
UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CONSTRUÇÃO DE COMPILADORES

# Analisador Sintático

Francisco Luiz Vicenzi
Gustavo Emanuel Kundlatsch
Thiago Sant Helena da Silva

**PROFESSOR**
Alvaro Junio Pereira Franco

Florianópolis
Abril de 2021

# Sumário

# 1 Introdução

Este relatório apresenta a descrição do trabalho realizado para a construção de um Analisador Sintático para a gramática **CC-2020-2**, descrita em 1.1.

A análise sintática é a segunda etapa do processo de compilação. Neste trabalho, o intuito é transformar a gramática de estudo em LL(1). Para isso, alguns passos tiveram que ser realizados.

Na seção 2, apresentamos a gramática de estudo na forma convencional, traduzindo-a da forma BNF. Já na seção 3, discutimos sobre a remoção de recursão e de fatoração à esquerda, processos necessários para a transformação. Na seção 4, apresentamos o produto de todo esse processo: gramática já transformada em LL(1).

A implementação do exercício-programa é apresentado na seção **??**. Nela, a ferramenta utilizada é descrita, enfatizando entrada esperada e saídas geradas. Além disso, são apontados e expostos alguns trechos de códigos julgados pertinentes para este relatório. Por fim, apresentamos exemplos de saída do exercício-programa, assim como a descrição dos quatro programas solicitados (três já existentes, um novo).

As referências bibliográficas utilizadas para este relatório foram (AHO et al., 2006), (DELAMARO, 2004) e as vídeo aulas disponibilizadas até então.

## 1.1 Gramática de estudo: CC-2020-2

$PROGRAM$ → (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] )$^*$

# 2 Forma Convencional

No processo de transformação para forma convencional, realizamos a adição de alguns não-terminais, a fim de facilitar o processo. O resultado disso pode ser visualizado em 1.

| | |
|---|---|
| $PROGRAM$ | → STATEMENT \| FUNCLIST \| & |
| $FUNCLIST$ | → FUNCDEF FUNCLIST \| FUNCDEF |
| $FUNCDEF$ | → def ident(PARAMLIST) {STATELIST} |
| $DATATYPE$ | → int \| float \| string |
| $PARAMLIST$ | → DATATYPE ident, PARAMLIST \| DATATYPE ident \| & |
| $STATEMENT$ | → VARDECL; \| ATRIBSTAT; \| PRINTSTAT; \| READSTAT; \| RETURNSTAT; \| IFSTAT \| FORSTAT \| {STATELIST} \| break; \| ; |
| $VARDECL$ | → DATATYPE ident OPT_VECTOR |
| $OPT\_VECTOR$ | → [int_constant] OPT_VECTOR \| & |
| $ATRIB\_RIGHT$ | → EXPRESSION \| ALOCEXPRESSION \| FUNCCALL |
| $ATRIBSTAT$ | → LVALUE= ATRIB_RIGHT |
| $FUNCCALL$ | → ident(PARAMLISTCALL) |
| $PARAMLISTCALL$ | → ident, PARAMLISTCALL \| ident \| & |
| $PRINTSTAT$ | → print EXPRESSION |
| $READSTAT$ | → read LVALUE |
| $RETURNSTAT$ | → return |
| $IFSTAT$ | → if( EXPRESSION ) STATEMENT OPT_ELSE |
| $OPT\_ELSE$ | → (STATEMENT) \| & |
| $FORSTAT$ | → for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT |
| $STATELIST$ | → STATEMENT OPT_STATELIST |
| $OPT\_STATELIST$ | → STATELIST \| & |
| $ALLOCEXPRESSION$ | → new DATATYPE [NUMEXPRESSION] OPT_ALLOC_NUMEXP |
| $OPT\_ALLOC\_NUMEXP$ | → [NUMEXPRESSION] OPT_ALLOC_NUMEXP \| & |
| $EXPRESSION$ | → NUMEXPRESSION OPT_REL_OP_NUM_EXPR |
| $OPT\_REL\_OP\_NUM\_EXPR$ | → REL_OP NUMEXPRESSION \| & |
| $REL\_OP$ | → < \| > \| <= \| >= \| == \| != |
| $NUMEXPRESSION$ | → TERM REC_PLUS_MINUS_TERM |
| $REC\_PLUS\_MINUS\_TERM$ | → PLUS_OR_MINUS TERM REC_PLUS_MINUS_TERM \| & |
| $PLUS\_OR\_MINUS$ | → + \| - |
| $TERM$ | → UNARYEXPR REC_UNARYEXPR |
| $REC\_UNARYEXPR$ | → UNARYEXPR_OP TERM \| & |
| $UNARYEXPR\_OP$ | → * \| / \| % |
| $UNARYEXPR$ | → PLUS_OR_MINUS FACTOR \| FACTOR |
| $FACTOR$ | → int constant \| float constant \| string constant \| null \| LVALUE \| (NUMEXPRESSION) |
| $LVALUE$ | → ident OPT_ALLOC_NUMEXP |

Figura 1 – Forma convencional da gramática: ConvCC-2020-1

# 3 Recursão e Fatoração à Esquerda

A recursão à esquerda acontece quando existem produções como a seguir:

$S \quad \rightarrow$ Sa | &

Após o processo de passar a gramática de estudo à forma convencional, notamos que **não** existia nenhuma recursão à esquerda para ser resolvida.

Já a fatoração à esquerda é necessária quando há prefixos comuns em produções, como a seguir:

$S \quad \rightarrow$ sA | sB | sC

Foram encontradas algumas produções que precisaram ser fatoradas à esquerda. Não iremos citar todas, mas apresentamos, a seguir, um exemplo. Em 2, temos a gramática original, de 1, com necessidade de fatoração à esquerda. Em 3 temos a produção de exemplo fatorada à esquerda. Outra produção fatorada que pensamos que merece destaque consiste na `ATRIB_RIGHT`. Antes da fatoração, encontrávamos *ident* como FIRST comum à `EXPRESSION` e `FUNCCALL`. Após, conseguimos dar sequência e verificar a assertividade do teorema.

$$\begin{array}{ll} FUNCLIST & \rightarrow \text{FUNCDEF FUNCLIST} \mid \text{FUNCDEF} \\ FUNCDEF & \rightarrow \text{def ident(PARAMLIST) \{STATELIST\}} \end{array}$$

Figura 2 – Produção antes de ser fatorada à esquerda.

$$\begin{array}{ll} FUNCLIST & \rightarrow \text{FUNCDEF FUNCLISTAUX} \\ FUNCLISTAUX & \rightarrow \text{FUNCLIST} \mid \& \\ FUNCDEF & \rightarrow \text{def ident(PARAMLIST) \{STATELIST\}} \end{array}$$

Figura 3 – Produção após ser fatorada à esquerda.

O processo de fatoração foi realizado à mão pelos integrantes do grupo, assim como a análise em busca de recursão. A gramática final está descrita na seção 4.

# 4 LL(1)

A seguir, em 4, apresentaremos a forma final de nossa gramática, após passarmos ela para forma convencional e fatorar à esquerda. Além destes processos, trocamos a produção do if de *STATEMENT* para *{STATELIST}*, com intuito de remover ambiguidade.

Para realizar a verificação da propriedade de LL(1) para a gramática, foi utilizada uma ferramenta implementada pelo grupo. A ferramenta, detalhadada em 5, constrói os conjuntos de FIRST e FOLLOW da gramática, aplicando, posteriormente, o teorema de verificação.

Os conjuntos de FIRST e FOLLOW da gramática, caso seja necessário fazer a verificação manual, estão listados nos apêndices A e B. Entretanto, reiteramos que o Teorema é satisfeito.

| | |
|---|---|
| *PROGRAM* | → STATEMENT \| FUNCLIST \| & |
| *FUNCLIST* | → FUNCDEF FUNCLISTAUX |
| *FUNCLISTAUX* | → FUNCLIST \| & |
| *FUNCDEF* | → def ident(PARAMLIST) {STATELIST} |
| *PARAMLIST* | → DATATYPE ident PARAMLISTAUX<br>\| & |
| *PARAMLISTAUX* | → , PARAMLIST<br>\| & |
| *DATATYPE* | → int \| float \| string |
| *STATEMENT* | → VARDECL; \|<br>ATRIBSTAT; \| PRINTSTAT; \|<br>READSTAT; \| RETURNSTAT; \|<br>IFSTAT \| FORSTAT \|<br>{STATELIST} \| break; \| ; |
| *VARDECL* | → DATATYPE ident OPT_VECTOR |
| *OPT_VECTOR* | → [int_constant] OPT_VECTOR \| & |
| *ATRIBSTAT* | → LVALUE = ATRIB_RIGHT |
| *ATRIB_RIGHT* | → FUNCCALL_OR_EXPRESSION \|<br>ALOCEXPRESSION |
| *FUNCCALL_*<br>*OR_EXPRESSION* | → + FACTOR REC_UNARYEXPR<br>REC_PLUS_MINUS_TERM<br>OPT_REL_OP_NUM_EXPR \|<br>- FACTOR REC_UNARYEXPR<br>REC_PLUS_MINUS_TERM<br>OPT_REL_OP_NUM_EXPR \|<br>int_constant REC_UNARYEXPR<br>REC_PLUS_MINUS_TERM<br>OPT_REL_OP_NUM_EXPR \|<br>float_constant REC_UNARYEXPR<br>REC_PLUS_MINUS_TERM<br>OPT_REL_OP_NUM_EXPR \|<br>string_constant REC_UNARYEXPR<br>REC_PLUS_MINUS_TERM<br>OPT_REL_OP_NUM_EXPR \|<br>null REC_UNARYEXPR<br>REC_PLUS_MINUS_TERM<br>OPT_REL_OP_NUM_EXPR \|<br>( NUMEXPRESSION )<br>REC_UNARYEXPR<br>REC_PLUS_MINUS_TERM<br>OPT_REL_OP_NUM_EXPR \|<br>ident FOLLOW_IDENT |
| *FOLLOW_IDENT* | → OPT_ALLOC_NUMEXP<br>REC_UNARYEXPR<br>REC_PLUS_MINUS_TERM<br>OPT_REL_OP_NUM_EXPR \|<br>( PARAMLISTCALL ) |
| *FUNCCALL* | → ident(PARAMLISTCALL) |
| *PARAMLISTCALL* | → ident PARAMLISTCALLAUX \| & |
| *PARAMLISTCALLAUX* | → , PARAMLISTCALL \| & |

| | |
|---|---|
| $PRINTSTAT$ | → print EXPRESSION |
| $READSTAT$ | → read LVALUE |
| $RETURNSTAT$ | → return |
| $IFSTAT$ | → if (EXPRESSION) {STATELIST} OPT_ELSE |
| $OPT\_ELSE$ | → else {STATELIST} \| & |
| $FORSTAT$ | → for (ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT |
| $STATELIST$ | → STATEMENT OPT_STATELIST |
| $OPT\_STATELIST$ | → STATELIST \| & |
| $ALLOCEXPRESSION$ | → new DATATYPE [NUMEXPRESSION] OPT_ALLOC_NUMEXP |
| $OPT\_ALLOC\_NUMEXP$ | → [NUMEXPRESSION] OPT_ALLOC_NUMEXP \| & |
| $EXPRESSION$ | → NUMEXPRESSION OPT_REL_OP_NUM_EXPR |
| $OPT\_REL\_OP\_NUM\_EXPR$ | → REL_OP NUMEXPRESSION \| & |
| $REL\_OP$ | → < \| > \| <= \| >= \| == \| != |
| $NUMEXPRESSION$ | → TERM REC_PLUS_MINUS_TERM |
| $REC\_PLUS\_MINUS\_TERM$ | → PLUS_OR_MINUS TERM REC_PLUS_MINUS_TERM \| & |
| $PLUS\_OR\_MINUS$ | → + \| - |
| $TERM$ | → UNARYEXPR REC_UNARYEXPR |
| $REC\_UNARYEXPR$ | → UNARYEXPR_OP TERM \| & |
| $UNARYEXPR\_OP$ | → * \| / \| % |
| $UNARYEXPR$ | → PLUS_OR_MINUS FACTOR \| FACTOR |
| $FACTOR$ | → int_constant \| float_constant \| string_constant \| null \| LVALUE \| (NUMEXPRESSION) |
| $LVALUE$ | → ident OPT_ALLOC_NUMEXP |

Figura 4 – Gramática final: ConvCC-2020-1

# 5 Implementação

## 5.1 Descrição das ferramentas utilizadas

Para esta entrega, foram implementados o gerador de conjuntos FIRST e FOLLOW, descrito em 5.1.1, o testador de Teorema LL(1), descrito em 5.1.2 e a análise sintática, descrita em 5.1.2.

### 5.1.1 Gerador de conjuntos FIRST e FOLLOW

A geração dos conjuntos FIRST e FOLLOW é realizada pela classe `CfgProcessor`, localizada em `src/utils/cfg_processor.py`. A função responsável por isso, `load_cfg`, é descrita abaixo.

```
def load_cfg(self, cfg: Cfg):
    self.cfg = cfg
    first = {i: set() for i in self.cfg.non_terminals}
    first.update((i, {i}) for i in self.cfg.terminals)
    first[self.__empty_symbol] = {self.__empty_symbol}

    follow = {i: set() for i in self.cfg.non_terminals}
    follow[self.__empty_symbol] = {self.__stack_base_symbol}
    follow[self.cfg.start_symbol] = {self.__stack_base_symbol}

    epsilon = {self.__empty_symbol}

    while True:
        updated = False

        for prod in self.cfg.productions:
            # Calculate FIRST
            for symbol in prod.body:
                updated |= union(first[prod.head], first[symbol])

                if symbol not in epsilon:
                    break

            else:
```

```
                    first[prod.head] |= {self.__empty_symbol}
                    updated |= union(epsilon, {prod.head})

                # Calcualte FOLLOW
                aux = follow[prod.head]
                for symbol in reversed(prod.body):
                    if symbol in follow:
                        updated |= union(follow[symbol], aux)

                    if symbol in epsilon:
                        aux = aux.union(first[symbol])
                    else:
                        aux = first[symbol]

            if not updated:
                break

        self.__first = first
        self.__follow = follow
        self.__epsilon = epsilon
```

### 5.1.2 Testador de Teorema LL(1)

O testador d Teorema de LL(1) também foi implementado na classe `CfgProcessor` (`src/utils/cfg_processor.py` ). O processo é composto por diversas funções auxiliares. Após a criação dos conjuntos FIRST e FOLLOW, é invocada a função `is_ll1`, descrita abaixo.

```
    def is_ll1(self) -> bool:
        """Check if cfg is LL(1) apply the theorem
            for A -> "alpha" | "beta",
            1 - First(alpha) intersect First(beta) = {}
            2 - If beta ->* &, First(alpha) intersect Follow(A) = {}
                If alpha ->* &, First(beta) intersect Follow(A) = {}
        """
        for nt in self.cfg.non_terminals:
            if not self.__apply_theorem_all_prods_of(nt):
                return False

        return True
```

Esta função agrega as duas cláusulas do teorema, descritos em seu comentário. A função `__theorem_first_clause` é responsável por testar a primeira cláusula do teorema, enquanto `__theorem_second_clause`, a segunda.

### 5.1.3 Tabela de análise

A geração da tablea de análise sintática é feita na classe CfgProcessor, utilizando a estrutura de dados SyntaticAnalyserMatrix definida em `src/utils/data_structures.py`. Através dessa estrutura, a implementação do parser resultou em um código semelhante ao pseudo-código estudado em aula.

```
class SyntaticAnalyserMatrix:
    """Syntatic Analyser Matrix"""

    def __init__(self, terminals: Set[str], non_terminals: Set[str],
                 stack_base: str = '$'):
        cols = terminals | {stack_base}
        rows = non_terminals

        self.__matrix: Dict[str, Dict[str, Optional[Production]]] = {}

        for row in rows:
            self.__matrix[row] = {}
            for col in cols:
                self.__matrix[row][col] = None

    def set_prod(self, non_terminal: str, terminal: str, prod: Production):
        curr_element = self.__matrix[non_terminal][terminal]
        if curr_element is not None \
                and curr_element != prod:
            raise ValueError('Table cell cannot be set twice!')
        self.__matrix[non_terminal][terminal] = prod

    def get_prod(self, non_terminal: str, terminal: str
                 ) -> Optional[Production]:
        return self.__matrix[non_terminal][terminal]

    def get_matrix(self) -> Dict[str, Dict[str, Optional[Production]]]:
        return self.__matrix
```

Para efetivamente preencher a matriz, a já mencionada classe CfgProcessor implementa o algoritmo abaixo, que completa os campos a partir dos conjuntos FIRST e FOLLOW da gramática LL(1)

```
def generate_matrix(self) -> SyntaticAnalyserMatrix:
    """Generate the analyser matrix, if the grammar is LL(1)"""
    if not self.is_ll1():
        logger.error('Cannot generate matrix for non LL(1) grammar')
        raise ValueError

    mat = SyntaticAnalyserMatrix(
        self.cfg.terminals, self.cfg.non_terminals)

    for prod in self.cfg.productions:
        body_first = self.first_of_prod_body(prod.body)
        for terminal in body_first - {self.__empty_symbol}:
            mat.set_prod(prod.head, terminal, prod)

        if self.__empty_symbol in body_first:
            for terminal in self.follow(prod.head):
                mat.set_prod(prod.head, terminal, prod)

    return mat
```

### 5.1.4 Analisador Sintático

A análise sintática é realizada dentro da classe `CC20202Parser`, localizada em *src/compiler/parser/CC20202_parser.py*. Essa classe utiliza as funções previamente descritas em 5.1.1 e 5.1.2 para construir o analisador sintático, utilzando a gramática de referência, localizada em `src/grammar/ConvCC-2020-2.csf`. Após isso, é realizada a análise sintática do código, de fato. A função utilizada é a `parse`, descrita abaixo.

```
def parse(self, symbols_table: SymbolTableType,
          tokens: List[LexToken]) -> Tuple[bool, Optional[LexToken]]:
    """Validate symbols and update symbols_table"""
    stack = deque()

    stack.append('$')
    stack.append(self.start_symbol)
```

```
        for token in tokens + [STACK_BOT_TOKEN]:
            token_terminal = TYPE_TO_TERMINAL_MAP[token.type]
            while True:
                # Terminal on top of stack and on code pointer
                # are equals, pop the stack and move the pointer
                if token_terminal == stack[-1]:
                    stack.pop()
                    break

                # Get production to be applied
                try:
                    prod = self.mat.get_prod(stack[-1], token_terminal)
                except KeyError:
                    return (False, token)

                # Reconize syntatic error
                if prod is None:
                    return (False, token)

                # Remove the top of the stack
                stack.pop()

                # Stack the symbols from corresponding production
                for symbol in reversed(prod.body):
                    if symbol != self.__empty_symbol:
                        stack.append(symbol)

        # If something other than the stack bottom is in the stack
        if len(stack) > 1:
            return (False, token)

        return (True, None)
```

## 5.2 Exemplo de saída

O funcionamento total da ferramenta é descrito na função main, localizada em `src/run.py`.

```
def main(filepath: str):
    """Main function"""
    with open(filepath) as f:
```

```
        source_code = f.read()

    tokens = []
    lexer.input(source_code)
    while True:
        try:
            token = lexer.token()
        except InvalidTokenError as exp:
            logger.error(exp)
            exit(1)

        if not token:
            break
        else:
            tokens.append(token)

    logger.info('Total tokens: %s' % len(tokens))
    # logger.info('Lista de tokens:')
    # for token in tokens:
    #     print('<%s, %s>' % (token.type, token.value))

    symbols_table = generate_symbol_table(tokens)
    logger.info('Imprimindo tabela de símbolos...')

    # Print table
    header = [
        'var_name',
        'token_index',
        'type',
        'line_declared',
        'lines_referenced']

    row_print = "{:<15} " * len(header)
    print(row_print.format(*header))
    for _, symbol_row in symbols_table.items():
        print(row_print.format(
            str(symbol_row.var_name),
            str(symbol_row.token_index),
            str(symbol_row.type),
```

```
            str(symbol_row.line_declared),
            str(symbol_row.lines_referenced)))

    logger.info('Running parser for list of tokens...')
    success, fail_token = parser.parse(symbols_table=symbols_table,
                                       tokens=tokens)

    if not success:
        line = linecache.getline(filepath, fail_token.lineno)
        logger.info('Invalid sintax at line %s:\n\t%s' %
                    (fail_token.lineno, line.strip()))
        logger.info('Token: %s' % fail_token)
        logger.error('Syntatic error detected!')
    else:
        logger.info('Syntatic analysis completed with success!')
```

Primeiramente, é invocado o analisador léxico, desenvolvido para a primeira entrega. Não houveram mudanças em relação à sua entrada, mas fizemos melhorias na tabela de símbolos gerada por ele. Após a análise léxica, passamos ao analisador sintático implementado a tabela de símbolos e a lista de tokens. Caso não exista erro sintático, uma mensagem de sucesso é emitida, além da tabela de símbolos gerada anteriormente. Caso contrário, uma mensagem de erro é lançada, apresentando a linha de onde aconteceu.

Exemplo de código sem erro sintático, após análise:

```
 Executing...
2021-04-03 18:36:21.342 | INFO     | __main__:main:33 - Total tokens: 162
2021-04-03 18:36:21.342 | INFO     | __main__:main:39 - Imprimindo tabela de símbolos...
var_name        token_index     type            line_declared   lines_referenced
x               3               Unknow          3               [7, 10, 11, 11, 12, 13]
z               6               Unknow          4               [12, 13]
i               9               Unknow          5               [9, 9, 9, 9, 24, 27, 27, 27, ...
max             12              Unknow          6               [8, 9]
y               69              Unknow          22              [25, 29, 37]
j               72              Unknow          23              [26, 29, 30, 30]
2021-04-03 18:36:21.342 | INFO     | __main__:main:59 - Running parser for list of tokens...
2021-04-03 18:36:21.342 | INFO     | __main__:main:70 - Syntatic analysis completed with success!
```

Exemplo de código com erro sintático proposital:

```
 Executing...
2021-04-03 18:34:13.378 | INFO     | __main__:main:33 - Total tokens: 163
2021-04-03 18:34:13.379 | INFO     | __main__:main:39 - Imprimindo tabela de símbolos...
var_name        token_index     type            line_declared   lines_referenced
x               3               Unknow          3               [7, 10, 11, 11, 12, 13]
z               6               Unknow          4               [12, 13]
i               9               Unknow          5               [9, 9, 9, 9, 24, 27, 27, 27, ...
max             12              Unknow          6               [8, 9]
y               69              Unknow          22              [25, 29, 37, 39]
j               72              Unknow          23              [26, 29, 30, 30]
2021-04-03 18:34:13.379 | INFO     | __main__:main:59 - Running parser for list of tokens...
2021-04-03 18:34:13.379 | INFO     | __main__:main:65 - Invalid sintax at line 39:
        return y;
2021-04-03 18:34:13.379 | INFO     | __main__:main:67 - Token: LexToken(IDENT,'y',39,521)
2021-04-03 18:34:13.379 | ERROR    | __main__:main:68 - Syntatic error detected!
```

## 5.3 Descrição dos programas escritos

Os quatro programas seguindo os padrões solicitados estão localizados na pasta *examples*, junto com os exemplos disponibilizados pelo professor. Além dos três programas exigidos no trabalho anterior (`example/bhaskara.ccc`, `example/math.ccc` e `example/geometry.ccc`), foi adicionado o código para `example/vinho.ccc`.

O programa `example/bhaskara.ccc` apresenta duas funções, além da `main`: `bhaskara` e `calculate_delta`. A função `calculate_delta` recebe como parâmetros três variáveis de ponto flutuante e calcula o delta, mostrando na tela o seu resultado, também. A função `bhaskara` resolve a equação quadrática, apresentando erro se o primeiro parâmetro for 0 e apresentando na tela os resultados. A função `main` chama as funções repetidamente, mas com argumentos diferentes em todas elas, com intuito de testar os casos possíveis.

O programa `example/math.ccc` visa representar o que seria a implementação de uma biblioteca de ferramentas matemáticas, dadas as restrições da linguagem. A função `gcd` é o cálculo do maior divisor comum entre dois números iterativamente, usando o algoritmo de Euclides, `is_prime` aplica um algoritmo de identificação de números primos, `pow` calcula o primeiro argumento elevado a potência do segundo, `factorial` calcula o fatorial de um número. A função `main` executa algumas chamadas sobre as funções previamente definidas, para fins de demonstração.

O programa `example/geometry.ccc` apresenta algumas funções geométricas. Para triângulos, a função `triangle_area` que calcula sua área dada a base e a altura e a função `form_triangle` que, dado o comprimento de três segmentos de reta, calcula se é possível formar um triângulo com eles. Para círculos, a função `calc_circle_circumference` calcula a circunferência, e a função `calc_circle_area` a área. Ambas recebem como entrada o raio do círculo em ponto flutuante. Por fim, a função `calc_square_area` calcula a área de um quadrado dado o comprimento do lado. A função `main` chama cada uma das funções geométricas para demonstrar seu funcionamento.

O programa `example/vinho.ccc` simula um decisor para combinar vinhos e comidas. O programa possui uma função `combinar` que recebe duas comidas dentre as opções disponíveis, e imprime na tela uma lista com o nome dos vinhos suportados pelo programa juntamente com uma pontuação de acordo com quão adequada é a harmonização do vinho com as comidas em questão. A combinação foi implementada baseada no diagrama presente na figura 5.

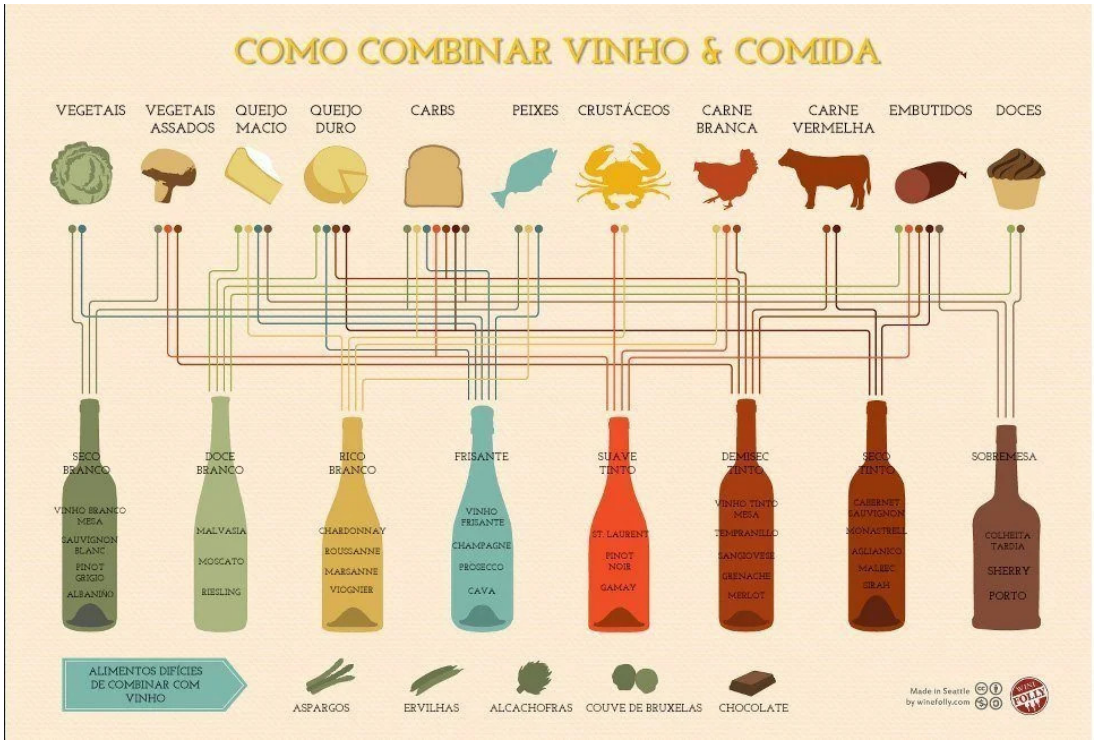


Figura 5 – Guia de harmonização de vinhos e comidas. Fonte: Almanaque SOS.

# APÊNDICE A – Conjuntos FIRST

FIRST(REC_UNARYEXPR): {'&', '*', '/', '%'}
FIRST(IFSTAT): {'if'}
FIRST(PARAMLIST): {'&', 'float', 'string', 'int'}
FIRST(FUNCCALL_OR_EXPRESSION): {'int_constant', 'string_constant', '(', 'null', 'float_constant', 'ident'}
FIRST(LVALUE): {'ident'}
FIRST(PROGRAM): {'def', 'read', 'float', 'string', '{', 'break', '&', 'print', 'return', 'if', 'int', ';', 'for', 'ident'}
FIRST(REL_OP): {'>=', '>', '==', '<', '!=', '<='}
FIRST(ATRIBSTAT): {'ident'}
FIRST(VARDECL): {'float', 'string', 'int'}
FIRST(NUMEXPRESSION): {'int_constant', 'null', '-', '+', 'string_constant', '(', 'ident', 'float_constant'}
FIRST(FUNCCALL): {'ident'}
FIRST(REC_PLUS_MINUS_TERM): {'-', '&', '+'}
FIRST(ALLOCEXPRESSION): {'new'}
FIRST(PARAMLISTCALL): {'&', 'ident'}
FIRST(TERM): {'int_constant', 'null', '-', '+', 'string_constant', '(', 'ident', 'float_constant'}
FIRST(ATRIB_RIGHT): {'int_constant', 'new', 'null', 'string_constant', '(', 'ident', 'float_constant'}
FIRST(FOLLOW_IDENT): {'<=', '-', '&', '+', '*', '(', '==', '<', '%', '!=', '[', '>=', '>', '/'}
FIRST(STATELIST): {'read', 'float', 'string', '{', 'break', 'print', 'return', 'if', 'int', ';', 'for', 'ident'}
FIRST(PARAMLISTAUX): {'&', ','}
FIRST(EXPRESSION): {'int_constant', 'null', '-', '+', 'string_constant', '(', 'ident', 'float_constant'}
FIRST(OPT_STATELIST): {'read', 'float', 'string', '{', 'break', '&', 'print', 'return', 'if', 'int', ';', 'for', 'ident'}
FIRST(RETURNSTAT): {'return'}
FIRST(DATATYPE): {'float', 'string', 'int'}
FIRST(FORSTAT): {'for'}
FIRST(FUNCLISTAUX): {'def', '&'}
FIRST(OPT_ELSE): {'&', 'else'}
FIRST(PRINTSTAT): {'print'}

FIRST(FUNCDEF): {'def'}
FIRST(READSTAT): {'read'}
FIRST(UNARYEXPR): {'int_constant', 'null', '-', '+', 'string_constant', '(', 'ident', 'float_constant'}
FIRST(OPT_REL_OP_NUM_EXPR): {'==', '<', '!=', '<=', '&', '>=', '>'}
FIRST(OPT_ALLOC_NUMEXP): {'[', '&'}
FIRST(PARAMLISTCALLAUX): {'&', ','}
FIRST(FUNCLIST): {'def'}
FIRST(OPT_VECTOR): {'[', '&'}
FIRST(FACTOR): {'int_constant', 'string_constant', '(', 'null', 'float_constant', 'ident'}
FIRST('-'): set() FIRST(UNARYEXPR_OP): {'/', '%', '*'} FIRST('+'): set() FIRST(PLUS_OR_MIN {'-', '+'}
FIRST(STATEMENT): {'if', 'read', 'float', 'string', '{', 'int', 'break', ';', 'print', 'for', 'ident', 'return'}
FIRST(else): {'else'}
FIRST(,): {','}
FIRST(read): {'read'}
FIRST(float): {'float'}
FIRST({): {'{'}
FIRST(break): {'break'}
FIRST(+): {'+'}
FIRST((): {'('}
FIRST(return): {'return'}
FIRST(<): {'<'}
FIRST(>=): {'>='}
FIRST(>): {'>'}
FIRST(for): {'for'}
FIRST(/): {'/'}
FIRST(ident): {'ident'}
FIRST(def): {'def'}
FIRST(string): {'string'}
FIRST(=): {'='}
FIRST(<=): {'<='}
FIRST(print): {'print'}
FIRST(-): {'-'}
FIRST(*): {'*'}
FIRST(float_constant): {'float_constant'}
FIRST(int_constant): {'int_constant'}
FIRST()): {')'}

FIRST(if): {'if'}
FIRST(==): {'=='}
FIRST(}): {'}'}
FIRST(new): {'new'}
FIRST(null): {'null'}
FIRST(%): {'%'}
FIRST(int): {'int'}
FIRST(!=): {'!='}
FIRST(;): {';'}
FIRST([): {'['}
FIRST(]): {']'}
FIRST(string_constant): {'string_constant'}
FIRST(&): {'&'}

# APÊNDICE B – Conjuntos FOLLOW

FOLLOW(NUMEXPRESSION): {'>=', '<=', ';', ')', ']', '!=', '>', '==', '<'}
FOLLOW('+'): {'(', 'null', 'string_constant', 'int_constant', 'ident', 'float_constant'}
FOLLOW(STATEMENT): {';', '$', 'if', 'string', 'read', 'ident', 'break', '}', 'print', '{', 'int', 'return', 'for', 'float'}
FOLLOW(READSTAT): {';'}
FOLLOW(RETURNSTAT): {';'}
FOLLOW(VARDECL): {';'}
FOLLOW(UNARYEXPR_OP): {'(', 'null', 'string_constant', 'int_constant', '-', '+', 'ident', 'float_constant'}
FOLLOW(LVALUE): {'>=', '<=', ';', '!=', '/', '+', '&', '==', '<', '=', ')', ']', '-', '>', '*'}
FOLLOW(PRINTSTAT): {';'}
FOLLOW(PLUS_OR_MINUS): {'(', 'null', 'string_constant', 'int_constant', '-', '+', 'ident', 'float_constant'}
FOLLOW(FOLLOW_IDENT): {';', ')'}
FOLLOW(FORSTAT): {';', '$', 'if', 'string', 'read', 'ident', 'break', '}', 'print', '{', 'int', 'return', 'for', 'float'}
FOLLOW(OPT_ALLOC_NUMEXP): {'>=', '<=', ';', '!=', '/', '+', '&', '==', '<', '=', ')', ']', '-', '>', '*'}
FOLLOW(FUNCCALL): set()
FOLLOW(ALLOCEXPRESSION): {';', ')'}
FOLLOW(STATELIST): {'}'}
FOLLOW(DATATYPE): {'[', 'ident'}
FOLLOW(OPT_REL_OP_NUM_EXPR): {';', ')'}
FOLLOW(ATRIBSTAT): {';', ')'}
FOLLOW(PROGRAM): {'$'}
FOLLOW(FUNCDEF): {'def', '$'}
FOLLOW(REC_PLUS_MINUS_TERM): {'>=', '<=', ';', ')', ']', '!=', '>', '==', '<'}
FOLLOW(REC_UNARYEXPR): {'>=', '<=', ';', '!=', '+', '==', '<', ')', ']', '-', '>'}
FOLLOW(UNARYEXPR): {'>=', '<=', ';', '!=', '/', '+', '&', '==', '<', ')', ']', '-', '>', '*'}
FOLLOW(IFSTAT): {';', '$', 'if', 'string', 'read', 'ident', 'break', '}', 'print', '{', 'int', 'return', 'for', 'float'}
FOLLOW(FACTOR): {'>=', '<=', ';', '!=', '/', '+', '&', '==', '<', ')', ']', '-', '>', '*'}
FOLLOW(EXPRESSION): {';', ')'}
FOLLOW(OPT_STATELIST): {'}'}
FOLLOW(FUNCLIST): {'$'}

FOLLOW(PARAMLISTAUX): {')'}
FOLLOW('-'): {'(', 'null', 'string_constant', 'int_constant', 'ident', 'float_constant'}
FOLLOW(PARAMLIST): {')'}
FOLLOW(ATRIB_RIGHT): {';', ')'}
FOLLOW(REL_OP): {'(', 'null', 'string_constant', 'int_constant', '-', '+', 'ident', 'float_constant'}
FOLLOW(PARAMLISTCALLAUX): {')'}
FOLLOW(TERM): {'>=', '<=', ';', '!=', '+', '==', '<', ')', ']', '-', '>'}
FOLLOW(OPT_ELSE): {';', '$', 'if', 'string', 'read', 'ident', 'break', '}', 'print', '{', 'int', 'return', 'for', 'float'}
FOLLOW(FUNCCALL_OR_EXPRESSION): {';', ')'}
FOLLOW(OPT_VECTOR): {';'}
FOLLOW(PARAMLISTCALL): {')'}
FOLLOW(FUNCLISTAUX): {'$'}
FOLLOW(&): {'>=', '<=', ';', '$', 'if', '!=', '+', 'read', '&', '==', '<', 'break', '=', ')', ']', '-', 'int', 'return', '*', 'for', 'string', '/', 'ident', '}', 'print', '{', '>', 'float'}

# Referências

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

DELAMARO, M. *Como Construir um Compilador Utilizando Ferramentas Java*. Novatec, 2004. ISBN 9788575220559. Disponível em: <https://books.google.com.br/books?id=_MpSXwAACAAJ>.