RPAREN statement ifstat1
    '''

def p_ifstat1_1(p):
    '''
    ifstat1 : ELSE statement
    '''

def p_ifstat1_2(p):
    '''
    ifstat1 : %prec IF
    '''
    p[0]=None

def p_forstat(p):
    '''
    forstat : FOR LPAREN forstat1 SEMICOLON forstat2 SEMICOLON forstat1
              RPAREN statement
    '''

def p_forstat1_1(p):
    '''
    forstat1 : IDENT forstat3
    '''

def p_forstat1_2(p):
    '''
    forstat1 :
    '''
    p[0]=None

def p_forstat2_1(p):
    '''
    forstat2 : expression
    '''

def p_forstat2_2(p):
    '''
    forstat2 :
    '''
    p[0]=None
```

```
def p_forstat3_1(p):
    '''
    forstat3 : LBRACKET numexpression RBRACKET lvalue1 ASSIGN atribstat1
    '''

def p_forstat3_2(p):
    '''
    forstat3 : ASSIGN atribstat1
    '''

def p_whilestat(p):
    '''
    whilestat : WHILE LPAREN expression RPAREN statement
    '''

def p_statelist_1(p):
    '''
    statelist : INT listdcl IDENT statelist2
    '''

def p_statelist_2(p):
    '''
    statelist : FLOAT listdcl IDENT statelist2
    '''

def p_statelist_3(p):
    '''
    statelist : STRING listdcl IDENT statelist2
    '''

def p_statelist_4(p):
    '''
    statelist : IDENT statelist3
    '''

def p_statelist_5(p):
    '''
    statelist : PRINT expression SEMICOLON statelist1
    '''

def p_statelist_6(p):
    '''
    statelist : READ IDENT statelist2
    '''

def p_statelist_7(p):
    '''
```

```
    statelist : RETURN returnstat1 SEMICOLON statelist1
    '''

def p_statelist_8(p):
    '''
    statelist : IF LPAREN expression RPAREN statement ifstat1 statelist1
    '''

def p_statelist_9(p):
    '''
    statelist : FOR LPAREN forstat1 SEMICOLON forstat2 SEMICOLON
                forstat1 RPAREN statement
                statelist1
    '''

def p_statelist_10(p):
    '''
    statelist : WHILE LPAREN expression RPAREN statement statelist1
    '''

def p_statelist_11(p):
    '''
    statelist : LBRACE statelist RBRACE statelist1
    '''

def p_statelist_12(p):
    '''
    statelist : BREAK SEMICOLON statelist1
    '''

def p_statelist_13(p):
    '''
    statelist : SEMICOLON statelist1
    '''

def p_statelist_14(p):
    '''
    statelist : IDENT LPAREN paramlistcall RPAREN SEMICOLON statelist1
    '''

def p_statelist1_1(p):
    '''
    statelist1 : INT listdcl IDENT statelist2
    '''

def p_statelist1_2(p):
    '''
```

```
    statelist1 : FLOAT listdcl IDENT statelist2
    '''

def p_statelist1_3(p):
    '''
    statelist1 : STRING listdcl IDENT statelist2
    '''

def p_statelist1_4(p):
    '''
    statelist1 : IDENT statelist3
    '''

def p_statelist1_5(p):
    '''
    statelist1 : PRINT expression SEMICOLON statelist1
    '''

def p_statelist1_6(p):
    '''
    statelist1 : READ IDENT statelist2
    '''

def p_statelist1_7(p):
    '''
    statelist1 : RETURN returnstat1 SEMICOLON statelist1
    '''

def p_statelist1_8(p):
    '''
    statelist1 : IF LPAREN expression RPAREN statement ifstat1
                    statelist1
    '''

def p_statelist1_9(p):
    '''
    statelist1 : FOR LPAREN forstat1 SEMICOLON forstat2 SEMICOLON
                    forstat1 RPAREN statement
                    statelist1
    '''

def p_statelist1_10(p):
    '''
    statelist1 : WHILE LPAREN expression RPAREN statement statelist1
    '''

def p_statelist1_11(p):
```

```
    '''
    statelist1 : LBRACE statelist RBRACE statelist1
    '''

def p_statelist1_12(p):
    '''
    statelist1 : BREAK SEMICOLON statelist1
    '''

def p_statelist1_13(p):
    '''
    statelist1 : SEMICOLON statelist1
    '''

def p_statelist1_14(p):
    '''
    statelist1 : IDENT LPAREN paramlistcall RPAREN SEMICOLON statelist1
    '''

def p_statelist1_15(p):
    '''
    statelist1 :
    '''
    p[0]=None

def p_statelist2_1(p):
    '''
    statelist2 : LBRACKET numexpression RBRACKET lvalue1 SEMICOLON
                statelist1
    '''

def p_statelist2_2(p):
    '''
    statelist2 : SEMICOLON statelist1
    '''

def p_statelist3_1(p):
    '''
    statelist3 : LBRACKET numexpression RBRACKET lvalue1 ASSIGN
                atribstat1 SEMICOLON statelist1
    '''

def p_statelist3_2(p):
    '''
    statelist3 : ASSIGN atribstat1 SEMICOLON statelist1
    '''
```

```
def p_allocexpression(p):
    '''
    allocexpression : NEW types LBRACKET numexpression RBRACKET lvalue1
    '''

def p_expression(p):
    '''
    expression : numexpression expression1
    '''

def p_expression1_1(p):
    '''
    expression1 : compoperator numexpression
    '''

def p_expression1_2(p):
    '''
    expression1 :
    '''
    p[0]=None

def p_compoperator_1(p):
    '''
    compoperator : GT
    '''

def p_compoperator_2(p):
    '''
    compoperator : LT
    '''

def p_compoperator_3(p):
    '''
    compoperator : GE
    '''

def p_compoperator_4(p):
    '''
    compoperator : LE
    '''

def p_compoperator_5(p):
    '''
    compoperator : EQ
    '''

def p_compoperator_6(p):
```



```
    '''
    compoperator : NEQ
    '''

def p_numexpression(p):
    '''
    numexpression : term numexpression1
    '''

def p_numexpression1_1(p):
    '''
    numexpression1 : addsub term
    '''

def p_numexpression1_2(p):
    '''
    numexpression1 :
    '''
    p[0]=None

def p_addsub_1(p):
    '''
    addsub : PLUS
    '''

def p_addsub_2(p):
    '''
    addsub : MINUS
    '''

def p_term(p):
    '''
    term : unaryexpr term1
    '''

def p_term1_1(p):
    '''
    term1 : multdiv unaryexpr term1
    '''

def p_term1_2(p):
    '''
    term1 :
    '''
    p[0]=None

def p_multdiv_1(p):
```

```
    '''
    multdiv : MULTIPLY
    '''

def p_multdiv_2(p):
    '''
    multdiv : DIVIDE
    '''

def p_multdiv_3(p):
    '''
    multdiv : REM
    '''

def p_unaryexpr_1(p):
    '''
    unaryexpr : addsub factor
    '''

def p_unaryexpr_2(p):
    '''
    unaryexpr : factor
    '''

def p_factor_1(p):
    '''
    factor : int_constant
    '''

def p_factor_2(p):
    '''
    factor : float_constant
    '''

def p_factor_3(p):
    '''
    factor : string_constant
    '''

def p_factor_4(p):
    '''
    factor : null_constant
    '''

def p_factor_5(p):
    '''
    factor : IDENT lvalue1
```

```

    '''

def p_factor_6(p):
    '''
    factor : LPAREN numexpression RPAREN
    '''

def p_lvalue1_1(p):
    '''
    lvalue1 : LBRACKET numexpression RBRACKET lvalue1
    '''

def p_lvalue1_2(p):
    '''
    lvalue1 :
    '''

text = ""

def p_error(p):
    if not p:
        print("End of File!")
        return

    print("Erro:", p)
    print(text[p.lexpos - 40:p.lexpos + 40])

import logging
logging.basicConfig(
    level = logging.DEBUG,
    filename = "./debug/parselog.txt",
    filemode = "w",
    format = "%(filename)10s:%(lineno)4d:%(message)s"
)
log = logging.getLogger()

precedence = (
    ('nonassoc', 'IF'),
    ('left', 'ELSE'),
    ('left', 'LBRACKET')
)

parser = yacc.yacc(start='program', outputdir='./debug') #build the
                                     parser

def analyze(input_value):
    global text

```

```
text = input_value
result = parser.parse(input=input_value, debug=log)
print(result)
```

Parte IV

Analizador Semântico

5.5 Introdução

Os analisadores semânticos, por definição, verificam os erros semânticos em um código. Sendo assim, é analisado o escopo dos nomes, correspondências entre declarações, uso dos nomes e compatibilidade dos tipos em expressões.

Nesta etapa do trabalho, verificamos todos estes processos. Para isso, utilizamos a gramática fatorada à esquerda **ConvCC-2021-2**. Através dela, criamos uma Árvore de Expressão (EXPA) para expressões aritméticas, uma SDD L-Atribuída e sua transformação de SDD para SDT, como também construímos uma árvore de expressão levando em consideração as regras semânticas.

Na parte de Declaração de Variáveis, realizamos a inserção do tipo na tabela e também construímos uma SDD L-Atribuída e convertemos-a para uma SDT. Para o problema de verificação de tipos, realizamos operações para verificar se uma expressão é válida, considerando os tipos de cada operando.

Além disso, para a Verificação de Identificadores por Escopo, onde um identificador não pode ser declarado duas vezes no mesmo escopo para tipos diferentes, foram adicionados escopos interno e externo para realizar a verificação. Por fim, durante a Verificação de Comandos dentro de escopos, como no caso de um comando *break* fora de um loop, foi utilizado uma verificação por escopo. Por fim, executamos os mesmos processos de SDD, SDD L-Atribuída, e SDT para ConvCC-2021-1, no qual foi utilizada no código intermediário.

5.6 Árvore de Expressão (EXPA)

Para iniciar o processo de construção da Árvore de Expressão (EXPA), foi necessário separar da gramática ConvCC-2021-2 fatorada à esquerda, as produções que derivam expressões aritméticas, formando a gramática de EXPA conforme descrita abaixo.

[GRAMATICA de EXPA]

EXPRESSION \rightarrow NUMEXPRESSION

NUMEXPRESSION \rightarrow TERM NUMEXPRESSION ' ,

NUMEXPRESSION ' \rightarrow ADDSUB TERM
| ϵ

ADDSUB \rightarrow +
| -

TERM \rightarrow UNARYEXPR TERM ' ,

```
TERM' → MULTDIV UNARYEXPR TERM'
      | ε
```

```
MULTDIV → *
          | /
          | %
```

```
UNARYEXPR → UNARYEXPR' FACTOR
```

```
UNARYEXPR' → FACTOR
```

```
FACTOR → int_constant
        | float_constant
        | string_constant
        | null
        | LVALUE
        | ( NUMEXPRESSION )
```

```
LVALUE → ident LVALUE'
```

```
LVALUE' → [ NUM_EXPRESSION ] LVALUE'
         | ε
```

Com a nossa gramática EXPA separada e levando em consideração possíveis ciclos que poderiam ser gerados através das produções, foi criada a SDD L-Atribuída. Para demonstrar a estruturação da SDD, utilizamos a produção TERM, com sua regra semântica na chave “**node**” e valor léxico na variável “**rtype**”. Note que para realizar a verificação de tipos é utilizado a função **check_type()**, que será demonstrada na seção 5.8. A implementação da SDD L-Atribuída com mais detalhes se encontra no arquivo *'Relatorio/Gramaticas/EXPA.sdd'*.

```
'''
term : unaryexpr term1
'''

if term1['code']:
    rtype = check_type(unaryexpr['node'], term1['node'], term1['operator'], unaryexpr.lineno)

    var = get_var()

    term = {
```



```

        'node': Node(term1['operator'], unaryexpr['node'], term1['node'],
                     , rtype),
        'code': [*unaryexpr['code'], "{var} = {unaryexpr['label']} {
                    term1['operator']} {term1['
                    label']}]"],
        'label': var
    }
else:
    term = unaryexpr

```

Para que uma SDD seja classificada como uma SDD L-Atribuída, é necessário que não exista ciclos entre as produções. Sendo assim, cada atributo deve ser sintetizado ou herdados do pai, dos irmãos ou de si mesmo, permitindo que suas produções sigam apenas uma única direção. Ao analisar a SDD L-Atribuída no arquivo *EXPA.sdd*, é possível notar que não existe ciclos entre as produções, os atributos são sintetizados ou herdados do pai. Levando em consideração as regras, a SDD é L-Atribuída.

A construção da SDT para a SDD de EXPA, pode ser encontrada em '*Relatorio/Gramaticas/EXPA.sdd*'. Para realizar a sua transformação, basta adicionar as regras semânticas definidas nas próprias produções. Segue uma amostra da SDT para a produção TERM.

```

term : unaryexpr term1 {
if term1['code']:
    rtype = check_type(unaryexpr['node'], term1['node'], term1['operator'],
                        unaryexpr.lineno)

    var = get_var()

    term.node = Node(term1['operator'], unaryexpr['node'], term1['node'],
                     , rtype),
    term.label = var
    term.code = [*unaryexpr['code'], "{var} = {unaryexpr['label']} {
                    term1['operator']} {term1['label']}]"],
else:
    term = unaryexpr
}

```

Para realizar a validação da expressões derivadas de EXPA, criamos uma árvore de expressão para “ $x + 1 * 2$ ”, conforme a árvore abaixo. Utilizando a SDD criada, seguimos a árvore por suas folhas da esquerda para à direita através dos seguintes passos:

Primeiramente, iniciamos pela folha "ident = x".

```
1) FACTOR = x
Node(IDENT + lvalue1['expression'], None, None,
type2str(var.type, var.dimension, lvalue1['dim']))

2) UNARYEXPR = x
Node(term1['operator'], unaryexpr['node'], term1['node'], rtype)

3) UNARYEXPR = FACTOR = X
4) TERM = x
Node(term1['operator'], unaryexpr['node'], term1['node'], rtype)
```

Agora, partindo pela folha de valor "1"

```
5) FACTOR = 1 = int
Node(int_constant, None, None, 'int')

6) UNARYEXPR = FACTOR = 1 = int
```

Agora, partindo pela folha de valor "2"

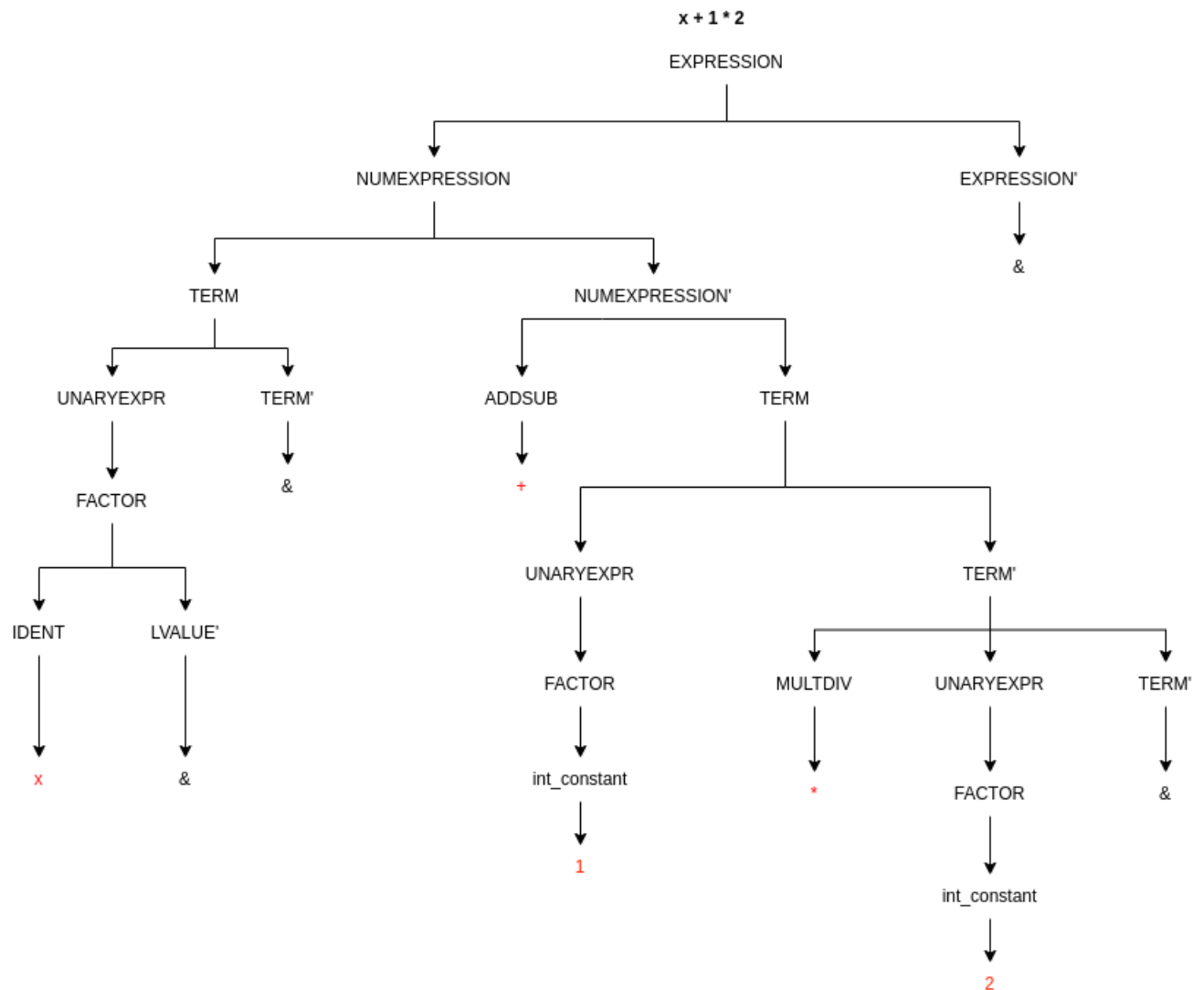
```
7) FACTOR = 2 = int
Node(int_constant, None, None, 'int')

8) TERM' = 2 = int
Node(term1['operator'], unaryexpr['node'], term1['node'], rtype)

9) NUMEXPRESSION' = 1 * 2
Node(addsub['operator'], term['node'], numexpression1['node'], rtype)

10) NUMEXPRESSION = x + 1 * 2
Node(numexpression1['operator'], term['node'], numexpression1['node'], rtype)

11) EXPRESSION = x + 1 * 2
expression = numexpression
```



5.7 Inserção do tipo na tabela de símbolos

Para realizar o processo de construção da árvore de declarações, realizamos os mesmos procedimentos semelhantes à construção da árvore EXPA. Primeiro, separamos a grámatica de DEC e utilizamos uma tabela de símbolos com uma função chamada **new_entry()** para inserir na tabela. Seus parâmetros são: identificador, tipo, tabela e número da linha em que foi encontrado.

Para demonstrar, que a SDD de DEC é L-Atribuída, basta verificar se os atributos são sintetizados ou se são herdados pelo pai, irmão ou por si mesmo, desde que não haja ciclos entre as produções. Ao verificar o arquivo em '*Relatorio/Gramáticas/DEC.sdd*' notamos que todas produções respeitam essas condições e, portanto, é L-Atribuída. Agora, para transformar a SDD em SDT, adicionamos as regras semânticas nas produções. Segue abaixo uma produção chamada PARAMLIST em SDD e outra em SDT. Note que a SDT de DEC se encontra em '*Relatorio/Gramáticas/DEC.sdt*'

```

# PARAMLIST EM SDD

'''
paramlist : STRING listdcl IDENT paramlist1
'''

scope = scope_stack[-1]
entry = EntradaTabela(IDENT, STRING, listdcl['dim'], [-1] * listdcl['dim'], IDENT.lineno)
scope.new_entry(entry)

paramlist = { 'dim': paramlist1['dim'] + 1, 'code': ['from_params {IDENT}', *paramlist1['code']] }

# PARAMLIST EM SDT

paramlist : STRING listdcl IDENT paramlist1 {
    scope = scope_stack[-1]
    entry = EntradaTabela(IDENT, STRING, listdcl['dim'], [-1] * listdcl['dim'], IDENT.lineno)
    scope.new_entry(entry)

    paramlist.dim = paramlist1['dim'] + 1
    paramlist.code = ['from_params {IDENT}'], *paramlist1['code']]
}

```

5.8 Verificação de Tipo

A verificação dos tipos é necessária para validar as entradas de uma expressão de acordo com uma regra semântica entre operações diferentes. Dentro da função `check_type()`, é verificado o tipo dos nodos da direita e da esquerda. Caso essa combinação não exista dentro do dicionário, é retornado o valor **None** e mostrado um erro chamado *InvalidTypeOperationError*, que indica um tipo de operação inválida.

```

def check_type(left, right, operation, lineno):
    valids = {
        ('string', '+', 'string'): 'string',
        ('string', '+', 'int'): 'string',
        ('string', '+', 'float'): 'string',
        ('int', '+', 'string'): 'string',
        ('float', '+', 'string'): 'string',
        ('int', '+', 'int'): 'int',
        ('int', '-', 'int'): 'int',
        ('int', '*', 'int'): 'int',
        ('int', '%', 'int'): 'int',

```

```

        ('int', '/', 'int'): 'float',
        ('float', '+', 'float'): 'float',
        ('float', '-', 'float'): 'float',
        ('float', '*', 'float'): 'float',
        ('float', '/', 'float'): 'float',
        ('float', '+', 'int'): 'float',
        ('float', '-', 'int'): 'float',
        ('float', '*', 'int'): 'float',
        ('float', '/', 'int'): 'float',
        ('int', '+', 'float'): 'float',
        ('int', '-', 'float'): 'float',
        ('int', '*', 'float'): 'float',
        ('int', '/', 'float'): 'float',
    }

    result = valids.get((left.result_type, operation, right.result_type),
                        , None)

    if result is None:
        raise InvalidTypeOperationError(f'Error on operation {operation}
                                         : {left.result_type},{right.result_type},{lineno}')

    return result

```

5.9 Verificação de Identificadores por Escopo

Para declarar uma variável dentro de um escopo, ela precisa seguir determinadas regras. Uma delas é que não pode aparecer o mesmo identificador com dois tipos diferentes. Caso seja declarada como **float x**, não pode ser possível declara-lá como **string x** dentro do mesmo escopo. Para tratar esse escopo, foi criada a classe `Escopo` e o método `new_entry()`. Dentro dessa classe, foi adicionada tabela de símbolos denominada `escopo`, uma lista de `escopo` internos e um booleano `loop` que mostra se um escopo é um comando normal ou de repetição. Todas as vezes que uma nova entrada é inserida, é verificado se ela já está presente na tabela. Caso esteja, é retornado o erro `VariableAlreadyDeclaredError`, indicando que ela já foi declarada.

```

class Escopo:
    ''' Classe de Escopo '''

    def __init__(self, label='global', escopo_pai=None, loop=False):
        self.label = label
        self.escopo_pai = escopo_pai
        self.loop = loop
        self.table = []

```

```

        self.escpos_internos = []

    def new_entry(self, entry):
        '''Adiciona entrada na tabela de simbolos'''

        # Se variavel ja foi declarada
        var = list(filter(lambda x: x.ident == entry.ident, self.table))

        if len(var) > 0:
            raise VariableAlreadyDeclaredError('Variavel ja declarada')

        self.table.append(entry)

```

5.10 Comandos dentro do Escopo

Seguindo as orientações, a implementação da linguagem deve considerar a presença de um *break* fora do escopo de um comando de repetição. Para realizar a verificação, utilizamos um escopo pai com o objetivo de verificar se o comando *break* está dentro de um escopo, do contrário é emitido o erro *BreakOutLoopError*.

```

def p_statement_10(p):
    '''
    statement : BREAK SEMICOLON
    '''
    scope = scope_stack[-1]
    while scope:
        if scope.loop:
            break

        scope = scope.escopo_pai

    if not scope:
        raise BreakOutLoopError(p.lineno(2))

    p[0] = { 'code': [f'goto {NEXT_LOOP_LABEL}'] }

```

Para simular o comando *break* fora de um loop, criamos um programa chamado *'tmp/-break_scope_error.ccc'*, dentro dele é utilizado um loop do tipo “for”, com um comando *break* fora do escopo. Segue abaixo a saída do programa com o comportamento esperado, um erro do tipo “*BreakOutLoopError*”.

```
$ make run file='tmp/break_scope_error.ccc'
```

```

IDENT      | (LINE,COLUMN)
main       | [(1, 5)]
  i        | [(2, 9), (5, 9), (5, 16), (5, 24), (5, 28), (7, 13)]
  a        | [(3, 9)]
  b        | [(4, 9), (7, 9)]                                     f
Generating LALR tables
WARNING: 2 shift/reduce conflicts
...
raise BreakOutLoopError(p.lineno(2))

classes.exception.BreakOutLoopError: 18
make: *** [Makefile:16: run] Error 1

```

5.11 Geração de Código Intermediário

A tradução do programa em alto nível para o código do processador está associado a traduzir para a linguagem-alvo a representação da árvore gramatical, ou seja, a transformação da árvore de derivação em um segmento de código. Uma das vantagens dessa realização é simplificar a implementação do compilador.

Nesse contexto, realizamos a transformação da gramática ConvCC-2021-1. Para isso, criamos a sua SDD L-Atribuída e a sua SDT. A prova de que a SDD é L-Atribuída segue as mesmas que utilizamos na SDD de EXPA e DEC. Dessa forma, conseguimos utilizá-la para geração do código intermediário. Para efeitos de demonstração, na produção FORSTART, podemos ver na chave **'code'** trechos que compõe o código intermediário. Note que os arquivos da SDD e SDT, se encontram respectivamente em: *'Relatorio/Gramaticas/ConvCC-2021-1.sdd'* e *'Relatorio/Gramaticas/ConvCC-2021-1.sdt'*

```

def p_forstat(p):
    '''
    forstat : new_loop_label FOR LPAREN forstat1 SEMICOLON forstat2
                                           SEMICOLON forstat1 RPAREN
                                           new_scope_loop LBRACE statelist
                                           RBRACE
    '''
    scope_stack.pop()

    start = generate_label()
    next_label = NEXT_LOOP_LABEL

    p[0] = {
        'code': [
            *p[4]['code'],

```

```

    f'$label:{start}:',
    *p[6]['code'],
    *([f"if False {p[6]['label']} goto {next_label}" if p[6]['label'] else []),

    *p[12]['code'],
    *p[8]['code'],
    f'goto {start}',
    f'$label:{next_label}:',
]
}

```

Codigo Intermediario Gerado

Arquivo 'debug/ci.txt' gerado ao compilar 'lil_example.ccc'

```
$ make run file='tmp/lil_example.ccc'
```

```

f:
    | goto L3
    |
    | from_params a
    | from_params b
    | t1 = a
    | read t1
    | t2 = b
    | print t2
L2:
    |
    | t3 = a
    | t4 = b
    | t5 = t3 > t4
    | if False t5 goto L1
    | t6 = b
    | t7 = 2
    | t8 = t6 - t7
    | a = t8
    | goto L2
L1:
    |
    | param t9
    | param t10
    | call f, 2
    | t11 = b
    | return t11
L3:
    |
    | goto L7
main:
    |
    | from_params argn
    | from_params argc

```



```

| int i
| t12 = 3
| t13 = 4
| int a[t12][t13]
| int b
| t14 = 0
| i = t14
L6: |
| t15 = i
| t16 = 10
| t17 = t15 < t16
| if False t17 goto L4
| t21 = 2
| string c[t21]
| t22 = 0
| t23 = "a"
| c[t22] = t23
| t24 = 1
| t25 = "b"
| c[t24] = t25
| t26 = 3
| float b[t26]
| float j
| t27 = 3
| t28 = new float[t27]
| b = t28
| t29 = i
| print t29
| t30 = i
| t31 = 2
| t32 = t30 >= t31
| if False t32 goto L5
| t33 = 4
| j = t33
| goto L4
L5: |
| t18 = i
| t19 = 1
| t20 = t18 + t19
| i = t20
| goto L6
L4: |
| t34 = 0
| return t34
```

L7: |

5.12 Lazy Check

Um problema encontrado no desenvolvimento do trabalho, foi a definição de funções recursivas, visto que quando uma função chama ela própria no seu código, sua definição ainda não existe na tabela de símbolos, pois essa inclusão só acontece após o final da interpretação da função. Dessa forma, para solucionar esse problema e possibilitar o uso de código recursivo, foi criada uma funcionalidade de checagem preguiçosa através da função lazy check e uma lista de variáveis que não existem no escopo. Nesse sentido, quando uma função termina de ser interpretada essa função lazy check é chamada, de forma que ela percorre essa lista de variáveis que não existem no escopo e procura pelo nome da função recursiva e caso esse nome seja encontrado, a função será reinterpretada mas agora com a definição existente na tabela de símbolos possibilitando a funcionalidade de recursão. Segue abaixo a implementação da função lazy check.

```
def lazy_check():
    global lazy_check_var

    for var in lazy_check_var:
        search_var(*var)

    lazy_check_var = []

def search_var(ident, line_no, lazy=False):
    global lazy_check_var
    if scope_stack:
        scope = scope_stack[-1]
        while scope:
            for e in scope.table:
                if e.ident == ident:
                    return e

            scope = scope.escopo_pai

    if lazy:
        lazy_check_var.append((ident, line_no))
        return
    else:
        raise VariableAlreadyDeclaredError(f'Linha {line_no}:
                                            Variavel {ident} nao
                                            declarada')

    raise ScopeNotExistError()
```

5.13 Exceções

Algumas exceções foram definidas para organizar melhor o código e melhorar a visualização caso ocorra um erro na compilação. No total, foram declaradas 8 exceções conforme mostrado abaixo.

```
class VariableAlreadyDeclaredError(Exception):
    ''' A variavel ja foi declarada '''

class VariableNotDeclaredError(Exception):
    ''' A variavel nao foi declarada '''

class ScopeNotExistError(Exception):
    ''' O escopo nao existe '''

class InvalidTypeOperationError(Exception):
    ''' Tipo de operacao invalida '''

class BreakOutLoopError(Exception):
    ''' Break fora de um loop '''

class TypeError(Exception):
    ''' Operacao com variavel de tipo invalido '''

class IdentifierNotFunction(Exception):
    ''' A variavel e um indentificador, nao uma funcao '''

class ParamCountError(Exception):
    ''' Numero invalido de parametros '''
```

5.14 Saída

A saída do compilador, em caso de sucesso, é demonstrada na saída abaixo ao compilar o arquivo 'lil_example.ccc'. Note que a saída é a tabela de símbolos, uma mensagem de sucesso para expressões aritméticas, declaração de variáveis e checagem de break. Além disso, é mostrado o local dos arquivos onde cada estrutura se encontra.

```
$ make run file='tmp/lil_example.ccc'
```

IDENT	(LINE, COLUMN)
f	[(1, 5), (9, 5)]
a	[(1, 11), (2, 10), (5, 12), (6, 9), (9, 10), (15, 9)]
b	[(1, 18), (3, 11), (5, 16), (6, 13), (9, 7), (11, 12), (16, 9),
main	[(13, 5)]
argn	[(13, 14)]

```

    argc    | [(13, 29)]
    i       | [(14, 9), (17, 9), (17, 16), (17, 24), (17, 28), (24, 15), (25, 13)]
    c       | [(18, 16), (19, 9), (20, 9)]
    j       | [(22, 15), (26, 13)]
Generating LALR tables
WARNING: 2 shift/reduce conflicts

Expressoes aritmeticas: Sucesso
Declaracao de Variaveis: Sucesso
Checagem de Breaks: Sucesso

Codigo Intermediario Gerado: ./debug/ci.txt
Arvore de expressoes numericas: ./debug/num_expressions.txt
Escopos e tabela de simbolos: ./debug/scopes.txt

```

5.15 Saída com Erro de Compilação

Para exemplificar uma saída com erro de compilação, tentamos compilar o arquivo de teste 'variable_not_declared_error.ccc' com uma variável que não foi declarada no escopo.

```

$ make run file='tmp/variable_not_declared_error.ccc'

IDENT          | (LINE,COLUMN)
main           | [(1, 5)]
a              | [(2, 9), (6, 5), (10, 11), (14, 22)]
b              | [(3, 9), (7, 5), (11, 11), (14, 26)]
c              | [(4, 9), (8, 5), (12, 11), (14, 30)]
d_not_declared | [(14, 5), (16, 11)]

Generating LALR tables
WARNING: 2 shift/reduce conflicts
...
    raise VariableAlreadyDeclaredError(f'Linha {line_no}:
    Variavel {ident} nao declarada')

    classes.exception.VariableAlreadyDeclaredError: Linha 0:
    Variavel d_not_declared nao declarada
make: *** [Makefile:16: run] Error 1

```

5.16 Árvore de Expressão (EXPA)

Exemplo da árvore gerada ao executar o programa 'lil_example.ccc'.

```
[
  {
    "ID": "(a>b)",
    "lineno": 0,
    "tree": {
      "left": {
        "left": null,
        "right": null,
        "value": "a"
      },
      "right": {
        "left": null,
        "right": null,
        "value": "b"
      },
      "value": ">"
    }
  },
  {
    "ID": "(b-2)",
    "lineno": 0,
    "tree": {
      "left": {
        "left": null,
        "right": null,
        "value": "b"
      },
      "right": {
        "left": null,
        "right": null,
        "value": "2"
      },
      "value": "-"
    }
  },
  {
    "ID": "(i<10)",
    "lineno": 0,
    "tree": {
      "left": {
```

```
        "left": null,
        "right": null,
        "value": "i"
    },
    "right": {
        "left": null,
        "right": null,
        "value": "10"
    },
    "value": "<"
}
},
{
    "ID": "(i+1)",
    "lineno": 0,
    "tree": {
        "left": {
            "left": null,
            "right": null,
            "value": "i"
        },
        "right": {
            "left": null,
            "right": null,
            "value": "1"
        },
        "value": "+"
    }
},
{
    "ID": "(i>=2)",
    "lineno": 0,
    "tree": {
        "left": {
            "left": null,
            "right": null,
            "value": "i"
        },
        "right": {
            "left": null,
            "right": null,
            "value": "2"
        },
        "value": ">="
    }
}
```

```

    }
  }
]

```

5.17 Tabela de Símbolos, Tipo e Escopo

Exemplo da tabela de símbolos, tipo e escopo ao executar o programa 'lil_example.ccc'.

```

{
  "escopos_internos": [
    {
      "escopos_internos": [
        {
          "escopos_internos": [],
          "label": 1,
          "table": []
        }
      ],
      "label": 1,
      "table": [
        {
          "dimension": 0,
          "ident": "b",
          "line": 32,
          "sizes": [],
          "type": "int"
        },
        {
          "dimension": 0,
          "ident": "a",
          "line": 32,
          "sizes": [],
          "type": "int"
        }
      ]
    },
    {
      "escopos_internos": [
        {
          "escopos_internos": [
            {
              "escopos_internos": [],
              "label": 1,
              "table": []
            }
          ]
        }
      ]
    }
  ]
}

```

```

    }
  ],
  "label": 1,
  "table": [
    {
      "dimension": 1,
      "ident": "c",
      "line": 49,
      "sizes": [
        "2"
      ],
      "type": "string"
    },
    {
      "dimension": 1,
      "ident": "b",
      "line": 52,
      "sizes": [
        "3"
      ],
      "type": "float"
    },
    {
      "dimension": 0,
      "ident": "j",
      "line": 53,
      "sizes": [],
      "type": "float"
    }
  ]
}
],
"label": 2,
"table": [
  {
    "dimension": 1,
    "ident": "argc",
    "line": 44,
    "sizes": [
      -1
    ],
    "type": "string"
  },
  {

```



```

        "dimension": 0,
        "ident": "argn",
        "line": 44,
        "sizes": [],
        "type": "int"
    },
    {
        "dimension": 0,
        "ident": "i",
        "line": 45,
        "sizes": [],
        "type": "int"
    },
    {
        "dimension": 2,
        "ident": "a",
        "line": 46,
        "sizes": [
            "3",
            "4"
        ],
        "type": "int"
    },
    {
        "dimension": 0,
        "ident": "b",
        "line": 47,
        "sizes": [],
        "type": "int"
    }
]
}
],
"label": "global",
"table": [
    {
        "dimension": 2,
        "ident": "f",
        "line": 32,
        "sizes": [],
        "type": "function"
    },
    {
        "dimension": 2,

```

```
    "ident": "main",  
    "line": 44,  
    "sizes": [],  
    "type": "function"  
  }  
]  
}
```

Conclusão

Os compiladores traduzem o código fonte de uma linguagem de programação de alto nível para uma linguagem de programação de baixo nível. Sem eles a tarefa de programar seria um trabalho extremamente lento e difícil. São nichos específicos e muito raros os casos em que desenvolvemos aplicações feitas diretamente em Assembly. Nos dias atuais, quase todas as linguagens possuem o seu compilador, facilitando e aumentando a efetividade dos programadores. Dito isto, com esse trabalho, foi possível entender a primeira parte da construção de um compilador, ou seja, do analisador léxico. Já em sua segunda parte, do analisador sintático, compreendemos os cuidados e formas de se criar uma gramática consistente. Utilizando a ferramenta PLY, entendemos os princípios utilizados para se construir um compilador de uma linguagem qualquer através da sua aplicação prática. Além disso, com o Diagrama de Transição foi possível entender como ocorre a leitura das palavras e como são identificados os tokens. Através da análise semântica e geração de código intermediário, compreendemos de como são utilizadas e aplicadas regras semânticas. Portanto, ao realizar este trabalho, ficou evidente como criar um compilador através de ferramentas geradores de analisadores léxicos e sintáticos e dos estudos e pesquisas sobre compiladores.

Apêndices

APÊNDICE A – SDD de ConvCC-2021-1

```

scope_stack = []
num_expressions = []

search_var(ident, lineno, lazy)
num_expressions_as_json()
new_scope(loop)
check_type(left, right, operation, lineno)
lazy_check()

'''
new_loop_label : &
'''

NEXT_LOOP_LABEL = generate_label()

'''
new_scope : &
'''

add_scope(False)

'''
new_scope_loop : &
'''

add_scope(True)

'''
program : new_scope statement | new_scope funclist
'''

global_scope = scope_stack.pop()
program = {
    'scopes': global_scope.as_json(),
    'num_expressions': num_expressions_as_json(),
    'code': statement['code']
}

'''

```

```

program : &
'''

'''
funclist : funcdef funclist1
'''

funclist = {
    'code': [*funcdef['code'], *funclist1['code']]
}

'''
funclist1 : funclist
'''

funclist1 = {'code': funclist['code']}

'''
funclist1 : &
'''

funclist1 = {'code': []}

'''
funcdef : DEF IDENT new_scope LPAREN paramlist RPAREN LBRACE statelist RBRACE
'''

scope_stack.pop()

scope = scope_stack[-1]
entry = EntradaTabela(IDENT, 'function', paramlist['dim'], [], IDENT.lineno)

scope.new_entry(entry)

lazy_check()

next = generate_label()
funcdef = { 'code': ['goto {next}', '{IDENT}:', *paramlist['code'], *statelist['code']] }

'''
types : INT
'''

```

```

types = {'type': INT, 'code': INT}

'''
types : FLOAT
'''

types = {'type': FLOAT, 'code': FLOAT}

'''
types : STRING
'''

types = {'type': STRING, 'code': STRING}

'''
paramlist : STRING listdcl IDENT paramlist1
'''

scope = scope_stack[-1]
entry = EntradaTabela(IDENT, STRING, listdcl['dim'], [-1] * listdcl['dim'],
scope.new_entry(entry)

paramlist = { 'dim': paramlist1['dim'] + 1, 'code': ['from_params {IDENT}'],

'''
paramlist : FLOAT listdcl IDENT paramlist1
'''

scope = scope_stack[-1]
entry = EntradaTabela(IDENT, FLOAT, listdcl['dim'], [-1] * listdcl['dim'],
scope.new_entry(entry)

paramlist = { 'dim': paramlist1['dim'] + 1, 'code': ['from_params {IDENT}'],

'''
paramlist : INT listdcl IDENT paramlist1
'''

scope = scope_stack[-1]
entry = EntradaTabela(IDENT, INT, listdcl['dim'], [-1] * listdcl['dim'], IDENT,
scope.new_entry(entry)

paramlist = { 'dim': paramlist1['dim'] + 1, 'code': [f'from_params {IDENT}'],

'''

```

```

paramlist : &
'''

paramlist = { 'dim': 0, 'code': [] }

'''

paramlist1 : COMMA paramlist
'''

paramlist1 = { 'dim': paramlist['dim'], 'code': paramlist['code'] }

'''

paramlist1 : &
'''

paramlist1 = { 'dim': 0, 'code': [] }

'''

listdcl-1 : LBRACKET RBRACKET listdcl-2
'''

listdcl-1 = {'dim': listdcl-2['dim'] + 1, 'code': "[]{"listdcl-2['code']"}"}

'''

listdcl : &
'''

listdcl = {'dim': 0, 'code': ''}

'''

statement : INT IDENT statement2
'''

entry = EntradaTabela(IDENT, INT, statement2['dim'], statement2['sizes'], IDENT)
scope = scope_stack[-1]
scope.new_entry(entry)

statement = {'code': [*statement2['code'], "int {IDENT}{statement2['aux_code']}"}

'''

```

```

statement : FLOAT IDENT statement2
'''

entry = EntradaTabela(IDENT, FLOAT, statement2['dim'], statement2['sizes'],
scope = scope_stack[-1]
scope.new_entry(entry)

statement = {'code': [*statement2['code'], "float {IDENT}{statement2['aux_co

'''

statement : STRING IDENT statement2
'''

entry = EntradaTabela(IDENT, STRING, statement2['dim'], statement2['sizes'],
scope = scope_stack[-1]
scope.new_entry(entry)

statement = {'code': [*statement2['code'], "string {IDENT}{statement2['aux_co

'''

statement : IDENT statement1
'''

var = search_var(IDENT, IDENT.lineno, statement1['type'] == 'function')
if statement1['type'] == 'function':
    statement = { 'code': [*statement1['code'], "call {IDENT}{statement1['au
else:
    statement = { 'code': [*statement1['code'], "{IDENT}{statement1['aux_cod

'''

statement : printstat SEMICOLON
'''

statement = { 'code' : printstat['code'] }

'''

statement : readstat SEMICOLON
'''

statement = { 'code' : readstat['code'] }

'''

```

```

statement : returnstat SEMICOLON
'''
statement = { 'code' : returnstat['code'] }

'''
statement : ifstat
'''
statement = { 'code' : ifstat['code'] }

'''
statement : forstat
'''
statement = { 'code' : forstat['code'] }

'''
statement : whilestat
'''
statement = { 'code' : whilestat['code'] }

'''
statement : new_scope LBRACE statelist RBRACE
'''
scope_stack.pop()

statement = { 'code' : statelist['code'] }

'''
statement : BREAK SEMICOLON
'''

statement = { 'code': [f'goto {NEXT_LOOP_LABEL}'] }

'''
statement : SEMICOLON
'''
statement = { 'code': [] }

'''
statement1 : LBRACKET numexpression RBRACKET lvalue1 ASSIGN atribstat1 SEMICOLON
'''

statement1 = {
    'type': 'attribution',

```

```

        'left_var_dim': lvalue1['dim'] + 1,
        'right_var': atribstat1,
        'code': [*numexpression['code'], *lvalue1['code'], *atribstat1['code']],
        'aux_code': "[{numexpression['label']}]{lvalue1['aux_code']} = {atribsta
    }

    '''
statement1 : ASSIGN atribstat1 SEMICOLON
'''

statement1 = {
    'type': 'attribution',
    'left_var_dim': 0,
    'right_var': atribstat1,
    'code': atribstat1['code'],
    'aux_code': " = {atribstat1['label']}"
}

'''
statement1 : LPAREN paramlistcall RPAREN SEMICOLON
'''

statement1 = { 'type': 'function', 'params': paramlistcall['params'], 'code':

'''
statement2 : LBRACKET numexpression RBRACKET lvalue1 SEMICOLON
'''

statement2 = {
    'dim': lvalue1['dim'] + 1,
    'sizes': [numexpression['node'], *lvalue1['sizes']],
    'code': [*numexpression['code'], *lvalue1['code']],
    'aux_code': "[{numexpression['label']}]{lvalue1['aux_code']}"
}

'''
statement2 : SEMICOLON
'''

statement2 = { 'dim': 0, 'sizes': [], 'code': [], 'aux_code': '' }

```

```

'''
atribstat1 : expression
'''
atribstat1 = { 'atrib_type' : 'expr', 'node': expression['node'], 'code': expres

'''
atribstat1 : allocexpression
'''
atribstat1 = { 'atrib_type' : 'alloc', 'alloc': allocexpression, 'code': allocex

'''
atribstat1 : funccall
'''
atribstat1 = { 'atrib_type' : 'funccall', 'function': funccall, 'code': funccall

'''
funccall : IDENT LPAREN paramlistcall RPAREN
'''

funccall = { 'code': [*paramlistcall['code'], "{tvar} = call {IDENT}, {len(param

'''
paramlistcall : factor paramlistcall2
'''

paramlistcall = { 'params': [var, *paramlistcall2['params']], 'code': [f"param {

'''
paramlistcall : &
'''
paramlistcall = { 'params': [], 'code': [] }

'''
paramlistcall1 : COMMA paramlistcall
'''
paramlistcall = { 'params': paramlistcall['params'], 'code': paramlistcall['code

```

```

'''
paramlistcall1 : &
'''
paramlistcall = { 'params': [], 'code': [] }

'''
paramlistcall2 : LBRACKET numexpression RBRACKET lvalue1 paramlistcall1
'''

paramlistcall2 = {
    'array': True,
    'dim': lvalue1['dim'] + 1,
    'sizes': [numexpression['node'], *lvalue1['sizes']],
    'params': paramlistcall1['params'],
    'code': [*numexpression['code'], *lvalue1['code'], *paramlistcall1['code']],
    'aux_code': "[{numexpression['label']}]{lvalue1['aux_code']}"
}

'''
paramlistcall2 : paramlistcall1
'''

paramlistcall2 = {
    'array': False,
    'code': paramlistcall1['code'],
    'params': paramlistcall1['params'],
    'aux_code': ''
}

'''
printstat : PRINT expression
'''

printstat = { 'code': [*expression['code'], "print {expression['label']}"] }

'''
readstat : READ expression
'''

```

```

readstat = { 'code': [*expression['code'], "read {expression['label']}"] }

,,,

returnstat : RETURN returnstat1
,,,

returnstat = { 'code': [*returnstat1['code'], "return {returnstat1['label']}"] }

,,,

returnstat1 : expression
,,,

returnstat1 = expression

,,,

returnstat1 : &
,,,

returnstat1 = { 'code': [], 'label': '' }

,,,

ifstat : IF LPAREN expression RPAREN statement ifstat1
,,,

end_if_label = generate_label()
if 'label' in ifstat1:
    next = ifstat1['label']

    ifstat = {
        'code': [
            *expression['code'],
            "if False {expression['label']} goto {next}",
            *statement['code'],
            "goto {end_if_label}",
            *ifstat1['code'],
            '{end_if_label}:'
        ]
    }
}
else:

```

```

        ifstat = {
            'code': [
                *expression['code'],
                "if False {expression['label']} goto {end_if_label}",
                *statement['code'],
                '{end_if_label}:'
            ]
        }

'''
ifstat1 : ELSE statement
'''

label = generate_label()
ifstat1 = {
    'code': [f'{label}:', *statement['code']],
    'label': label
}

'''
ifstat1 : &
'''

ifstat1 = { 'code': [] }

'''
forstat : new_loop_label FOR LPAREN forstat1-1 SEMICOLON forstat2 SEMICOLON
'''

scope_stack.pop()

start = generate_label()
next_label = NEXT_LOOP_LABEL

forstat = {
    'code': [
        *forstat1-1['code'],
        '{start}:',
        *forstat2['code'],
        *([f"if False {forstat2['label']} goto {next_label}"] if forstat2['l
        *statelist['code'],

```

```

        *forstat1-2['code'],
        'goto {start}',
        '{next_label}:',
    ]
}

'''
forstat1 : IDENT forstat3
'''

search_var(IDENT, IDENT.lineno)
forstat1 = { 'code': [*forstat3['code'], "{IDENT} {forstat3['aux_code']}"] }

'''
forstat1 : &
'''

forstat1 = { 'code': [] }

'''
forstat2 : expression
'''

forstat2 = { 'code': expression['code'], 'label': expression['label'] }

'''
forstat2 : &
'''

forstat2 = { 'code': [], 'label': None }

'''
forstat3 : LBRACKET numexpression RBRACKET lvalue1 ASSIGN atribstat1
'''
forstat3 = {
    'code': [*numexpression['code'], *lvalue1['code']],
    'aux_code': "[{numexpression['label']}] {lvalue1['aux_code']} = {atribstat1["
}

```

```

    , , ,
forstat3 : ASSIGN atribstat1
    , , ,
forstat3 = { 'code': atribstat1['code'], 'aux_code': "= {atribstat1['label']"

    , , ,
whilestat : new_loop_label WHILE LPAREN expression RPAREN new_scope_loop LBR
    , , ,

scope_stack.pop()

start = generate_label()
next = NEXT_LOOP_LABEL

whilestat = {
    'code': [
        '{start}:',
        *expression['code'],
        "if False {expression['label']} goto {next}",
        *statement['code'],
        'goto {start}',
        '{next}:',
    ]
}

    , , ,
statelist : statement statelist1
    , , ,

statelist = { 'code': [*statement['code'], *statelist1['code']] }

    , , ,
statelist1 : statelist
    , , ,

statelist1 = { 'code': statelist['code'] }

    , , ,
statelist1 : &
    , , ,

```

```

statelist1 = { 'code': [] }

'''
allocexpression : NEW types LBRACKET numexpression RBRACKET lvalue1
'''

num_expressions.append((numexpression['node'], numexpression.lineno))

var = get_var()

allocexpression = {
    'type': convert_type(types['type']),
    'dim': lvalue1['dim'] + 1,
    'label': var,
    'code': [*numexpression['code'], *lvalue1['code'], "{var} = new {types['code']}"]
}

'''
expression : numexpression expression1
'''

if expression1['code']:
    rvar = get_var()
    expression = {
        'node': Node(expression1['operator'], numexpression['node'], expression1['node']),
        'label': rvar,
        'code': [
            *numexpression['code'],
            *expression1['code'],
            "{rvar} = {numexpression['label']}" + (f" {expression1['operator']}" if expression1['operator'] else "")
        ]
    }
else:
    expression = numexpression

num_expressions.append((expression['node'], numexpression.lineno))

'''
expression1 : compoperator numexpression
'''

expression1 = {
    'operator': compoperator,
    'node': numexpression['node'],

```

```
        'label': numexpression['label'],
        'code': numexpression['code']
    }
```

```
'''
```

```
expression1 : &
```

```
'''
```

```
expression1 = { 'code': '' }
```

```
'''
```

```
compoperator : GT
```

```
'''
```

```
compoperator = '>'
```

```
'''
```

```
compoperator : LT
```

```
'''
```

```
compoperator = '<'
```

```
'''
```

```
compoperator : GE
```

```
'''
```

```
compoperator = '>='
```

```
'''
```

```
compoperator : LE
```

```
'''
```

```
compoperator = '<='
```

```
'''
```

```
compoperator : EQ
```

```
'''
```

```
compoperator = '=='
```

```
'''
```

```
compoperator : NEQ
```

```
'''
```

```
compoperator = '!='
```

```

'''
numexpression : term numexpression1
'''

if numexpression1['code']:
    rtype = check_type(term['node'], numexpression1['node'], numexpression1['ope

    var = get_var()
    numexpression = {
        'node': Node(numexpression1['operator'], term['node'], numexpression1['n
        'code': [*term['code'], *numexpression1['code'], "{var} = {term['label']}
        'label': var
    }
else:
    numexpression = term

'''
numexpression1 : addsub term
'''

numexpression1 = { 'node': term['node'], 'operator': addsub, 'code': term['code']

'''
numexpression1 : &
'''

numexpression1 = { 'code': [] }

'''
addsub : PLUS
'''

addsub = '+'

'''
addsub : MINUS
'''

addsub = '-'

'''

```

```

term : unaryexpr term1
'''

if term1['code']:
    rtype = check_type(unaryexpr['node'], term1['node'], term1['operator'],

    var = get_var()

    term = {
        'node': Node(term1['operator'], unaryexpr['node'], term1['node'], rt
        'code': [*unaryexpr['code'], "{var} = {unaryexpr['label']} {term1['c
        'label': var
    }
else:
    term = unaryexpr

'''
term1 : multdiv unaryexpr term1
'''

if term1['code']:
    rtype = check_type(unaryexpr['node'], term1['node'], term1['operator'],

    var = get_var()

    term1 = {
        'node': Node(term1['operator'], unaryexpr['node'], term1['node'], rt
        'operator': multdiv,
        'code': [*unaryexpr['code'], *term1['code'], "{var} = {unaryexpr['la
        'label': var
    }
else:
    term1 = { 'node': unaryexpr['node'], 'operator': multdiv, 'code': unarye

'''
term1 : &
'''

term1 = { 'code': [] }

'''
multdiv : MULTIPLY

```

```

'''

multdiv = '*'

'''

multdiv : DIVIDE
'''

multdiv = '/'

'''

multdiv : REM
'''

multdiv = '%'

'''

unaryexpr : addsub factor
'''
if (addsub == '-'):
    factor['node'].value *= -1

var = get_var()

unaryexpr = { 'node': factor['node'], 'code': [*factor['code'], *([f"{var} = {ad

'''

unaryexpr : factor
'''

unaryexpr = factor

'''

factor : int_constant
'''
var = get_var()

factor = { 'node': Node(int_constant, None, None, 'int'), 'code': [f'{var} = {in

'''

factor : float_constant

```

```

'''

var = get_var()

factor = { 'node': Node(float_constant, None, None, 'float'), 'code': [f'{var}
]

'''

factor : string_constant
'''

var = get_var()

factor = { 'node': Node(string_constant, None, None, 'string'), 'code': [f'{var}
]

'''

factor : null_constant
'''

var = get_var()

factor = { 'node': Node('null', None, None, 'null'), 'code': [f'{var} = null
]

'''

factor : IDENT lvalue1
'''

var = search_var(IDENT, IDENT.lineno)
nvar = get_var()

factor = {
    'node': Node(IDENT + lvalue1['expression'], None, None, type2str(var.type)),
    'code': [f"{nvar} = {IDENT}{lvalue1['aux_code']}"],
    'label': nvar,
    'vartype': 'ident'
}

'''

factor : LPAREN numexpression RPAREN
'''

factor = numexpression

```

```

num_expressions.append((numexpression['node'], numexpression.lineno))

'''
lvalue1 : LBRACKET numexpression RBRACKET lvalue1
'''
lvalue1 = {
    'dim': lvalue1['dim'] + 1,
    'sizes': [numexpression['node'], *lvalue1['sizes']],
    'expression': '[{numexpression}]{lvalue1}',
    'code': [*numexpression['code'], *lvalue1['code']],
    'aux_code': "[{numexpression['label']}]{lvalue1['aux_code']}"
}

num_expressions.append((numexpression['node'], numexpression.lineno))

'''
lvalue1 : &
'''
lvalue1 = {
    'dim': 0,
    'sizes': [],
    'expression': '',
    'code': [],
    'aux_code': ''
}

```

APÊNDICE B – SDT de ConvCC-2021-1

```

scope_stack = []
num_expressions = []

search_var(ident, lineno, lazy)
num_expressions_as_json()
new_scope(loop)
check_type(left, right, operation, lineno)

new_loop_label : & { NEXT_LOOP_LABEL = generate_label() }

new_scope : & { add_scope(False) }

new_scope_loop : & { add_scope(True) }

program : new_scope statement {
    global_scope = scope_stack.pop()

    program.scopes = global_scope.as_json(),
    program.num_expressions = num_expressions_as_json(),
    program.code = statement['code']
}

program : new_scope funclist {
    global_scope = scope_stack.pop()

    program.scopes = global_scope.as_json(),
    program.num_expressions = num_expressions_as_json(),
    program.code = statement['code']
}

program : &

funclist : funcdef funclist1 { funclist.code = [*funcdef['code'], *funclist1] }

funclist1 : funclist { funclist1.code = funclist['code'] }

funclist1 : & { funclist1.code = [] }

funcdef : DEF IDENT new_scope LPAREN paramlist RPAREN LBRACE statelist RBRACE

```

```

    scope_stack.pop()

    scope = scope_stack[-1]
    entry = EntradaTabela(IDENT, 'function', paramlist['dim'], [], IDENT.lineno)

    scope.new_entry(entry)

    lazy_check()

    next = generate_label()
    funcdef.code = ['goto {next}', '{IDENT}:', *paramlist['code'], *statelist['c
}

types : INT {
    types.type= INT
    types.code = INT
}

types : FLOAT {
    types.type = FLOAT
    types.code = FLOAT
}

types : STRING {
    types.type = STRING
    types.code = STRING
}

paramlist : STRING listdcl IDENT paramlist1 {
    scope = scope_stack[-1]
    entry = EntradaTabela(IDENT, STRING, listdcl['dim'], [-1] * listdcl['dim'],
    scope.new_entry(entry)

    paramlist.dim = paramlist1['dim'] + 1
    paramlist.code = ['from_params {IDENT}', *paramlist1['code']]
}

paramlist : FLOAT listdcl IDENT paramlist1 {
    scope = scope_stack[-1]

```

```

    entry = EntradaTabela(IDENT, FLOAT, listdcl['dim'], [-1] * listdcl['dim'])
    scope.new_entry(entry)

    paralist.dim = paramlist1['dim'] + 1
    paramlist.code = ['from_params {IDENT}', *paramlist1['code']]

}

paramlist : INT listdcl IDENT paramlist1 {
    scope = scope_stack[-1]
    entry = EntradaTabela(IDENT, INT, listdcl['dim'], [-1] * listdcl['dim'])
    scope.new_entry(entry)

    paralist.dim = paramlist1['dim'] + 1
    paramlist.code = ['from_params {IDENT}', *paramlist1['code']]
}

paramlist: & {
    paramlist.dim = 0
    paramlist.code = []
}

paramlist1 : COMMA paramlist {
    paramlist1.dim = paramlist['dim']
    paramlist1.code = paramlist['code']
}

paramlist1 : & {
    paramlist1.dim = 0
    paramlist1.code = []
}

listdcl-1 : LBRACKET RBRACKET listdcl-2 {
    listdcl-1.dim = listdcl['dim'] + 1
    listdcl-1.code = "[]{" + listdcl['code'] + "}"
}

listdcl : & {
    listdcl.dim = 0
    listdcl.code = ''
}

statement : INT IDENT statement2 {
    entry = EntradaTabela(IDENT, INT, statement2['dim'], statement2['sizes'])

```

```

    scope = scope_stack[-1]
    scope.new_entry(entry)

    statement.code = [*statement2['code'], "int {IDENT}{statement2['aux_code']}"]
}

statement : FLOAT IDENT statement2 {
    entry = EntradaTabela(IDENT, FLOAT, statement2['dim'], statement2['sizes'],
    scope = scope_stack[-1]
    scope.new_entry(entry)

    statement.code = [*statement2['code'], "int {IDENT}{statement2['aux_code']}"]
}

statement : STRING IDENT statement2 {
    entry = EntradaTabela(IDENT, STRING, statement2['dim'], statement2['sizes'],
    scope = scope_stack[-1]
    scope.new_entry(entry)

    statement.code = [*statement2['code'], "int {IDENT}{statement2['aux_code']}"]
}

statement : IDENT statement1 {
    var = search_var(IDENT, IDENT.lineno, statement1['type'] == 'function')
    if statement1['type'] == 'function':
        statement.code = [*statement1['code'], "call {IDENT}{statement1['aux_cod
    else:
        statement.code = [*statement1['code'], "{IDENT}{statement1['aux_code']}"]
}

statement : printstat { statement.code = printstat['code'] } SEMICOLON

statement : readstat { statement.code = readstat['code'] } SEMICOLON

statement : returnstat { statement.code = returnstat['code'] } SEMICOLON

statement : ifstat { statement.code = ifstat['code'] }

statement : forstat { statement.code = forstat['code'] }

statement : whilestat { statement.code = whilestat['code'] }

statement : new_scope LBRACE statelist {
    scope_stack.pop()

```

```

        statement.code = statelist['code']
    } RBRACE

statement : BREAK { statement.code = ['goto {NEXT_LOOP_LABEL}'] } SEMICOLON

statement : SEMICOLON { statement.code = [] }

statement1 : LBRACKET numexpression RBRACKET lvalue1 ASSIGN atribstat1 {
    statement1.type = 'attribution'
    statement1.left_var_dim = lvalue1['dim'] + 1
    statement1.right_var = atribstat1
    statement1.code = [*numexpression['code'], *lvalue1['code'], *atribstat1]
    statement1.aux_code = "[{numexpression['label']}]{lvalue1['aux_code']}"
} SEMICOLON

statement1 : ASSIGN atribstat1 {
    statement1.type = 'attribution'
    statement1.left_var_dim = 0
    statement1.right_var = atribstat1
    statement1.code = atribstat1['code']
    statement1.aux_code = " = {atribstat1['label']}"
} SEMICOLON

statement1 : LPAREN paramlistcall {
    statement1.type = 'function'
    statement1.params = paramlistcall['params']
    statement1.code = paramlistcall['code']
    statement1.aux_code = ", {len(paramlistcall['params'])}"
} RPAREN SEMICOLON

statement2 : LBRACKET numexpression RBRACKET lvalue1 {
    statement2.dim = lvalue1['dim'] + 1
    statement2.sizes = [str(numexpression['node']), *lvalue1['sizes']]
    statement2.code = [*numexpression['code'], *lvalue1['code']]
    statement2.aux_code = "[{numexpression['label']}]{lvalue1['aux_code']}"
} SEMICOLON

statement2 : SEMICOLON {
    statement2.dim = 0
    statement2.sizes = []
    statement2.code = []
    statement2.aux_code = ''
}

```

```

atribstat1 : expression {
    atribstat1.atrib_type = 'expr'
    atribstat1.node = expression['node']
    atribstat1.code = expression['code']
    atribstat1.label = expression['label']
}

atribstat1 : allocexpression {
    atribstat1.'atrib_type' : 'alloc'
    atribstat1.alloc = allocexpression
    atribstat1.code = allocexpression['code']
    atribstat1.label = allocexpression['label']
}

atribstat1 : funccall {
    atribstat1.atrib_type = 'funccall'
    atribstat1.function = funccall
    atribstat1.code = funccall['code']
    atribstat1.label = funccall['label']
}

funccall : IDENT LPAREN paramlistcall {
    funccall.code = [*paramlistcall['code'], "{tvar} = call {IDENT}, {len(paramlistcall)}"],
    funccall.label = tvar
} RPAREN

paramlistcall : factor paramlistcall2 {
    paramlistcall.params = [var, *paramlistcall2['params']]
    paramlistcall.code = ["param {factor['label']} {paramlistcall2['aux_code']}"],
}

paramlistcall : & {
    paramlistcall.params = []
    paramlistcall.code = []
}

paramlistcall1 : COMMA paramlistcall {
    paramlistcall1.params = paramlistcall['params']
    paramlistcall1.code = paramlistcall['code']
}

```

```

paramlistcall1 : & {
    paramlistcall1.params = []
    paramlistcall1.code = []
}

paramlistcall2 : LBRACKET numexpression RBRACKET lvalue1 paramlistcall1 {
    paramlistcall2.array = True
    paramlistcall2.dim = lvalue1['dim'] + 1
    paramlistcall2.sizes = [numexpression['node'], *lvalue1['sizes']]
    paramlistcall2.params = paramlistcall1['params']
    paramlistcall2.code = [*numexpression['code'], *lvalue1['code'], *paramlistcall1['code']]
    paramlistcall2.aux_code = "[{numexpression['label']}]{lvalue1['aux_code']}]"
}

paramlistcall2 : paramlistcall1 {
    paramlistcall2.array = False
    paramlistcall2.code = paramlistcall1['code']
    paramlistcall2.params = paramlistcall1['params']
    paramlistcall2.aux_code = ''
}

printstat : PRINT expression { printstat.code = [*expression['code'], "print "],
                                printstat.label = expression['label'] }

readstat : READ expression { readstat.code = [*expression['code'], "read {expression['label']}"],
                                readstat.label = expression['label'] }

returnstat : RETURN returnstat1 { returnstat.code = [*returnstat1['code'], "return {returnstat1['label']}"],
                                returnstat.label = returnstat1['label'] }

returnstat1 : expression { returnstat1 = expression }

returnstat1 : & {
    returnstat1.code = [],
    returnstat1.label = ''
}

ifstat : IF LPAREN expression RPAREN statement ifstat1 {
    end_if_label = generate_label()
    if 'label' in ifstat1:
        next = ifstat1['label']

    ifstat.code = [
        *expression['code'],
        "if False {expression['label']} goto {next}",
    ]
}

```

```

        *statement['code'],
        "goto {end_if_label}",
        *ifstat1['code'],
        '{end_if_label}:'
    ]
else:
    ifstat.code = [
        *expression['code'],
        "if False {expression['label']} goto {end_if_label}",
        *statement['code'],
        '{end_if_label}:'
    ]
}

ifstat1 : ELSE statement {
    label = generate_label()
    ifstat1.code = ['{label}:', *statement['code']]
    ifstat1.label = label
}

ifstat1 : & { ifstat1.code = [] }

forstat : new_loop_label FOR LPAREN forstat1-1 SEMICOLON forstat2 SEMICOLON forstat3
    scope_stack.pop()

    start = generate_label()
    next_label = NEXT_LOOP_LABEL

    forstat.code = [
        *forstat1-1['code'],
        '{start}:',
        *forstat2['code'],
        *(["if False {forstat2['label']} goto {next_label}"] if forstat2['label'] else ''),
        *statelist['code'],
        *forstat1-2['code'],
        'goto {start}',
        '{next_label}:'
    ]
} RBRACE

forstat1 : IDENT forstat3 {
    search_var(IDENT, IDENT.lineno)
    forstat1.code = [*forstat3['code'], "{IDENT} {forstat3['aux_code']}"]
}

```

```

forstat1 : & { forstat1.code = [] }

forstat2 : expression {
    forstat2.code = expression['code']
    forstat2.label = expression['label']
}

forstat2 : & {
    forstat2.code = []
    forstat2.label = None
}

forstat3 : LBRACKET numexpression RBRACKET lvalue1 ASSIGN atribstat1 {
    forstat3.code = [*numexpression['code'], *lvalue1['code']]
    forstat3.aux_code = "[{numexpression['label']}] {lvalue1['aux_code']} = +
}

forstat3 : ASSIGN atribstat1 {
    forstat3.code = atribstat1['code']
    forstat3.aux_code = "= {atribstat1['label']}"
}

whilestat : new_loop_label WHILE LPAREN expression RPAREN new_scope_loop LBRACE
    scope_stack.pop()

    start = generate_label()
    next = NEXT_LOOP_LABEL

    whilestat.code = [
        '{start}:',
        *expression['code'],
        "if False {expression['label']} goto {next}",
        *statement['code'],
        'goto {start}',
        '{next}:',
    ]
} RBRACE

statelist : statement statelist1 { statelist.code = [*statement['code'], *st

statelist1 : statelist { statelist1.code = statelist['code'] }

statelist1 : & { statelist1.code = [] }

```

```

allocexpression : NEW types LBRACKET numexpression {
    num_expressions.append((numexpression['node'], numexpression.lineno))
} RBRACKET lvalue1 {
    var = get_var()

    allocexpression.label = var
    allocexpression.type = convert_type(types['type'])
    allocexpression.dim = lvalue1['dim'] + 1
    allocexpression.code = [*numexpression['code'], *lvalue1['code'], "{var} = r
}

expression : numexpression expression1 {
    if expression1['code']:
        rvar = get_var()
        expression.node = Node(expression1['operator'], numexpression['node'], e
        expression.label = rvar
        expression.code = [
            *numexpression['code'],
            *expression1['code'],
            "{rvar} = {numexpression['label']}" + (" {expression1['operator']
        ]
    else:
        expression = numexpression

    num_expressions.append((expression['node'], numexpression.lineno))
}

expression1 : compoperator { expression1.operator = compoperator } numexpression
    expression1.node = numexpression['node']
    expression1.label = numexpression['label']
    expression1.'code' = numexpression['code']
}

expression1 : & { expression1.code = '' }

compoperator : GT { compoperator = '>' }

compoperator : LT { compoperator = '<' }

compoperator : GE { compoperator = '>=' }

compoperator : LE { compoperator = '<=' }

```

```

compoperator : EQ { compoperator = '==' }

compoperator : NEQ { compoperator = '!=' }

numexpression : term numexpression1 {
    if numexpression1['code']:
        rtype = check_type(term['node'], numexpression1['node'], numexpression1['code'])

        var = get_var()
        numexpression.node = Node(numexpression1['operator'], term['node'],
                                   numexpression1['node'],
                                   numexpression1['code'])
        numexpression.label = var
        numexpression.code = [
            *term['code'],
            *numexpression1['code'],
            "{var} = {term['label']} {numexpression1['operator']} {numexpression1['code']}"
        ]
    else:
        numexpression = term
}

numexpression1 : addsub { numexpression1.operator = addsub } term {
    numexpression1.node = term['node']
    numexpression1.code = term['code']
    numexpression1.label = term['label']
}

numexpression1 : & { numexpression1.code = [] }

addsub : PLUS { addsub = '+' }

addsub : MINUS { addsub = '-' }

term : unaryexpr term1 {
    if term1['code']:
        rtype = check_type(unaryexpr['node'], term1['node'], term1['operator'])

        var = get_var()

        term.node = Node(term1['operator'], unaryexpr['node'], term1['node'],
                           term1['code'])
        term.label = var
        term.code = [*unaryexpr['code'], "{var} = {unaryexpr['label']} {term1['operator']} {term1['code']}"
        ]
    else:
        term = unaryexpr
}

```

```

term1-1 : multdiv { term1-1.operator = multdiv } unaryexpr term1-2 {
    if term1['code']:
        rtype = check_type(unaryexpr['node'], term1-2['node'], term1-2['operator'])

        var = get_var()

        term1-1.node = Node(term1-2['operator'], unaryexpr['node'], term1-2['node'], rtype)
        term1-1.label = var
        term1-1.code = [*unaryexpr['code'], *term1-2['code'], "{var} = {unaryexpr} {term1-2['code']}"]
    else:
        term1-1.node = unaryexpr['node']
        term1-1.label = unaryexpr['label']
        term1-1.code = unaryexpr['code']
}

term1 : & { term1.code = [] }

multdiv : MULTIPLY { multdiv = '*' }

multdiv : DIVIDE { multdiv = '/' }

multdiv : REM { multdiv = '%' }

unaryexpr : addsub factor {
    if (addsub == '-'):
        factor['node'].value *= -1

    var = get_var()

    unaryexpr.node = factor['node']
    unaryexpr.code = [*factor['code'], *("{var} = {addsub}{factor['label']}")]
    unaryexpr.label = var
}

unaryexpr : factor { unaryexpr = factor }

factor : int_constant {
    var = get_var()

    factor.node = Node(int_constant, None, None, 'int')
    factor.code = ['{var} = {int_constant}'], 'label = var'
}

```

```

factor : float_constant {
    var = get_var()

    factor.node = Node(float_constant, None, None, 'float')
    factor.code = ['{var} = {float_constant}']
    factor.label = var
}

factor : string_constant {
    var = get_var()

    factor.node = Node(string_constant, None, None, 'string')
    factor.code = ['{var} = {string_constant}']
    factor.label = var
}

factor : null_constant {
    var = get_var()

    factor.node = Node('null', None, None, 'null')
    factor.code = code = ['{var} = null']
    factor.label = var
}

factor : IDENT lvalue1 {
    var = search_var(IDENT, IDENT.lineno)
    nvar = get_var()

    factor.node = Node(IDENT + lvalue1['expression'], None, None, type2str(v
    factor.code = ["{nvar} = {IDENT}{lvalue1['aux_code']}"]
    factor.label = nvar
    factor.vartype = 'ident'
}

factor : LPAREN numexpression RPAREN {
    factor = numexpression

    num_expressions.append((numexpression['node'], numexpression.lineno))
}

lvalue1 : LBRACKET numexpression RBRACKET lvalue1 {
    lvalue1.dim = lvalue1['dim'] + 1
    lvalue1.sizes = [str(numexpression['node']), *lvalue1['sizes']]
    lvalue1.expression = ' [{str(numexpression)}] {lvalue1}'

```

```
lvalue1.code = [*numexpression['code'], *lvalue1['code']]
lvalue1.aux_code = "[{numexpression['label ']}]{lvalue1['aux_code ']}"

num_expressions.append((numexpression['node'], numexpression.lineno))
}

lvalue1 : & {
    lvalue1.dim = 0
    lvalue1.sizes = []
    lvalue1.expression = ''
    lvalue1.code = []
    lvalue1.aux_code = ''
}
```