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
Maio de 2021

Sumário

1 Introdução

Este relatório apresenta a descrição do trabalho realizado para a construção de um Analisador Sintático e Gerador de Código Intermediário para a gramática **ConvCC-2020-1**, originalmente concebida como descrita em 1.1, e com as modificações realizadas nos trabalhos anteriores conforma apresentado em 1.2.

Para a análise semântica, foram produzidos esquemas de definição dirigida por sintaxe (SDDs) L-Atribuídas e tradução dirigida por sintaxe (SDT), tanto para construção da árvore de expressão quanto para declaração de variáveis. Além disso, foram definidas regras para a definição de tipos (operações aceitas por nossa linguagem de acordo com o tipo dos operandos), declaração de variáveis por escopo e a detecção de que comandos `break` sempre estão contidos no escopo de um comando de repetição. Depois disso, foi implementada a geração de código intermediário para a gramática **ConvCC-2020-2** utilizando código de três endereços.

A implementação do exercício-programa é apresentado na seção 9. 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). Além disso, foram adicionados três novos programas com exemplo dos erros a serem verificados.

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

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])*

Figura 1 – Gramática ConvCC-2020-1

1.2 Gramática CC-2020-2 Modificada

No trabalho anterior, a gramática **ConvCC-2020-2** foi passada para a forma convencional e fatorar à esquerda. Além destes processos, trocamos a produção do if de *STATEMENT* para *{STATELIST}*, com intuito de remover ambiguidade. Para esse trabalho, também foi adicionada a necessidade do uso de chaves ao declarar um comando de repetição `for`, com o objetivo de simplificar a implementação.

PROGRAM	→ STATEMENT | FUNCLIST | &
FUNCLIST	→ FUNCDEF FUNCLISTAUX
FUNCLISTAUX	→ FUNCLIST | &
FUNCDEF	→ def ident(PARAMLIST) {STATELIST}
PARAMLIST	→ DATATYPE ident PARAMLISTAUX | &
PARAMLISTAUX	→ , PARAMLIST | &
DATATYPE	→ int | float | string
STATEMENT	→ VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; | RETURNSTAT; | IFSTAT | FORSTAT | {STATELIST} | break; | ;
VARDECL	→ DATATYPE ident OPT_VECTOR
OPT_VECTOR	→ [int_constant] OPT_VECTOR | &
ATRIBSTAT	→ LVALUE = ATRIB_RIGHT
ATRIB_RIGHT	→ FUNCCALL_OR_EXPRESSION | ALOCEXPRESSION
*FUNCCALL_* *OR_EXPRESSION*	→ + FACTOR REC_UNARYEXPR REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR | - FACTOR REC_UNARYEXPR REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR | int_constant REC_UNARYEXPR REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR | float_constant REC_UNARYEXPR REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR | string_constant REC_UNARYEXPR REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR | null REC_UNARYEXPR REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR | (NUMEXPRESSION) REC_UNARYEXPR REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR | ident FOLLOW_IDENT
FOLLOW_IDENT	→ OPT_ALLOC_NUMEXP REC_UNARYEXPR REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR | (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) { STATELIST }
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 2 – Gramática ConvCC-2020-1 modificada

2 Construção da Árvore de Expressão: EXPA

Para a construção da árvore de expressão, primeiro foram separadas da gramática **ConvCC-2020-2** as produção que derivam expressões aritméticas, para compor a gramática chamada de **EXPA**, apresentada abaixo:

$NUMEXPRESSION$	→ 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 |
OPT_ALLOC_NUMEXP	→ [NUMEXPRESSION]
$LVALUE$	→ ident OPT_ALLOC_NUMEXP

A partir dela foi criada a SDD L-atribuída cujo o objetivo é construir uma árvore de expressão. Ela utiliza a função `new_node` para inserir elementos na árvore, recebendo como argumento o nodo a sua esquerda e direita, seu valor ou operação que está sendo realizada. A prova de que essa SDD é L-atribuída está presente na seção 8. A SDD completa é apresentada no apêndice A, enquanto a SDT é apresentada no apêndice B. Nestes apêndices, já estão apresentados alguns detalhes da implementação, inclusive, como a questão de inserção nas árvores, verificação e resultado dos tipos.

No projeto encaminhado, a SDD está no arquivo `src/grammar/EXPA.sdd`, enquanto a SDT está no arquivo `src/grammar/EXPA.sdt`. A sua implementação ocorre no arquivo `src/compiler/semantic/CC20202_semantic.py`.

3 Inserção do tipo na Tabela de Símbolos: DEC

Para a construção da árvore de expressão, primeiro foram separadas da gramática **ConvCC-2020-2** as produção que derivam expressões aritméticas, para compor a gramática chamada de **DEC**, apresentada abaixo:

$VARDECL$	→ DATATYPE ident OPT_VECTOR
$DATATYPE$	→ int | float | string
OPT_VECTOR	→ [int_constant] OPT_VECTOR | &
$FUNCDEF$	→ def ident (PARAMLIST) STATELIST
$PARAMLIST$	→ DATATYPE ident PARAMLISTAUX | &
$PARAMLISTAUX$	→ , PARAMLIST | &

A partir dela, foi construída a SDD, que utiliza uma tabela de símbolos com a função `new_table_entry`, que recebe o identificador, seu tipo, o número de dimensões com o tamanho e a linha na qual foi encontrado. O número de dimensões diz respeito a arrays multidimensionais. Uma variável comum terá esse vetor vazio, um array `a[2]` terá a entrada [2], ou seja, uma dimensão de tamanho dois e uma matrix `m[3][4]` terá a entrada [3, 4], pois sua primeira dimensão possui tamanho 3 e a segunda tamanho 4. A prova de que a SDD é L-atribuída está disponível no seção 8.

A sua implementação ocorre no arquivo `src/compiler/semantic/CC20202_semantic.py`.

3.1 SDD

Segue a SDD completa, disponível no arquivo `src/grammar/DEC.sdd`:

```
new_table_entry(
    identifier_label: str,
    datatype: str,
    dimesions: List[int],
    line: int
)
concat(l1: List[Any], l2: List[Any])

production:
    VARDECL : DATATYPE "ident" OPT_VECTOR
rules:
```

```
    VARDECL.sin = new_table_entry(
        ident.text,
        DATATYPE.type,
        OPT_VECTOR.sin,
        ident.lineno
    )

production:
    DATATYPE : "int"
rules:
    DATATYPE.type = "int"

production:
    DATATYPE : "float"
rules:
    DATATYPE.type = "float"

production:
    DATATYPE : "string"
rules:
    DATATYPE.type = "string"

production:
    OPT_VECTOR : "[" "int_constant" "]" \
rules:
    OPT_VECTOR.sin = concat([int_constant.value], OPT_VECTOR1.sin)

production:
    OPT_VECTOR : &
rules:
    OPT_VECTOR.sin = []
```

3.2 SDT

Por fim, podemos transformar a SDD em SDT imbuindo as regras definidas às próprias produções, obtendo o seguinte resultado:

```
production:
    VARDECL : DATATYPE "ident" OPT_VECTOR { VARDECL.sin = new_table_entry(
    ident.text, DATATYPE.type, OPT_VECTOR.sin, ident.lineno) }
```

```
production:
    DATATYPE : "int" {DATATYPE.type = "int"}

production:
    DATATYPE : "float" { DATATYPE.type = "float" }

production:
    DATATYPE : "string" { DATATYPE.type = "string" }

production:
    OPT_VECTOR : "[" "int_constant" "]" OPT_VECTOR1 { OPT_VECTOR.sin =
    concat([int_constant.value], OPT_VECTOR1.sin) }

production:
    OPT_VECTOR : & { OPT_VECTOR.sin = [] }
```

4 Verificação de tipos

A verificação de tipos consiste em validar que as entradas de uma expressão são válidas de acordo com a regra definida para as operações entre tipos diferentes. Em nossa implementação, consideramos válidas as expressões que seguem possuem os seguintes tipos e operações:

- **int com int** – soma, subtração, multiplicação, divisão e módulo.
- **float com float** – soma, subtração, multiplicação e divisão.
- **int com float** – soma, subtração, multiplicação e divisão.
- **string com string** – soma (concatenação).

A verificação foi implementada através de uma função `check_type` que possui um dicionário que define quais são as operações válidas. Essa função recebe dois nodos da árvore de expressão, a operação que está sendo realizada entre eles e a linha em que a operação está sendo realizada. É realizado o teste verificando se a tupla contendo o atributo `result_type` de cada um dos nodos e o símbolo da operação possui uma entrada no dicionário. Caso essa verificação retorne `None` significa que a combinação de tipos com a operação não é aceita na nossa linguagem, e um erro é retornado para o usuário indicando a linha em que a operação se encontra. As operações válidas, junto com a implementação dessa função, são apresentadas a seguir.

```
def check_type(left: Node, right: Node, operation: str, lineno: int) -> str:
    valids = {
        ('string', '+', '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'{left.result_type},{right.result_type},{lineno}')

    return result
```

Foi adiciona um programa exemplo com intuito de mostrar esse erro sendo capturado. Maiores detalhes sobre o conteúdo e a saída são apresentados na seção 9.

5 Declaração de variáveis de escopo

A declaração de variáveis dentro de um mesmo escopo não pode ocorrer para o mesmo identificador com dois tipos diferentes, isto é, caso tenha sido declarada a variável `int a`, não deve ser possível declarar a variável `string a` dentro do mesmo escopo.

Para tratar dos escopos, foi implementada uma estrutura de dados `scope`, que possui uma tabela de símbolos, uma lista de escopos internos, uma variável booleana que indica se o escopo é de um comando de repetição e seu escopo externo. A função para adicionar uma entrada na tabela de símbolos é a seguinte:

```
def add_entry(self, entry: TableEntry):
    is_present, line_declared = self.var_already_present(
        entry.identifier_label)
    if is_present:
        raise VariableAlreadyDeclaredInScopeError(line_declared)
    self.table.append(entry)
```

Todas as vezes que uma entrada nova tenta ser inserida, é verificado se ela já está presente, retornando um valor booleano que indica se ela já estava presente ou não e a linha na qual ela foi declarada. Caso ela já estivesse na tabela, um erro é disparado e o compilador mostra para o usuário em qual linha houve a tentativa de instanciar uma variável duplicada. A verificação `var_already_present` é realizada simplesmente buscando se o identificador recebido já possuia alguma entrada na tabela, que é implementada com um dicionário.

Foi adiciona um programa exemplo com intuito de mostrar esse erro sendo capturado. Maiores detalhes sobre o conteúdo e a saída são apresentados na seção 9.

6 Comandos dentro do escopo

A implementação de nossa linguagem deve considerar erro a presença de um `break` sem escopo de comando de repetição. Conforme descrito na seção anterior, todo escopo possui um atributo booleano que indica se ele é o escopo de um comando de repetição ou não. Além disso, o escopo guarda a referência para seu escopo externo, ou seja, o escopo que o engloba. Com essas duas ferramentas, é possível identificar se o break não possui escopo simplesmente percorrendo a cadeia de escopos atuais e verificando se pelo menos um deles é de um comando de repetição. Na nossa implementação, isso foi realizado da seguinte maneira:

```
while True:
    if current_scope.is_loop:
        break

    current_scope = current_scope.upper_scope

    if current_scope is None:
        raise BreakWithoutLoopError(p.lineno(2))
```

Ao chegar no escopo mais externo do programa, a variável `current_scope` recebe o valor None, o que resulta no disparo do erro `BreakWithoutLoopError`, que informa ao usuário em qual linha há a declaração de um break sem escopo de comando de repetição.

Foi adiciona um programa exemplo com intuito de mostrar esse erro sendo capturado. Maiores detalhes sobre o conteúdo e a saída são apresentados na seção 9.

7 Geração de Código Intermediário

A geração de código intermediário foi executada por meio de uma SDT implementada diretamente com o `framework` da biblioteca descrita na seção 9 e presente no arquivo `/src/compiler/gci/CC20202_gci.py`. A abordagem utilizada foi fazer com que produções mais próximas das folhas da árvore de derivação criassem os inícios dos códigos iniciais, e os nodos acima concatenem a estes o código necessário para representar a operação em andamento, de forma que na raíz da árvore tenha como atributo o código completo gerado.

O código abaixo foi extraído do arquivo citado e faz a ordenação das ações para a criação de um desvio de fluxo `if/else`.

```
1 def p_ifstat(p: yacc.YaccProduction):
2     """IFSTAT : IF LPAREN EXPRESSION RPAREN LBRACKETS STATELIST RBRACKETS
OPT_ELSE"""
3     cond_temp_var = p[3]['temp_var']
4     next_label = new_label()
5 
6     else_start_label = p[8].get('start_label', None)
7     cond_false_next_label = else_start_label if else_start_label else
next_label
8 
9     code = p[3]['code'] +
f"if False {cond_temp_var} goto {cond_false_next_label}\n" + \
10         p[6]['code'] + p[8]['code'] + next_label + ':\n'
11 
12     p[0] = {
13         'code': code
14     }
```

A linha 3 mostra como a regra semântica associada a produção acessa a variável temporária onde a derivação do não-terminal `EXPRESSION` atribuiu o resultado da expressão. A linha 4 gera uma nova *label* para ser utilizada como a label do final do comando, a linha 6 e 7 trata da existência ou não de um *else*. As linhas 9 e 10 fazem a concatenação do código gerado pelo símbolos no corpo da produção, e na linha 12 o atribudo `code` é associado à cabeça da produção.

Os códigos abaixo são exemplos de entrada e saída, respectivamente.

Entrada:

```
if (i % 2 == 0){
  y[j] = i + 1;
  j = j + 1;
} else {
  print 0;
}
```

Saída:

```
t39 = 0
t36 = i
t38 = t36 % t37
t40 = t38 == t39
if False t40 goto LABEL4
t43 = 1
t44 = i + t43
t41 = j
y[t41] = t44
t46 = 1
t47 = j + t46
j = t47
goto LABEL5
LABEL4:
t48 = 0
t49 = t48
print t49
LABEL5:
```

Da mesma forma foram implementados o comando `for`, observado abaixo, mantendo um trecho de código destinado a avaliação da condicional e um, o código contido no escopo do comando, a incrementação da variável de controle, o desvio para a reavaliação da condicional e ao final a *label* para onde a avaliação da condicional desvia o código na saída do *loop*

```
def p_forstat(p: yacc.YaccProduction):
    """FORSTAT : new_for_loop_label FOR LPAREN ATRIBSTAT SEMICOLON
    EXPRESSION SEMICOLON ATRIBSTAT RPAREN LBRACKETS STATELIST RBRACKETS"""
    start_label = new_label()
    next_label = for_loop_control.get_for_loop_next_label()
    cond_code_body = p[6]['code']
    cond_temp_var = p[6]['temp_var']
```

```
first_atrib_code = p[4]['code'] + '\n'
cond_code = f'if False {cond_temp_var} goto {next_label}\n'
body_code = p[11]['code']
increment_code = p[8]['code']
go_to_start_code = f'goto {start_label}\n'

code = first_atrib_code +\
    start_label + ':\n' +\
    cond_code_body +\
    cond_code +\
    body_code +\
    increment_code +\
    go_to_start_code +\
    next_label + ':\n'

p[0] = {
    'code': code
}
```

8 Prova que as SDDs são L-Atribuídas

Para que uma SDD seja classificada como L-atribuída, é necessário que ela não possua ciclos de dependências entre suas produções. Além disso, esse tipo de SDD pode possuir tanto atributos herdados quando atributos sintetizados.

Para garantir que as SDD produzidas nesse trabalho são L-atribuídas, partimos de um princípio básico para evitar que qualquer tipo de ciclo seja gerado: Apenas atributos sintetizados ou atributos herdados da produção a esquerda da qual está sendo derivada podem ser utilizados, ou seja, os atributos herdados sempre vem de uma única direção. O uso de atributos sintetizados evita que existam ciclos entre pais e filhos diretamente, dessa forma evitando que nossa SDD como um todo possua ciclos. A criação das SDDs com essas regras facilitou o uso do `framework` para implementação das SDTs em código, uma vez que o mesmo não da da suporte para acesso a atributos de símbols a direita, como descrito na seção 9.

9 Implementação

9.1 Descrição das ferramentas utilizadas

Para a implementação da regras semânticas e efeitos colaterais das SDDs utilizadas para geração de código intermediário, controle de escopos, tipos das variáveis e geração da árvore de expressão, utilizamos o módulo *yacc* biblioteca PLY (PLY...,).

Neste módulo, podemos associar as regras e efeitos ao código em Python, como no exemplo abaixo.

```
1  def p_numexp(p: yacc.YaccProduction):
2      """NUMEXPRESSION : TERM REC_PLUS_MINUS_TERM"""
3      if p[2] is None:
4          p[0] = p[1]
5      else:
6          result_type = check_type(p[1]['node'],
7                                   p[2]['node'],
8                                   p[2]['operation'],
9                                   p.lineno(1))
10         p[0] = {
11             'node': Node(p[1]['node'],
12                          p[2]['node'],
13                          p[2]['operation'],
14                          result_type)
15         }
```

Na linha 1 do código exemplo, é nomeada uma função. A biblioteca utiliza funções declaradas com o nome no padrão `p_*` para reconhecer quais funções contém em sua *docstring* uma produção a ser mapeada e o código associado a esta.

Dentro da função da produção, estão o que nas SDDs produzidas tratamos como efeitos colaterais e o processamento de atributos sintetizados. O acesso com índice na variável `p` recebida pela função é utilizado para acessar os valores atribuídos a variável quando esta fora a cabeça da produção na função associada. Índices negativos podem ser usados para acessar as propriedades dos símbolos que foram processados anteriormente e tem seus atributos disponíveis.

Ao processar um código fonte, o `parser` criado pela biblioteca utiliza um analisador LALR para definir as produções aplicadas, e executa as derivações de todos os símbolos do corpo da produção na `docstring` da função, para então executar a função. Essa execução é feita da

esquerda para a direita, portanto em cada função só existe formas de acessar as propriedades dos símbolos do corpo da função e dos símbolos à esquerda. Isso implica que toda SDD escrita com `framework` necessite ser L-atribuída.

Para representar as SDDs produzidas, foi utilizada a estrutura de dicionário (ou `hashmap`) da linguagem para criar o efeito semelhante a criação de atributos acessados por um ponto. Por exemplo, a atribuição `NUMEXPRESSION.node = Node(...)` na sintaxe da SDD, pode ser representada pelo código das 10 até 15.

9.2 Exemplo de saída

A execução do programa gera três arquivos: a árvore de expressões aritméticas (arquivo `expressions.json`), a tabela de símbolos com os tipos e escopos (arquivo`symbol_tables.json`) e o código intermediário gerado (arquivo `intermediary_code.gic`). Como exemplo, utilizaremos o programa `examples/exemplo2.ccc` para apresentar as saídas geradas.

9.2.1 Árvore de Expressão para EXPA

O programa apresenta três expressões aritméticas, em dois escopos diferentes. A árvore gerada é apresentada a seguir, com as árvores para todas elas.

A árvore gerada é composta por nodos com esquerda e direita, cada qual com seus respectivos filhos. São guardados os valores, que podem ser identificadores, constantes ou as operações. Por exemplo, a primeira árvore é composta pelo operador "+", com o nodo à esquerda "A"e o nodo à direita "B", gerada a partir da expressão $A + B$.


```
[
  {
    "ID": "<NodeId: 3cf6b4ba-355f-4d58-b305-f537b6bf1a9b>",
    "lineno": 6,
    "tree": {
      "value": "+",
      "left": {
        "value": "A",
        "left": null,
        "right": null
      },
      "right": {
        "value": "B",
        "left": null,
        "right": null
      }
```

```
    }
  },
  {
    "ID": "<NodeId: 792a7e7b-587d-43a4-9df2-12965cc18d35>",
    "lineno": 7,
    "tree": {
      "value": "*",
      "left": {
        "value": "B",
        "left": null,
        "right": null
      },
      "right": {
        "value": "C",
        "left": null,
        "right": null
      }
    }
  },
  {
    "ID": "<NodeId: 41fc03a0-53b2-4b88-922e-7589b73ddf81>",
    "lineno": 21,
    "tree": {
      "value": "+",
      "left": {
        "value": "C",
        "left": null,
        "right": null
      },
      "right": {
        "value": 5,
        "left": null,
        "right": null
      }
    }
  }
]
```

9.2.2 Tabela de Símbolos: tipos e escopos

A tabela de símbolos, com os tipos das variáveis e os escopos, é apresentada a seguir. Nela, verificamos primeiramente a definição de duas *labels* de identificação, para as funções existentes: `func1` e `principal`. A seguir, são apresentados os escopos internos, que também podem conter seus próprios escopos (mais internos ainda). Para cada variável, é apresentando seu identificador, o seu tipo, suas dimensões (arrays) e sua respectiva linha.


```
{
  "table": [
    {
      "identifier_label": "func1",
      "datatype": "function",
      "dimesions": [],
      "line": 1
    },
    {
      "identifier_label": "principal",
      "datatype": "function",
      "dimesions": [],
      "line": 13
    }
  ],
  "inner_scopes": [
    {
      "table": [
        {
          "identifier_label": "B",
          "datatype": "int",
          "dimesions": [],
          "line": 1
        },
        {
          "identifier_label": "A",
          "datatype": "int",
          "dimesions": [],
          "line": 1
        },
        {
          "identifier_label": "C",
```

```
        "datatype": "int",
        "dimesions": [],
        "line": 3
      },
      {
        "identifier_label": "SM",
        "datatype": "int",
        "dimesions": [
          2
        ],
        "line": 5
      },
      {
        "identifier_label": "i",
        "datatype": "int",
        "dimesions": [],
        "line": 9
      }
    ],
    "inner_scopes": []
  },
  {
    "table": [
      {
        "identifier_label": "C",
        "datatype": "int",
        "dimesions": [],
        "line": 15
      },
      {
        "identifier_label": "D",
        "datatype": "int",
        "dimesions": [],
        "line": 16
      },
      {
        "identifier_label": "R",
        "datatype": "int",
        "dimesions": [],
```

```
          "line": 17
        },
        {
          "identifier_label": "total",
          "datatype": "float",
          "dimesions": [],
          "line": 18
        }
      ],
      "inner_scopes": []
    }
  ]
}
```


9.2.3 Código Intermediário

O código intermediário gerado para o programa é apresentado a seguir. Verificamos a utilização de variáveis temporárias, as labels e as chamadas de função.

```
goto LABEL0
func1:
from_params A
from_params B
int C
t1 = 666
C = t1
int SM[2]
t4 = B
t5 = A + t4
t2 = 0
SM[t2] = t5
t8 = C
t9 = B * t8
t6 = 1
SM[t6] = t9
int i
return

LABEL0:
goto LABEL1
```

```
principal:
int C
int D
int R
float total
t11 = 4
C = t11
t13 = 5
D = t13
t15 = 5
t16 = C + t15
total = t16
param C
param D
t18 = call func1, 1
R = t18
return

LABEL1:
```

9.2.4 Mensagens de saída

Todos os exemplos, fora os que possuem erro proposital, são funcionais. Algumas alterações e correções foram feitas. A seguir, serão apresentadas as mensagens de sucesso e de erros que o programa emite.

9.2.4.1 Mensagem de sucesso

Como exemplo, a mensagem de sucesso para o `examples/exemplo1.ccc` é apresentada a seguir. São impressas na tela todas as verificações realizadas, assim como o caminho de saída para os arquivos de análise gerados: tabela de símbolos, árvore de expressão e código intermediário.

```
 Executing...
2021-05-13 23:12:24.421 | INFO     | __main__:main:44 - Total tokens: 166
2021-05-13 23:12:24.421 | INFO     | __main__:main:46 - Running parser for list of tokens...
2021-05-13 23:12:24.421 | INFO     | __main__:main:59 - Syntatic analysis completed with success!
2021-05-13 23:12:24.422 | INFO     | __main__:main:61 - Running semantic analyser...
2021-05-13 23:12:24.425 | INFO     | __main__:main:69 - All break statments are inside loops
2021-05-13 23:12:24.425 | INFO     | __main__:main:109 - Semantic analyser run successfuly
2021-05-13 23:12:24.425 | INFO     | __main__:main:111 - Exporting symbol tables to symbol_tables.json
2021-05-13 23:12:24.426 | INFO     | __main__:main:117 - Exporting symbol tables to expressions.json
2021-05-13 23:12:24.426 | INFO     | __main__:main:122 - Running intermediary code generation...
2021-05-13 23:12:24.430 | INFO     | __main__:main:125 - Exportind intermediary code to
...intermediary_code.gic
```

9.2.4.2 Mensagem de erro na verificação de tipos

O programa `examples/example_error_var_type.ccc` foi desenvolvido a fim de demonstrar a captura de erros em relação às operações ilegais entre variáveis. Primeiramente, é demonstrado que as operações aritméticas entre `int` e `float` são válidas (variáveis `a, b, c`), além da soma de `strings` (variáveis `d, e`), uma vez que o programa não avisa erro. Após, é realizada uma operação de soma entre a `string` e e o `int` a. Como apresentado na seção 5, essa ultima operação não é permitida. A mensagem de erro na saída é apresentada a seguir, em que notamos que o erro semântico é apontado, assim como a linha e a operação que o causou.

```
 Executing...
2021-05-13 22:54:01.714 | INFO     | __main__:main:44 - Total tokens: 49
2021-05-13 22:54:01.714 | INFO     | __main__:main:46 - Running parser for list of tokens...
2021-05-13 22:54:01.714 | INFO     | __main__:main:59 - Syntatic analysis completed with success!
2021-05-13 22:54:01.714 | INFO     | __main__:main:61 - Running semantic analyser...
2021-05-13 22:54:01.716 | INFO     | __main__:main:92 - Invalid operation between string and int...
2021-05-13 22:54:01.716 | INFO     | __main__:main:94 -     e = e + a;

2021-05-13 22:54:01.716 | ERROR    | __main__:main:95 - Semantic error detected!
```

9.2.4.3 Mensagem de erro na declaração de variáveis por escopo

O programa `examples/example_error_var_scope.ccc` foi desenvolvido a fim de demonstrar a captura de erros em relação às declarações repetidas dentro de um escopo. O erro que ocorre é devido à definição da variável `error` como `int` e como `string` no mesmo escopo. A mensagem de erro na saída é apresentada a seguir, em que notamos que o erro semântico é apontado, assim como a linha e a operação que o causou.

```
Executing...
2021-05-13 22:56:31.224 | INFO     | __main__:main:44 - Total tokens: 42
2021-05-13 22:56:31.224 | INFO     | __main__:main:46 - Running parser for list of tokens...
2021-05-13 22:56:31.224 | INFO     | __main__:main:59 - Syntatic analysis completed with success!
2021-05-13 22:56:31.224 | INFO     | __main__:main:61 - Running semantic analyser...
2021-05-13 22:56:31.224 | INFO     | __main__:main:82 - Variable already declared!
...First declaration at line 3
2021-05-13 22:56:31.225 | INFO     | __main__:main:84 -     int error;

2021-05-13 22:56:31.225 | ERROR    | __main__:main:85 - Semantic error detected!
```

9.2.4.4 Mensagem de erro em comandos dentro do escopo

O programa `examples/example_error_break.ccc` foi desenvolvido a fim de demonstrar a captura de erros em relação à verificação de comandos `break` fora de comandos de repetição. Primeiramente, inserimos um `break` em um escopo válido, dentro de um `for`. Posteriormente, o erro apontado é devido ao `break` fora deste escopo. A mensagem de erro na saída é

apresentada a seguir, em que notamos que o erro semântico é apontado, assim como a linha e a operação que o causou.

```
 Executing...
2021-05-13 22:58:06.537 | INFO     | __main__:main:44 - Total tokens: 52
2021-05-13 22:58:06.537 | INFO     | __main__:main:46 - Running parser for list of tokens...
2021-05-13 22:58:06.537 | INFO     | __main__:main:59 - Syntatic analysis completed with success!
2021-05-13 22:58:06.537 | INFO     | __main__:main:61 - Running semantic analyser...
2021-05-13 22:58:06.538 | INFO     | __main__:main:73 - Invalid break usage at line 13:
        break;
2021-05-13 22:58:06.538 | ERROR    | __main__:main:75 - Semantic error detected!
```

9.3 Descrição dos programas escritos

Sete programas seguindo os padrões solicitados estão localizados na pasta *examples*, junto com os exemplos disponibilizados pelo professor. Desses sete, quatro são os mesmos programas disponibilizados no trabalho anterior, pois todos já utilizavam chamadas de função, que é o requisito adicional necessário para essa entrega. Além disso, foram adicionados três programas com erros propositais, a fim de demonstrar as mensagens de erros: `examples/example_error_break.ccc`, `examples/example_error_var_type.ccc` e `examples/example_error_var_scope.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 3.

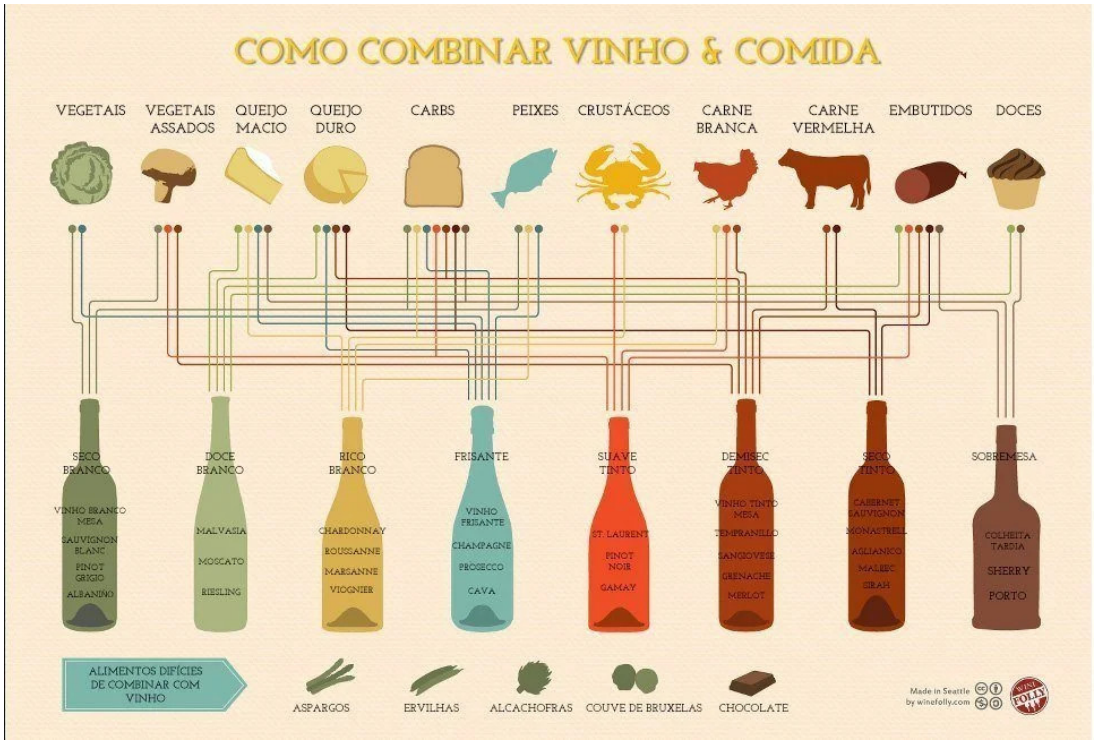

Figura 3 – Guia de harmonização de vinhos e comidas. Fonte: Almanaque SOS.

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>.

PLY (Python Lex-Yacc). Disponível em: <https://ply.readthedocs.io/en/latest/>.

APÊNDICE A – SDD EXPA

```
scope_stack = ScopeStack()
num_expressions: List[Tuple[Node, int]] = []
get_var_type(ident, lineno)
num_expressions_as_json()
new_scope(is_loop: bool)
check_type(left, right, operation, lineno)
```

```
production:
    new_loop_scope : &
rules:
    new_scope(is_loop=True)
```

```
productions:
    PROGRAM : new_scope STATEMENT
    PROGRAM : new_scope FUNCLIST
    PROGRAM : &
rules:
    global_scope = scope_stack.pop()

    PROGRAM.scopes = global_scope.as_json()
    PROGRAM.num_expressions = num_expressions_as_json()

production:
    FUNCDEF : DEF IDENT new_scope LPAREN PARAMLIST RPAREN
        LBRACKETS STATELIST RBRACKETS
rules:
    scope_stack.pop()

    scope = scope_stack.seek()
    entry = TableEntry(IDENT, 'function', [], lineno(IDENT))
    scope.add_entry(entry)
```

```
productions:
    PARAMLIST : DATATYPE IDENT PARAMLISTAUX
rules:
    scope = scope_stack.seek()
    entry = TableEntry(IDENT, DATATYPE, [], lineno(IDENT))
    scope.add_entry(entry)

production:
    DATATYPE : INT_KEYWORD
rules:
    DATATYPE.sin = 'int'

production:
    DATATYPE : FLOAT_KEYWORD
rules:
    DATATYPE.sin = 'float'

production:
    DATATYPE : STRING_KEYWORD
rules:
    DATATYPE.sin = 'string'


production:
    STATEMENT : new_scope LBRACKETS STATELIST RBRACKETS
rules:
    scope_stack.pop()

production:
    STATEMENT : BREAK SEMICOLON
rules:
    current_scope = scope_stack.seek()

    while True:
        if current_scope.is_loop:
            break

        current_scope = current_scope.upper_scope
```

```
        if current_scope is None:
            raise BreakWithoutLoopError(lineno(SEMICOLON))


production:
    VARDECL : DATATYPE IDENT OPT_VECTOR
rules:
    entry = TableEntry(IDENT, DATATYPE, OPT_VECTOR, lineno(IDENT))
    scope = scope_stack.seek()
    scope.add_entry(entry)


production:
    OPT_VECTOR : LSQBRACKETS INT_CONSTANT RSQBRACKETS OPT_VECTOR1
rules:
    OPT_VECTOR.sin = [INT_CONSTANT.sin, *OPT_VECTOR1.sin]


production:
    OPT_VECTOR : &
rules:
    OPT_VECTOR.sin = []

production
    FUNCCALL_OR_EXPRESSION : PLUS FACTOR REC_UNARYEXPR
        REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR
rules:
    right_node = FACTOR.node

    if REC_UNARYEXPR.node:
        result_type = check_type(REC_UNARYEXPR.node,
                                 right_node,
                                 REC_UNARYEXPR.operation,
                                 lineno(PLUS))

        right_node = Node(REC_UNARYEXPR.node,
                          right_node,
                          REC_UNARYEXPR.operation,
                          result_type)
```

```
    if REC_PLUS_MINUS_TERM.node:
        result_type = check_type(REC_PLUS_MINUS_TERM.node,
                                 right_node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(PLUS))
        right_node = Node(REC_PLUS_MINUS_TERM.node,
                          right_node,
                          REC_PLUS_MINUS_TERM.operation,
                          result_type)

    num_expressions.append(right_node)

productions:
    FUNCCALL_OR_EXPRESSION : MINUS FACTOR REC_UNARYEXPR
        REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR
rules:
    right_node = FACTOR.node
    right_node.value *= -1

    if REC_UNARYEXPR.node:
        result_type = check_type(REC_UNARYEXPR.node,
                                 right_node,
                                 REC_UNARYEXPR.operation,
                                 lineno(PLUS))

        right_node = Node(REC_UNARYEXPR.node,
                          right_node,
                          REC_UNARYEXPR.operation,
                          result_type)

    if REC_PLUS_MINUS_TERM.node:
        result_type = check_type(REC_PLUS_MINUS_TERM.node,
                                 right_node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(PLUS))
        right_node = Node(REC_PLUS_MINUS_TERM.node,
                          right_node,
                          REC_PLUS_MINUS_TERM.operation,
```

```
                            result_type)

    num_expressions.append(right_node)


production:
    FUNCCALL_OR_EXPRESSION : INT_CONSTANT REC_UNARYEXPR
        REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR
rules:
    node = Node(None, None, INT_CONSTANT.sin, 'int')

    if REC_UNARYEXPR.node:
        result_type = check_type(node,
                                 REC_UNARYEXPR.node,
                                 REC_UNARYEXPR.operation,
                                 lineno(REC_UNARYEXPR))
        node = Node(
            node,
            REC_UNARYEXPR.node,
            REC_UNARYEXPR.operation,
            result_type
        )

    if REC_PLUS_MINUS_TERM.node:
        result_type = check_type(node,
                                 REC_PLUS_MINUS_TERM.node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(REC_PLUS_MINUS_TERM))
        node = Node(
            node,
            REC_PLUS_MINUS_TERM.node,
            REC_PLUS_MINUS_TERM.operation,
            result_type
        )

    FUNCCALL_OR_EXPRESSION.node = node

    num_expressions.append((node, lineno(REC_PLUS_MINUS_TERM)))
```

```
production:
    FUNCCALL_OR_EXPRESSION : FLOAT_CONSTANT REC_UNARYEXPR
        REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR
rules:
    node = Node(None, None, FLOAT_CONSTANT.sin, 'float')

    if REC_UNARYEXPR.node:
        result_type = check_type(node,
                                 REC_UNARYEXPR.node,
                                 REC_UNARYEXPR.operation,
                                 lineno(REC_UNARYEXPR))
        node = Node(
            node,
            REC_UNARYEXPR.node,
            REC_UNARYEXPR.operation,
            result_type
        )

    if REC_PLUS_MINUS_TERM.node:
        result_type = check_type(node,
                                 REC_PLUS_MINUS_TERM.node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(REC_PLUS_MINUS_TERM))
        node = Node(
            node,
            REC_PLUS_MINUS_TERM.node,
            REC_PLUS_MINUS_TERM.operation,
            result_type
        )

    FUNCCALL_OR_EXPRESSION.node = node

    num_expressions.append((node, lineno(REC_PLUS_MINUS_TERM)))


production:
    FUNCCALL_OR_EXPRESSION : STRING_CONSTANT REC_UNARYEXPR
        REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR
```

```
rules:
    node = Node(None, None, STRING_CONSTANT.sin, 'string')

    if REC_UNARYEXPR.node:
        result_type = check_type(node,
                                 REC_UNARYEXPR.node,
                                 REC_UNARYEXPR.operation,
                                 lineno(REC_UNARYEXPR))
        node = Node(
            node,
            REC_UNARYEXPR.node,
            REC_UNARYEXPR.operation,
            result_type
        )

    if REC_PLUS_MINUS_TERM.node:
        result_type = check_type(node,
                                 REC_PLUS_MINUS_TERM.node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(REC_PLUS_MINUS_TERM))
        node = Node(
            node,
            REC_PLUS_MINUS_TERM.node,
            REC_PLUS_MINUS_TERM.operation,
            result_type
        )

    FUNCCALL_OR_EXPRESSION.node = node

    num_expressions.append((node, lineno(REC_PLUS_MINUS_TERM)))


production:
    FUNCCALL_OR_EXPRESSION : LPAREN NUMEXPRESSION RPAREN REC_UNARYEXPR
        REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR
rules:
    node = NUMEXPRESSION.node

    if REC_UNARYEXPR:
```

```
        result_type = check_type(node,
                                 REC_UNARYEXPR.node,
                                 REC_UNARYEXPR.operation,
                                 lineno(LPAREN))
        node = Node(
            node,
            REC_UNARYEXPR.node,
            REC_UNARYEXPR.operation,
            result_type
        )

    if REC_PLUS_MINUS_TERM:
        result_type = check_type(node,
                                 REC_PLUS_MINUS_TERM.node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(LPAREN))
        node = Node(
            node,
            REC_PLUS_MINUS_TERM.node,
            REC_PLUS_MINUS_TERM.operation,
            result_type
        )

    FUNCCALL_OR_EXPRESSION.node = node

    num_expressions.append((node, lineno(LPAREN)))


production:
    FUNCCALL_OR_EXPRESSION : IDENT FOLLOW_IDENT
rules:
    node = Node(None, None, IDENT, get_var_type(IDENT, lineno(IDENT)))

    if FOLLOW_IDENT:
        node.value += FOLLOW_IDENT.vec_access
        result_type = check_type(node,
                                 FOLLOW_IDENT.node,
                                 FOLLOW_IDENT.operation,
                                 lineno(IDENT))
```

```
        node = Node(
            node,
            FOLLOW_IDENT.node,
            FOLLOW_IDENT.operation,
            result_type
        )

        num_expressions.append((node, lineno(IDENT)))


production:
    FOLLOW_IDENT : OPT_ALLOC_NUMEXP REC_UNARYEXPR REC_PLUS_MINUS_TERM
        OPT_REL_OP_NUM_EXPR
rules:
    node = None
    operation = ''

    if REC_UNARYEXPR:
        node = REC_UNARYEXPR.node
        operation = REC_UNARYEXPR.operation

    if REC_PLUS_MINUS_TERM:
        if node is None:
            node = REC_PLUS_MINUS_TERM.node
            operation = REC_PLUS_MINUS_TERM.operation

        else:
            result_type = check_type(node,
                                     REC_PLUS_MINUS_TERM.node,
                                     REC_PLUS_MINUS_TERM.operation,
                                     lineno(FOLLOW_IDENT))
            node = Node(
                node,
                REC_PLUS_MINUS_TERM.node,
                REC_PLUS_MINUS_TERM.operation,
                result_type
            )

    FOLLOW_IDENT.vec_access = OPT_ALLOC_NUMEXP.sin
```

```
    FOLLOW_IDENT.node = node
    FOLLOW_IDENT.operation = operation


production:
    IFSTAT : IF LPAREN EXPRESSION RPAREN new_scope LBRACKETS STATELIST
        RBRACKETS OPT_ELSE
rules:
    scope_stack.pop()


production:
    OPT_ELSE : ELSE new_scope LBRACKETS STATELIST RBRACKETS
rules:
    scope_stack.pop()


production:
    FORSTAT : FOR LPAREN ATRIBSTAT SEMICOLON EXPRESSION SEMICOLON
        ATRIBSTAT RPAREN new_loop_scope LBRACKETS STATELIST RBRACKETS
rules:
    scope_stack.pop()


producttion:
    ALLOCEXPRESSION : NEW DATATYPE LSQBRACKETS NUMEXPRESSION RSQBRACKETS
        OPT_ALLOC_NUMEXP
rules:
    num_expressions.append((NUMEXPRESSION.node, lineno(NEW)))


production:
    OPT_ALLOC_NUMEXP : &
rules:
    OPT_ALLOC_NUMEXP.sin = ''


production:
    OPT_ALLOC_NUMEXP : LSQBRACKETS NUMEXPRESSION RSQBRACKETS
```

```
        OPT_ALLOC_NUMEXP1
rules:
    OPT_ALLOC_NUMEXP.sin = '[' + NUMEXPRESSION.node.id + ']' +
        OPT_ALLOC_NUMEXP1.sin

    num_expressions.append((NUMEXPRESSION.node, lineno(NEW)))


production:
    EXPRESSION : NUMEXPRESSION OPT_REL_OP_NUM_EXPR
rules:
    num_expressions.append((NUMEXPRESSION.node, lineno(NUMEXPRESSION)))


production:
    OPT_REL_OP_NUM_EXPR : REL_OP NUMEXPRESSION
rules:
    num_expressions.append((NUMEXPRESSION.node, lineno(REL_OP)))

production:
    NUMEXPRESSION : TERM REC_PLUS_MINUS_TERM
rules:
    if REC_PLUS_MINUS_TERM.node:
        NUMEXPRESSION.node = TERM.node

    else:
        result_type = check_type(TERM.node,
                                 REC_PLUS_MINUS_TERM.node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(TERM))
        NUMEXPRESSION.node = Node(TERM.node,
                                  REC_PLUS_MINUS_TERM.node,
                                  REC_PLUS_MINUS_TERM.operation,
                                  result_type)


production:
    REC_PLUS_MINUS_TERM : PLUS_OR_MINUS TERM REC_PLUS_MINUS_TERM1
rules:
```

```
    if REC_PLUS_MINUS_TERM1.node:
        result_type = check_type(TERM.node,
                                 REC_PLUS_MINUS_TERM1.node,
                                 REC_PLUS_MINUS_TERM1.operation,
                                 lineno(PLUS_OR_MINUS))

        REC_PLUS_MINUS_TERM.node = Node(TERM.node,
                                        REC_PLUS_MINUS_TERM1.node,
                                        REC_PLUS_MINUS_TERM1.operation,
                                        result_type),
        REC_PLUS_MINUS_TERM.operation = PLUS_OR_MINUS.operation

    else:
        REC_PLUS_MINUS_TERM.node = TERM.node,
        REC_PLUS_MINUS_TERM.operation = PLUS_OR_MINUS.operation


production:
    PLUS_OR_MINUS : PLUS
rules:
    PLUS_OR_MINUS.operation = '+'

production:
    PLUS_OR_MINUS : MINUS
rules:
    PLUS_OR_MINUS.operation = '-'


production:
    TERM : UNARYEXPR REC_UNARYEXPR
rules:
    if REC_UNARYEXPR.node:
        result_type = check_type(UNARYEXPR.node,
                                 REC_UNARYEXPR.node,
                                 REC_UNARYEXPR.operation,
                                 lineno(UNARYEXPR))

        TERM.node = Node(
            UNARYEXPR.node,
```

```
            REC_UNARYEXPR.node,
            REC_UNARYEXPR.operation,
            result_type
        ),
        TERM.operation = REC_UNARYEXPR.operation

    else:
        TERM.node = UNARYEXPR.node


production:
    REC_UNARYEXPR : UNARYEXPR_OP TERM
rules:
    REC_UNARYEXPR.node = TERM.node,
    REC_UNARYEXPR.operation = UNARYEXPR_OP.operation


production:
    UNARYEXPR_OP : TIMES
rules:
    UNARYEXPR_OP.operation = '*'


production:
    UNARYEXPR_OP : MODULE
rules:
    UNARYEXPR_OP.operation = '%'


production:
    UNARYEXPR_OP : DIVIDE
rules:
    UNARYEXPR_OP.operation = '/'


production:
    UNARYEXPR : PLUS_OR_MINUS FACTOR
rules:
    if PLUS_OR_MINUS.operation == '-':
```

```
        FACTORnode.value *= -1

    UNARYEXPR.node = FACTOR.node


production:
    UNARYEXPR : FACTOR
rules:
    UNARYEXPR.node = FACTOR.node


production:
    FACTOR : INT_CONSTANT
rules:
    FACTOR.node = Node(None, None, INT_CONSTANT.val, 'int')


production:
    FACTOR : FLOAT_CONSTANT
rules:
    FACTOR.node = Node(None, None, FLOAT_CONSTANT.val, 'float')


production:
    FACTOR : STRING_CONSTANT
rules:
    FACTOR.node = Node(None, None, STRING_CONSTANT.val, 'string')


production:
    FACTOR : NULL
rules:
    FACTOR.node = Node(None, None, NULL.val, 'null')


production:
    FACTOR : LVALUE
rules:
    FACTOR.node = LVALUE.node
```

```
production:
    FACTOR : LPAREN NUMEXPRESSION RPAREN
rules:
    FACTOR.node = NUMEXPRESSION.node

    num_expressions.append((NUMEXPRESSION.node, lineno(LPAREN)))


production:
    LVALUE : IDENT OPT_ALLOC_NUMEXP
rules:
    LVALUE.node = Node(None, None, IDENT.val + OPT_ALLOC_NUMEXP.sin,
                       result_type=get_var_type(IDENT.val, lineno(IDENT)))
```

APÊNDICE B – SDT EXPA

```
scope_stack = ScopeStack()
num_expressions: List[Tuple[Node, int]] = []
get_var_type(ident, lineno)
num_expressions_as_json()
new_scope(is_loop: bool)
check_type(left, right, operation, lineno)


new_loop_scope : & { new_scope(is_loop=True) }
new_scope : & { new_scope(is_loop=False) }

PROGRAM : new_scope STATEMENT { PROGRAM.scopes = global_scope.as_json();
    PROGRAM.num_expressions = num_expressions_as_json() }

PROGRAM : new_scope FUNCLIST { PROGRAM.scopes = global_scope.as_json();
    PROGRAM.num_expressions = num_expressions_as_json() }

PROGRAM : & { global_scope = scope_stack.pop() {
    PROGRAM.scopes = global_scope.as_json();
    PROGRAM.num_expressions = num_expressions_as_json() }

FUNCDEF : DEF IDENT new_scope LPAREN PARAMLIST RPAREN LBRACKETS
    STATELIST RBRACKETS { scope_stack.pop();
    scope = scope_stack.seek(); entry = TableEntry(
        IDENT,
        'function',
        [],
        lineno(IDENT)
    );
    scope.add_entry(entry);
}

PARAMLIST : DATATYPE IDENT PARAMLISTAUX { scope = scope_stack.seek();
    entry = TableEntry(IDENT, DATATYPE, [], lineno(IDENT));
    scope.add_entry(entry)
```

```
}

DATATYPE : INT_KEYWORD { DATATYPE.sin = 'int' }

DATATYPE : FLOAT_KEYWORD { DATATYPE.sin = 'float' }

DATATYPE : STRING_KEYWORD { DATATYPE.sin = 'string' }

STATEMENT : new_scope LBRACKETS STATELIST RBRACKETS { scope_stack.pop() }

STATEMENT : BREAK SEMICOLON {
    current_scope = scope_stack.seek();

    while True:
        if current_scope.is_loop:
            break

        current_scope = current_scope.upper_scope

        if current_scope is None:
            raise BreakWithoutLoopError(lineno(SEMICOLON)) }

VARDECL : DATATYPE IDENT OPT_VECTOR { entry = TableEntry(
        IDENT,
        DATATYPE,
        OPT_VECTOR,
        lineno(IDENT)
    )
    scope = scope_stack.seek()
    scope.add_entry(entry)

OPT_VECTOR : LSQBRACKETS INT_CONSTANT RSQBRACKETS OPT_VECTOR1 {
    OPT_VECTOR.sin = [INT_CONSTANT.sin, *OPT_VECTOR1.sin] }

OPT_VECTOR : & { OPT_VECTOR.sin = [] }

FUNCCALL_OR_EXPRESSION : PLUS FACTOR REC_UNARYEXPR
    REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR { right_node = FACTOR.node };
```

```
    if REC_UNARYEXPR.node:
        result_type = check_type(REC_UNARYEXPR.node,
                                 right_node,
                                 REC_UNARYEXPR.operation,
                                 lineno(PLUS))

        right_node = Node(REC_UNARYEXPR.node,
                          right_node,
                          REC_UNARYEXPR.operation,
                          result_type)

    if REC_PLUS_MINUS_TERM.node:
        result_type = check_type(REC_PLUS_MINUS_TERM.node,
                                 right_node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(PLUS))
        right_node = Node(REC_PLUS_MINUS_TERM.node,
                          right_node,
                          REC_PLUS_MINUS_TERM.operation,
                          result_type)

    num_expressions.append(right_node) }

FUNCCALL_OR_EXPRESSION : MINUS FACTOR REC_UNARYEXPR
    REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR {
    right_node = FACTOR.node;
    right_node.value *= -1

    if REC_UNARYEXPR.node:
        result_type = check_type(REC_UNARYEXPR.node,
                                 right_node,
                                 REC_UNARYEXPR.operation,
                                 lineno(PLUS))

        right_node = Node(REC_UNARYEXPR.node,
                          right_node,
                          REC_UNARYEXPR.operation,
                          result_type)
```

```
    if REC_PLUS_MINUS_TERM.node:
        result_type = check_type(REC_PLUS_MINUS_TERM.node,
                                 right_node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(PLUS))
        right_node = Node(REC_PLUS_MINUS_TERM.node,
                          right_node,
                          REC_PLUS_MINUS_TERM.operation,
                          result_type)

    num_expressions.append(right_node)
}
```

```
FUNCCALL_OR_EXPRESSION : INT_CONSTANT REC_UNARYEXPR REC_PLUS_MINUS_TERM
    OPT_REL_OP_NUM_EXPR {
    node = Node(None, None, INT_CONSTANT.sin, 'int');

    if REC_UNARYEXPR.node:
        result_type = check_type(node,
                                 REC_UNARYEXPR.node,
                                 REC_UNARYEXPR.operation,
                                 lineno(REC_UNARYEXPR))
        node = Node(
            node,
            REC_UNARYEXPR.node,
            REC_UNARYEXPR.operation,
            result_type
        )

    if REC_PLUS_MINUS_TERM.node:
        result_type = check_type(node,
                                 REC_PLUS_MINUS_TERM.node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(REC_PLUS_MINUS_TERM))
        node = Node(
            node,
            REC_PLUS_MINUS_TERM.node,
```

```
            REC_PLUS_MINUS_TERM.operation,
            result_type
        )

    FUNCCALL_OR_EXPRESSION.node = node

    num_expressions.append((node, lineno(REC_PLUS_MINUS_TERM)))
}



FUNCCALL_OR_EXPRESSION : FLOAT_CONSTANT REC_UNARYEXPR
    REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR {
    node = Node(None, None, FLOAT_CONSTANT.sin, 'float');

    if REC_UNARYEXPR.node:
        result_type = check_type(node,
                                 REC_UNARYEXPR.node,
                                 REC_UNARYEXPR.operation,
                                 lineno(REC_UNARYEXPR))
        node = Node(
            node,
            REC_UNARYEXPR.node,
            REC_UNARYEXPR.operation,
            result_type
        )

    if REC_PLUS_MINUS_TERM.node:
        result_type = check_type(node,
                                 REC_PLUS_MINUS_TERM.node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(REC_PLUS_MINUS_TERM))
        node = Node(
            node,
            REC_PLUS_MINUS_TERM.node,
            REC_PLUS_MINUS_TERM.operation,
            result_type
        )
```

```
    FUNCCALL_OR_EXPRESSION.node = node

    num_expressions.append((node, lineno(REC_PLUS_MINUS_TERM)))
}


FUNCCALL_OR_EXPRESSION : STRING_CONSTANT REC_UNARYEXPR
    REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR {
    node = Node(None, None, STRING_CONSTANT.sin, 'string');

    if REC_UNARYEXPR.node:
        result_type = check_type(node,
                                 REC_UNARYEXPR.node,
                                 REC_UNARYEXPR.operation,
                                 lineno(REC_UNARYEXPR))
        node = Node(
            node,
            REC_UNARYEXPR.node,
            REC_UNARYEXPR.operation,
            result_type
        )

    if REC_PLUS_MINUS_TERM.node:
        result_type = check_type(node,
                                 REC_PLUS_MINUS_TERM.node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(REC_PLUS_MINUS_TERM))
        node = Node(
            node,
            REC_PLUS_MINUS_TERM.node,
            REC_PLUS_MINUS_TERM.operation,
            result_type
        )

    FUNCCALL_OR_EXPRESSION.node = node

    num_expressions.append((node, lineno(REC_PLUS_MINUS_TERM)))
}
```

```
FUNCCALL_OR_EXPRESSION : LPAREN NUMEXPRESSION RPAREN REC_UNARYEXPR
    REC_PLUS_MINUS_TERM OPT_REL_OP_NUM_EXPR {
    node = NUMEXPRESSION.node;

    if REC_UNARYEXPR:
        result_type = check_type(node,
                                 REC_UNARYEXPR.node,
                                 REC_UNARYEXPR.operation,
                                 lineno(LPAREN))
        node = Node(
            node,
            REC_UNARYEXPR.node,
            REC_UNARYEXPR.operation,
            result_type
        )

    if REC_PLUS_MINUS_TERM:
        result_type = check_type(node,
                                 REC_PLUS_MINUS_TERM.node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(LPAREN))
        node = Node(
            node,
            REC_PLUS_MINUS_TERM.node,
            REC_PLUS_MINUS_TERM.operation,
            result_type
        )

    FUNCCALL_OR_EXPRESSION.node = node;
    num_expressions.append((node, lineno(LPAREN)))
}

FUNCCALL_OR_EXPRESSION : IDENT FOLLOW_IDENT {
    node = Node(None, None, IDENT, get_var_type(IDENT, lineno(IDENT)));
    if FOLLOW_IDENT:
        node.value += FOLLOW_IDENT.vec_access
        result_type = check_type(node,
                                 FOLLOW_IDENT.node,
                                 FOLLOW_IDENT.operation,
```

```
                                    lineno(IDENT))
        node = Node(
            node,
            FOLLOW_IDENT.node,
            FOLLOW_IDENT.operation,
            result_type
        )
        num_expressions.append((node, lineno(IDENT)))
}

FOLLOW_IDENT : OPT_ALLOC_NUMEXP REC_UNARYEXPR REC_PLUS_MINUS_TERM
    OPT_REL_OP_NUM_EXPR {
    node = None;
    operation = '';
    if REC_UNARYEXPR:
        node = REC_UNARYEXPR.node
        operation = REC_UNARYEXPR.operation

    if REC_PLUS_MINUS_TERM:
        if node is None:
            node = REC_PLUS_MINUS_TERM.node
            operation = REC_PLUS_MINUS_TERM.operation

        else:
            result_type = check_type(node,
                                     REC_PLUS_MINUS_TERM.node,
                                     REC_PLUS_MINUS_TERM.operation,
                                     lineno(FOLLOW_IDENT))
            node = Node(
                node,
                REC_PLUS_MINUS_TERM.node,
                REC_PLUS_MINUS_TERM.operation,
                result_type
            )

    FOLLOW_IDENT.vec_access = OPT_ALLOC_NUMEXP.sin
    FOLLOW_IDENT.node = node
    FOLLOW_IDENT.operation = operation
}
```

```
IFSTAT : IF LPAREN EXPRESSION RPAREN new_scope
    LBRACKETS STATELIST RBRACKETS OPT_ELSE { scope_stack.pop() }

OPT_ELSE : ELSE new_scope LBRACKETS STATELIST
    RBRACKETS { scope_stack.pop() }

FORSTAT : FOR LPAREN ATRIBSTAT SEMICOLON EXPRESSION
    SEMICOLON ATRIBSTAT RPAREN new_loop_scope LBRACKETS
    STATELIST RBRACKETS { scope_stack.pop() }

ALLOCEXPRESSION : NEW DATATYPE LSQBRACKETS NUMEXPRESSION RSQBRACKETS
    OPT_ALLOC_NUMEXP {
        num_expressions.append((NUMEXPRESSION.node, lineno(NEW)))
    }

OPT_ALLOC_NUMEXP : & { OPT_ALLOC_NUMEXP.sin = ''

OPT_ALLOC_NUMEXP : LSQBRACKETS NUMEXPRESSION RSQBRACKETS
    OPT_ALLOC_NUMEXP1 {
        OPT_ALLOC_NUMEXP.sin = '[' + NUMEXPRESSION.node.id + ']'
        + OPT_ALLOC_NUMEXP1.sin; num_expressions.append(
            (NUMEXPRESSION.node, lineno(NEW))
        )
    }

EXPRESSION : NUMEXPRESSION OPT_REL_OP_NUM_EXPR {
    num_expressions.append((NUMEXPRESSION.node, lineno(NUMEXPRESSION)))
}

OPT_REL_OP_NUM_EXPR : REL_OP NUMEXPRESSION {
    num_expressions.append((NUMEXPRESSION.node, lineno(REL_OP)))
}

NUMEXPRESSION : TERM REC_PLUS_MINUS_TERM {
    if REC_PLUS_MINUS_TERM.node:
        NUMEXPRESSION.node = TERM.node

    else:
```

```
        result_type = check_type(TERM.node,
                                 REC_PLUS_MINUS_TERM.node,
                                 REC_PLUS_MINUS_TERM.operation,
                                 lineno(TERM))
        NUMEXPRESSION.node = Node(TERM.node,
                                  REC_PLUS_MINUS_TERM.node,
                                  REC_PLUS_MINUS_TERM.operation,
                                  result_type)
}

REC_PLUS_MINUS_TERM : PLUS_OR_MINUS TERM REC_PLUS_MINUS_TERM1 {
    if REC_PLUS_MINUS_TERM1.node:
        result_type = check_type(TERM.node,
                                 REC_PLUS_MINUS_TERM1.node,
                                 REC_PLUS_MINUS_TERM1.operation,
                                 lineno(PLUS_OR_MINUS))

        REC_PLUS_MINUS_TERM.node = Node(TERM.node,
                                        REC_PLUS_MINUS_TERM1.node,
                                        REC_PLUS_MINUS_TERM1.operation,
                                        result_type),
        REC_PLUS_MINUS_TERM.operation = PLUS_OR_MINUS.operation

    else:
        REC_PLUS_MINUS_TERM.node = TERM.node,
        REC_PLUS_MINUS_TERM.operation = PLUS_OR_MINUS.operation}



PLUS_OR_MINUS : PLUS { PLUS_OR_MINUS.operation = '+' }


PLUS_OR_MINUS : MINUS { PLUS_OR_MINUS.operation = '-' }

TERM : UNARYEXPR REC_UNARYEXPR {
    if REC_UNARYEXPR.node:
        result_type = check_type(UNARYEXPR.node,
                                 REC_UNARYEXPR.node,
                                 REC_UNARYEXPR.operation,
```

```
                                    lineno(UNARYEXPR))

        TERM.node = Node(UNARYEXPR.node, REC_UNARYEXPR.node,
            REC_UNARYEXPR.operation, result_type),
        TERM.operation = REC_UNARYEXPR.operation

    else:
        TERM.node = UNARYEXPR.node}



REC_UNARYEXPR : UNARYEXPR_OP TERM {
    REC_UNARYEXPR.node = TERM.node;
    REC_UNARYEXPR.operation = UNARYEXPR_OP.operation
}

UNARYEXPR_OP : TIMES { UNARYEXPR_OP.operation = '*' }

UNARYEXPR_OP : MODULE { UNARYEXPR_OP.operation = '%' }

UNARYEXPR_OP : DIVIDE { UNARYEXPR_OP.operation = '/' }

UNARYEXPR : PLUS_OR_MINUS FACTOR {
    if PLUS_OR_MINUS.operation == '-':
        FACTORnode.value *= -1
    UNARYEXPR.node = FACTOR.node}

UNARYEXPR : FACTOR { UNARYEXPR.node = FACTOR.node }

FACTOR : INT_CONSTANT {
    FACTOR.node = Node(None, None, INT_CONSTANT.val, 'int')
}

FACTOR : FLOAT_CONSTANT {
    FACTOR.node = Node(None, None, FLOAT_CONSTANT.val, 'float')
}

FACTOR : STRING_CONSTANT {
    FACTOR.node = Node(None, None, STRING_CONSTANT.val, 'string')
```

```
}

FACTOR : NULL {
    FACTOR.node = Node(None, None, NULL.val, 'null')
}

FACTOR : LVALUE {
    FACTOR.node = LVALUE.node
}

FACTOR : LPAREN NUMEXPRESSION RPAREN {
    FACTOR.node = NUMEXPRESSION.node;
    num_expressions.append((NUMEXPRESSION.node, lineno(LPAREN)))
}

LVALUE : IDENT OPT_ALLOC_NUMEXP {
    LVALUE.node = Node(None,
                       None,
                       IDENT.val + OPT_ALLOC_NUMEXP.sin,
                       result_type=get_var_type(IDENT.val, lineno(IDENT)))
}
```