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

Relatório do Exercício Programa 3: Analisador Semântico

Grupo:
Artur Barichello (16200636)
Lucas Verdade (17104409)
Lucas Zacchi (16104597)

Professor:
Álvaro Junio Pereira Franco

Agosto
2021

Sumário

1 Introdução

Este relatório descreve a implementação de um Analisador Semântico da gramática CC-2021-1, como descrita na seção 2

Para este relatório foram produzidas duas gramáticas adicionais, derivadas de CC-2021-1: São elas a EXPA, descrita na seção 4.1 e DEC, apresentada em 5.1. A partir dessas derivações, forama produzidas SDDs e SDTs para ambas as gramáticas.

Na seção 9 foi descrita a implementação do exercício-programa, bem como a ferramenta utilizada e as entradas e saídas obtidas durante a execução.

Na seção 10.2 são demonstrados os códigos que foram desenvolvidos para validar os pontos que foram solicitados no enunciado do trabalho.

Em relação ao Analisador Semântico, foram feitas alterações referentes ao exercício programa 2. Este processo é descrito na seção 3.

2 Gramática CC-2021-1

$PROGRAM$	→ (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 | nil | LVALUE | (NUMEXPRESSION))
$LVALUE$	→ ident([NUMEXPRESSION])*

3 Mudanças e correções

Conforme especificado no enunciado do Exercício Programa 3, foi elaborada esta seção para apresentar as correções feitas em tópicos referentes à implementação do Analisador Sintático.

3.1 Definição de recursão a esquerda

Recursões à esquerda são definidas como produções do tipo:

$$S \quad \rightarrow \text{Sb}$$

E produções com recursão indireta à esquerda são definidas como produções do tipo:

$$S \quad \rightarrow \text{Ab}$$
$$A \quad \rightarrow \text{Sb}$$

3.2 Interromper execução ao encontrar erro sintático

No exercício programa anterior, a execução não era interrompida ao encontrar um erro sintático. Foram feitas modificações para executar da forma correta como solicitada. Ao encontrar um erro sintático o compilador exibe o erro. O erro capturado no *catch* é um tipo de *Exception* que criamos no arquivo *src/output.py* para tratar cada tipo de erro do compilador. A modificação no código consistiu no trecho abaixo do arquivo *main.py*:

```
except Exception as error:
    print(f"Erro na etapa de análise sintática: {error}")
    sys.exit(-1)
```

3.3 Correção dos códigos de exemplos

Haviam erros sintáticos nos códigos de exemplo program1.lua, program2.lua, program3.lua, program4.lua que não estavam de acordo com a forma corrreta da gramática modificada. Portanto os mesmos tiveram que ser modificados para se adequar a gramática final da linguagem.

3.4 Correção do analisador sintático

O código do analisador sintático desenvolvido anteriormente possuia algumas implementações de análise semântica, e isso estava causando problemas

já que foram feitas algumas implementações mínimas de análise semântica na tentativa de avançar além do solicitado. Esse problema foi solucionado ao remover o corpo das funções do tipo p_TOKEN e deixar apenas a definição das produções. Dessa forma apenas a análise sintática é feita. O código de análise sintática agora se encontra no arquivo *parser.py*. O código de *parser.py* é executado na main no trecho abaixo.

```
try:
    syntax_parser = yacc.yacc(start="PROGRAM", check_recursion=True)
    syntax_result = syntax_parser.parse(src, debug=args.debug, lexer=syntax_
except Exception as error:
    print(f"Erro na etapa de análise sintática: {error}")
    sys.exit(-1)
```

3.5 Gramática modificada para estar em LL(1)

Nessa entrega a gramática foi modificada para que estivesse em LL(1). Essas alterações foram feitas manualmente pelo grupo e a gramática resultante se encontra na descrição abaixo.

PROGRAM	→ STATEMENT
	| FUNCLIST
	| &
FUNCLIST	→ FUNCDEF FUNCLISTTMP
FUNCLISTTMP	→ FUNCLIST
	| &
FUNCDEF	→ def ident (PARAMLIST) STATELIST
PARAMLIST	→ DATATYPE ident PARAMLISTTMP
	| &
PARAMLISTTMP	→ , PARAMLIST
	| &
DATATYPE	→ int
	| float
	| string
STATEMENT	→ VARDECL ;
	| ATRIBSTAT ;
	| PRINTSTAT ;
	| READSTAT ;
	| RETURNSTAT ;
	| IFSTAT
	| FORSTAT
	| STATELIST
	| break ;
	| ;
VARDECL	→ DATATYPE ident OPTIONAL_VECTOR
OPTIONAL_VECTOR	→ [int_constant] OPTIONAL_VECTOR
	| &
ATRIBSTAT	→ LVALUE = ATRIB_RIGHT
ATRIB_RIGHT	→ EXPRESSION_OR_FUNCCALL
	| ALLOCEXPRESSION
EXPRESSION_OR_FUNCCALL	→ + FACTOR RECURSIVE_UNARYEXPR RECURSIVE_MINUS_OR_PLUS OPTIONAL_REL_OP_NUMEXPRESSION
	| - FACTOR RECURSIVE_UNARYEXPR RECURSIVE_MINUS_OR_PLUS OPTIONAL_REL_OP_NUMEXPRESSION
	| int_constant RECURSIVE_UNARYEXPR RECURSIVE_MINUS_OR_PLUS OPTIONAL_REL_OP_NUMEXPRESSION
	| float_constant RECURSIVE_UNARYEXPR RECURSIVE_MINUS_OR_PLUS OPTIONAL_REL_OP_NUMEXPRESSION
	| string_constant RECURSIVE_UNARYEXPR RECURSIVE_MINUS_OR_PLUS OPTIONAL_REL_OP_NUMEXPRESSION
	| null RECURSIVE_UNARYEXPR RECURSIVE_MINUS_OR_PLUS OPTIONAL_REL_OP_NUMEXPRESSION
	| (NUMEXPRESSION) RECURSIVE_UNARYEXPR RECURSIVE_MINUS_OR_PLUS OPTIONAL_REL_OP_NUMEXPRESSION
	| ident FOLLOW_LABEL
FOLLOW_LABEL	→ OPTIONAL_ALLOC_NUMEXPRESSION RECURSIVE_UNARYEXPR RECURSIVE_MINUS_OR_PLUS OPTIONAL_REL_OP_NUMEXPRESSION
	| (PARAMLISTCALL)

PARAMLISTCALL → ident PARAMLISTCALLTMP
| &

PARAMLISTCALLTMP → , PARAMLISTCALL
| &

PRINTSTAT → print EXPRESSION

READSTAT → read LVALUE

RETURNSTAT → return

IFSTAT → if (EXPRESSION) STATELIST OPTIONAL_ELSE

OPTIONAL_ELSE → else STATELIST
| &

FORSTAT → for (ATRIBSTAT ; EXPRESSION ; ATRIBSTAT) STATELIST

STATELIST → STATEMENT OPTIONAL_STATELIST

OPTIONAL_STATELIST → STATELIST
| &

STATEMENT → VARDECL; | ATRIBSTAT; | PRINTSTAT; |

ALLOCEXPRESSION → new DATATYPE [NUMEXPRESSION] OPTIONAL_ALLOC_NUMEXPRESSION

OPTIONAL_ALLOC_NUMEXPRESSION → [NUMEXPRESSION] OPTIONAL_ALLOC_NUMEXPRESSION
| &

EXPRESSION → NUMEXPRESSION OPTIONAL_REL_OP_NUMEXPRESSION

OPTIONAL_REL_OP_NUMEXPRESSION → REL_OP NUMEXPRESSION
| &

REL_OP → < | > | <= | >= | == | !=

NUMEXPRESSION → TERM RECURSIVE_MINUS_OR_PLUS

RECURSIVE_MINUS_OR_PLUS → MINUS_OR_PLUS TERM RECURSIVE_MINUS_OR_PLUS
| &

MINUS_OR_PLUS → +
| -

TERM → UNARYEXPR RECURSIVE_UNARYEXPR

RECURSIVE_UNARYEXPR → UNARYEXPR_OPERATOR TERM
| &

UNARYEXPR_OPERATOR → *
|
| %

UNARYEXPR → MINUS_OR_PLUS FACTOR
| FACTOR

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

LVALUE → ident OPTIONAL_ALLOC_NUMEXPRESSION

3.6 Mostrando que está em LL(1)

Para mostrar que a gramática está em LL(1) foi utilizada uma ferramenta externa online chamada *"LL(1) parser visualization"* disponibilizada pela *Princeton University*[2]. A ferramenta aceita somente gramática em LL(1) como entrada e sua saída são tabelas com Anulável/First/Follow e a tabela de transição.

Ao executar a gramática final na ferramenta as tabelas de anuláveis, First, follow e de transição são geradas com sucesso, o que significa que ela está em LL(1). Ao tentar executar com uma gramática que não esteja em LL(1) a ferramenta exibe um aviso de erro de conflito ou de que não está em LL(1).

Caso seja necessário é possível usar os conjuntos first e follow gerados para testar o teorema. Como as tabelas da gramáticas são relativamente grandes, foram omitidas deste relatório, mas podem ser visualizadas ao executar a gramática final na própria ferramenta[2].

A ferramenta exige um formato específico para entrada de gramáticas e por isso fizemos um arquivo com nossa gramática final adaptada ao formato de entrada. O arquivo de entrada para a ferramenta é o *ll1.txt* localizado dentro da pasta *src* do projeto. Para testar a gramática basta copiar o conteúdo do arquivo *ll1.txt* e colar no campo de texto da ferramenta com o título:

"1. Write your LL(1) grammar (empty string " represents ϵ)".

4 Árvore de Expressão (EXPA)

Para construção da árvore de expressão, as produções que geram expressões aritméticas foram separadas da gramática **CC-2021-1** e agrupadas na gramática **EXPA**, descrita abaixo.

4.1 Separação da Gramática

EXPRESSION	→ NUMEXPRESSION OPTIONAL_REL_OP_NUMEXPRESSION
OPTIONAL_REL_OP_NUMEXPRESSION	→ REL_OP NUMEXPRESSION
	| &
REL_OP	→ < | > | <= | >= | == | !=
NUMEXPRESSION	→ TERM RECURSIVE_MINUS_OR_PLUS
RECURSIVE_MINUS_OR_PLUS	→ MINUS_OR_PLUS
	TERM RECURSIVE_MINUS_OR_PLUS
	| &
MINUS_OR_PLUS	→ +
	| -
TERM	→ UNARYEXPR RECURSIVE_UNARYEXPR
RECURSIVE_UNARYEXPR	→ UNARYEXPR_OPERATOR TERM
	| &
UNARYEXPR_OPERATOR	→ *
	|
	| %
UNARYEXPR	→ MINUS_OR_PLUS FACTOR
	| FACTOR
FACTOR	→ int_constant
	| float_constant
	| string_constant
	| null
	| LVALUE
	| (NUMEXPRESSION)
LVALUE	→ ident OPTIONAL_ALLOC_NUMEXPRESSION

4.2 SDD L-Atribuída

A partir de **EXPA**, foi construída a SDD L-atribuída com o intuito de construir a árvore de expressão. A SDD é apresentada abaixo, e a prova de que ela é L-atribuída está descrita na seção 9.8

Código Fonte 1: SDD da gramática EXPA

```
production:
    EXPRESSION : NUMEXPRESSION OPTIONAL_REL_OP_NUMEXPRESSION
rules:
    expressions.append((numexpr_node, lineno(numexpression)))


production:
    OPTIONAL_REL_OP_NUMEXPRESSION : REL_OP NUMEXPRESSION
rules:
    expressions.append((expression.node, lineno(expression)))

production:
    NUMEXPRESSION : TERM RECURSIVE_MINUS_OR_PLUS
rules:
   if recursive is None:
        expression = term
    else:
        result_type = check_valid_operation(
            term(node), recursive(node),
            recursive(node), lineno(term)
        )
        expression = {
            node: TreeNode(
                term(node), recursive(node),
                recursive(op), result_type
            )
        }

production:
    RECURSIVE_MINUS_OR_PLUS : MINUS_OR_PLUS TERM RECURSIVE_MINUS_OR_PLUS
rules:
    if recursive is None:
        expression = term
    elif expression[4]:
        # more recursions
        result_type = check_valid_operation(
            minus_or_plus_1(node), term(node),
            minus_or_plus_2(node)[operation], lineno(expression)
```

```
        )
        recursive_minus_or_plus = {
            "node": TreeNode(
                term(node), minus_or_plus_2(node),
               minus_or_plus_2(node)[operation], result_type
            ),
            op: term(node)(op),
        }
    else:
        # last recursion
        recursive_minus_or_plus = {node: term(node),
                     op: minus_or_plus_1(op)}


production:
    MINUS_OR_PLUS : OPERATION
rules:
    minus_or_plus = op


production:
    TERM : UNARYEXPR RECURSIVE_UNARYEXPR
rules:
    if recursion:
        result_type = check_valid_operation(
            unaryexpr(node), recursion(node),
            recursion(op), lineno(unaryexpr)
        )
        term = {
            node: TreeNode(
                unaryexpr(node), recursion(node),
                recursion(op), result_type
            ),
            op: recursion(op),
        }
    else:
        term = {node: unaryexpr_node}


production:
    RECURSIVE_UNARYEXPR : UNARYEXPR_OPERATION TERM
```

```
rules:
    recursive_unaryexpr(node) = term(node),
    recursive_unaryexpr(op) = unaryexp_operation(op)


production:
    UNARYEXPR_OPERATION : OPERATION
rules:
    unaryexpr_operation(op) = operation


production:
    UNARYEXPR : MINUS_OR_PLUS FACTOR
rules:
    if operation(op) == "-":
        factpr(node).value = -factor(node).value
    unaryexpr = factor



production:
    UNARYEXPR : FACTOR
rules:
    unaryexp(node) = factor(node)


production:
    FACTOR : INT_CONSTANT
rules:
        factor = {node: TreeNode(None, None, int_constant, "int")}



production:
    FACTOR : FLOAT_CONSTANT
rules:
    factor = {node: TreeNode(None, None, float_constant, "float")}


production:
    FACTOR : STRING_CONSTANT
```

```
rules:
     factor = {node: TreeNode(None, None, string_constant, "string")}


production:
    FACTOR : NULL
rules:
    factor = {node: TreeNode(None, None, None, "null")}

production:
    FACTOR : LVALUE
rules:
    factor(node) = lvalue(node)


production:
    FACTOR : LPAREN NUMEXPRESSION RPAREN
rules:
    numexpr = p[2]
    numexpr_node = numexpr["node"]
    factor = numexpr
    expressions.append((numexpr_node, p.lineno(1)))


production:
    LVALUE : IDENT OPTIONAL_ALLOC_NUMEXPRESSION
rules:
    res_type = get_variable_type(label, p.lineno(1))
    lvalue = {
        node: TreeNode(
            None,
            None,
            label + optional_alloc_numexpr,
            res_type = get_variable_type(label, label(lineno))
        )
    }
```

4.3 SDT

Foi gerada também uma SDT para a SDD de EXPA, disponível abaixo na seção de código 2:

Código Fonte 2: SDT da gramática EXPA

```
EXPRESSION : NUMEXPRESSION OPT_REL_OP_NUM_EXPR {
    expressions.append((numexpr_node, lineno(numexpression)))
}

OPTIONAL_REL_OP_NUMEXPRESSION : REL_OP NUMEXPRESSION {
    xpressions.append((expression.node, lineno(expression)))
}

NUMEXPRESSION : TERM RECURSIVE_MINUS_OR_PLUS {
    f recursive is None:
        expression = term
    else:
        result_type = check_valid_operation(
            term(node), recursive(node),
            recursive(node), lineno(term)
        )
        expression = {
            node: TreeNode(
                term(node), recursive(node),
                recursive(op), result_type
            )
        }

RECURSIVE_MINUS_OR_PLUS : MINUS_OR_PLUS TERM RECURSIVE_MINUS_OR_PLUS {
    f recursive is None:
        expression = term
    elif expression[4]:
        # more recursions
        result_type = check_valid_operation(
            minus_or_plus_1(node), term(node),
            minus_or_plus_2(node)[operation], lineno(expression)
        )
        recursive_minus_or_plus = {
            "node": TreeNode(
                term(node), minus_or_plus_2(node),
```

```
                minus_or_plus_2(node)[operation], result_type
            ),
            op: term(node)(op),
        }
    else:
        # last recursion
        recursive_minus_or_plus = {node: term(node),
                    op: minus_or_plus_1(op)}

MINUS_OR_PLUS : OPERATION{  minus_or_plus = op }

TERM : UNARYEXPR RECURSIVE_UNARYEXPR {
    f recursion:
        result_type = check_valid_operation(
            unaryexpr(node), recursion(node),
            recursion(op), lineno(unaryexpr)
        )
        term = {
            node: TreeNode(
                unaryexpr(node), recursion(node),
                recursion(op), result_type
            ),
            op: recursion(op),
        }
    else:
        term = {node: unaryexpr_node}

RECURSIVE_UNARYEXPR : UNARYEXPR_OPERATION TERM {
    recursive_unaryexpr(node) = term(node),
    recursive_unaryexpr(op) = unaryexp_operation(op)
}
UNARYEXPR_OPERATION : OPERATION { unaryexpr_operation(op) = operation }

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

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

UNARYEXPR : MINUS_OR_PLUS FACTOR {
    if operation(op) == "-":
        factpr(node).value = -factor(node).value
    unaryexpr = factor
```

```
UNARYEXPR : FACTOR { unaryexp(node) = factor(node) }

FACTOR : INT_CONSTANT {
    factor = {node: TreeNode(None, None, int_constant, "int")}
}

FACTOR : FLOAT_CONSTANT {
    factor = {node: TreeNode(None, None, float_constant, "float")}
}

FACTOR : STRING_CONSTANT {
    factor = {node: TreeNode(None, None, string_constant, "string")}
}

FACTOR : NULL { factor = {
    node: TreeNode(None, None, None, "null")}
}

FACTOR : LVALUE {
    factor(node) = lvalue(node)
}

FACTOR : LPAREN NUMEXPRESSION RPAREN {
    expressions.append( (numexpr(node), numexpr(lineno)) )
}

LVALUE : IDENT OPTIONAL_ALLOC_NUMEXPRESSION {
    lvalue = {
        node: TreeNode(
            None,
            None,
            label + optional_alloc_numexpr,
            res_type = get_variable_type(label, label(lineno))
        )
    }
}
```

4.4 Construção da Árvore de Expressão

A partir da SDD, foi implementada uma função que constrói a árvore de expressão no arquivo *srx/syntax.py*. A árvore é composta pela ID dos nodos, a linha onde eles se encontram e seus respectivos valores, além dos valores dos nodos à esquerda e à direita. Um exemplo de uma árvore de expressão gerada a partir do **program1.lua** fornecido com o projeto encaminhado pode ser visto na seção **Entradas e Saídas**, no trecho de código 21

5 Inserção de tipos na tabela de símbolos (DEC)

Para a inserção de tipos na tabela de símbolos, primeiramente foi construída a gramática **DEC**, com as produções de **CC-2021-1** que produzem declarações de variáveis, como presentado abaixo, na seção construção da árvore de expressão, as produções que geram expressões aritméticas foram separadas da gramática **CC-2021-1** e agrupadas na gramática **EXPA**, descrita abaixo, na seção 5.1.

5.1 Separação da Gramática

$FUNCDEF$	→ def ident (PARAMLIST) STATELIST
$PARAMLIST$	→ DATATYPE ident PARAMLISTTMP
	| &
$PARAMLISTTMP$	→ , PARAMLIST
	| &
$DATATYPE$	→ int
	| float
	| string
$VARDECL$	→ DATATYPE ident OPTIONAL_VECTOR
$OPTIONAL\_VECTOR$	→ [int_constant] OPTIONAL_VECTOR
	| &

5.2 SDD

A partir de **DEC**, foi construída a SDD apresentada abaixo, e a prova de que ela é L-atribuída está descrita na seção 9.8

Código Fonte 3: SDD da gramática DEC

```
production:
    FUNCDEF : FUNCTION_DECLARATION LABEL NEW_SCOPE
              LEFT_PARENTHESIS PARAMLIST RIGHT_PARENTHESIS
              LEFT_BRACKET STATELIST RIGHT_BRACKET
rules:
    scopes.pop()
    current_scope = scopes.seek()
    func_label = p[2]
    func_line_number = p.lineno(2)

    new_func = EntryTable(
        label=funcdef(label), datatype="FUNCTION",
        values=[], lineno=funcdef(lineno)
    )

    if current_scope is not None:
        current_scope.add_entry(new_func)
    pass

production:
    PARAMLIST : DATATYPE LABEL PARAMLISTTMP
rules:
    if len(paramlist) > 2:
        current_scope = scopes.seek()

    paramlist_type = p[1]
    paramlist_label = p[2]
    paramlist_lineno = p.lineno(2)

    paramlist = EntryTable(
        label=paramlist(label),
        datatype=paramlist(datatype),
        values=[],
        lineno=paramlist(lineno),
    )
    if current_scope is not None:
        current_scope.add_entry(paramlist)
```

```
production:
    VARDECL : DATATYPE LABEL OPTIONAL_VECTOR
rules:
    variable = EntryTable(
        label=vardecl(label),
        datatype=vardecl(datatype),
        values=vardecl(optional_vector),
        lineno=vardecl(lineno),
    )

    current_scope = scopes.seek()

    if current_scope is not None:
        current_scope.add_entry(variable)

production:
    OPTIONAL_VECTOR : LEFT_SQUARE_BRACKET INTEGER_CONSTANT
                      RIGHT_SQUARE_BRACKET OPTIONAL_VECTOR
rules:
    if len(optional_vector) > 2:
        optional_vector = [optional_vector(integer_constant),
                           *optional_vector(optional_vector)]
    else:
        optional_vector = []
    pass
```

5.3 SDT

Foi gerada também uma SDT para a SDD de EXPA, disponível abaixo na seção de código 4:

Código Fonte 4: SDT da gramática DEC

```
FUNCDEF : FUNCTION_DECLARATION LABEL NEW_SCOPE
          LEFT_PARENTHESIS PARAMLIST
          RIGHT_PARENTHESIS LEFT_BRACKET
          STATELIST RIGHT_BRACKET {

    scopes.pop()
    current_scope = scopes.seek()
    func_label = p[2]
    func_line_number = p.lineno(2)

    new_func = EntryTable(
        label=funcdef(label), datatype="FUNCTION",
        values=[], lineno=funcdef(lineno)
    )

    if current_scope is not None:
        current_scope.add_entry(new_func)
    pass

}

PARAMLIST : DATATYPE LABEL PARAMLISTTMP {
    if len(paramlist) > 2:
        current_scope = scopes.seek()

    paramlist_type = p[1]
    paramlist_label = p[2]
    paramlist_lineno = p.lineno(2)

    paramlist = EntryTable(
        label=paramlist(label),
        datatype=paramlist(datatype),
        values=[],
        lineno=paramlist(lineno),
    )
    if current_scope is not None:
        current_scope.add_entry(paramlist)
}
```

```
VARDECL : DATATYPE LABEL OPTIONAL_VECTOR {

    variable = EntryTable(
        label=vardecl(label),
        datatype=vardecl(datatype),
        values=vardecl(optional_vector),
        lineno=vardecl(lineno),
    )

    current_scope = scopes.seek()

    if current_scope is not None:
        current_scope.add_entry(variable)
}

OPTIONAL_VECTOR : LEFT_SQUARE_BRACKET INTEGER_CONSTANT
                  RIGHT_SQUARE_BRACKET OPTIONAL_VECTOR {

    if len(optional_vector) > 2:
        optional_vector = [optional_vector(integer_constant),
                           *optional_vector(optional_vector)]
    else:
        optional_vector = []
    pass
}
```

6 Verificação de Tipos

A etapa de verificação de tipos tem o objetivo de validar as expressões de acordo cmo as regras definidas parar operações entre diferentes tipos. Para a implementação deste analisador semântico, foram consideradas as seguintes operações entre tipos:

- *string* e *string*: concatenação (soma);
- *int* e *int*: soma, subtração, multiplicação, divisão e módulo;
- *float* e *float*: soma, subtração, multiplicação e divisão;
- *int* e *float*: soma, subtração, multiplicação e divisão;
- *float* e *int*: soma, subtração, multiplicação e divisisão.

A verificação de tipos foi implementada na função *check_ valid_ operation*, que funciona da seguinte maneira:

Ela recebe dois Nodos da árvore de expressão *TreeNode*, representando os termos à esquerda e à direita da operação. Recebe também a própria operação como *string*, e o número da linha como *int*.

A função também contém um dicionário *valid_ operations*, cujas chaves são as operações (soma, subtração, multiplicação, divisão e módulo) e os valores são, por sua vez uma lista de dicionários contendo os tipos válidos para as operações em questão.

O teste é feito comparando a operação recebida pela função com o dicionário de operações válidas e analizando seu resultado. Caso ele seja *None*, significa que a operação não é válida, ou seja, os tipos dos operandos não são condizentes com as regras da operação. Nesse caso é apresentado um erro contendo a operação em questão.

Um trecho dessa função está descrito no trecho de código 5.

Código Fonte 5: Trecho da Verificação de Tipos

```
    def check_valid_operation(
    left: TreeNode, right: TreeNode, operation: str, lineno: int
) -> str:
    valid_operations = {
        "+": [
            {"left": "int", "right": "int", "result": "int"},
            {"left": "float", "right": "float", "result": "float"},
            {"left": "string", "right": "string", "result": "string"},
```

```
            {"left": "float", "right": "int", "result": "float"},
            {"left": "int", "right": "float", "result": "float"},
        ],
        "*": [
            {"left": "int", "right": "int", "result": "int"},
            {"left": "float", "right": "float", "result": "float"},
            {"left": "float", "right": "int", "result": "float"},
            {"left": "int", "right": "float", "result": "float"},
        ],
        "%": [
            {"left": "int", "right": "int", "result": "int"},
        ],
    }
    op_list = valid_operations.get(operation)
    if op_list is None:
        raise InvalidBinaryOperation(f"invalid operation {operation}")
    result = list(
        filter(
            lambda op: op["left"] == left.res_type == right.res_type,
            op_list,
        )
    )
    return result[0]["result"]
```

7 Verificação de identificadores por escopo

Consiste de tratar as variáveis declaradas dentro de um mesmo escopo. O escopo delimita a vida útil de uma certa variável em um programa, e para evitar a duplicação dos identificadores foi utilizada uma estrutura que guarda o nome utilizado, o tipo primitivo, o array de valores e a linha que foi declarada. Tal estrutura é utilizada dentro da estrutura de escopos que possui uma lista dessas entradas.

Toda vez que uma variável nova é declarada os dados disponíveis são checados para evitar a declaração de duas variáveis com o mesmo nome. O código `test-code/semantic-analysis/valid-variable-declaration.lua` demonstra diversos casos corretos de declaração de variáveis enquanto o arquivo `test-code/semantic-analysis/invalid-variable-declaration.lua` demonstra diversos casos que gerarão erros de compilação.

Código Fonte 6: Trecho de código que revisa as declarações de variáveis, em caso de repetição é levantado um VariableInScopeError que será impresso na tela com informações do erro como a localização do identificador que foi repetido e onde ele foi declarado anteriormente

```
def add_entry(self, entry: EntryTable) -> None:
        has_var, lineno = self.contains_var(entry.label)

        if has_var:
            raise VariableInScopeError(
                f"""Variável {entry.label} na linha {entry.lineno} já
                foi declarada na linha {lineno}"""
            )
        self.entry_table.append(entry)
```

Código Fonte 7: Casos válidos de declaração de variáveis

```
-- Código de demonstração da verificação de declarações de variáveis por escopo

def main() {
    print "OK - declaração de variavel em escopo da função 'main'";
    int a;
    a = 2;
}

def util() {
    int i;
    for (i = 0; i < 2; i = i + 1) {
        print "OK - declaração de variavel com nome igual porém utilizado ";
        print "anteriormente em outro escopo";
        int a;
        print "loop!";
    }

    print "OK - declaração de variavel com nome igual porém utilizado em ";
    print "outro escopo anteriormente";
    int a;
    a = 5;
}
```

Código Fonte 8: Código que possui um caso inválido de declaração de variáveis

```
-- Código de demonstração da verificação de declarações de variáveis por escopo

def main() {
    if (1 < 2) {
        print "OK - declaração de variavel dentro de escopo";
        int i;
        i = 3;
    }
    print "OK - declaração de variavel em escopo global válida mesmo com ";
    print "nome utilizado anteriormente em outro escopo";
    int i;

    -- ///

    print "ERRO - declaração de variável em mesmo escopo com mesmo nome 'i'";
    string i;
    i = "error!";
}
```

8 Comandos dentro de escopos

O compilador verifica o uso correto do operador `break`, tal operador deve operar segundo as regras definidas pela gramática. Pode ser utilizado nas produções que geram um STATEMENT como no corpo de um `if` ou `for`.

Escopos neste trabalho possuem dois tipos: os gerais e os utilizados em comandos de repetição. Um erro de operator `break` inválido (chamado de `InvalidBreakError`) é disparado após percorrer todos os escopos disponíveis e não for encontrado um escopo externo. Tal operação é realizada pelo corpo da produção `BREAK_STATEMENT` que pode ser verificado abaixo.

Código Fonte 9: Corpo da produção BREAK_STATEMENT

```
def p_BREAK_STATEMENT(p: yacc.YaccProduction) -> None:
    """
    BREAK_STATEMENT : BREAK SEMICOLON
    """
    current_scope = scopes.seek()

    while True:
        if current_scope is None or current_scope.loop:
            break
        else:
            current_scope = current_scope.outer_scope

            # If there is no outer scope then it's an error
            if current_scope is None:
                # Get error line number and raise an error
                error_lineno = p.lineno(2)
                raise InvalidBreakError(
                    f"Operador 'break' inválido na linha {error_lineno}"
                )
```

Também foram feitos dois códigos para verificar o correto funcionamento deste módulo, um deles chamado de `test-code/semantic-analysis/valid-break-operator.lua` demonstra o uso correto do operador e deve ser compilado sem problemas. Já o arquivo `test-code/semantic-analysis/invalid-break-operator.lua` apresenta casos inválidos que resultarão em erro de compilação. Estes dois arquivos estão incluídos abaixo:

Código Fonte 10: Código com usos do operator break incorreto

```
-- Exemplos de uso inválido do operador break

def main() {
    -- Descomente qualquer caso abaixo para receber um erro do operador break.
    -- Lembrando que o compilador para a execução após o primeiro
    -- erro encontrado.

    -- Uso inválido na raiz do programa
    break;

    -- Uso inválido no lugar de uma expression
    -- if (break;) {
    --     int i;
    -- }
}
```

Código Fonte 11: Código com exemplos de uso correto do operator break

```
-- Código que demonstra o uso válido do operator 'break'

def main() {
    -- Uso válido dentro de um for
    int i;
    for (i = 0; i < 2; i = i + 1) {
        string str;
        str = "operator break válido!";
        break;
    }
}
```

9 Implementação

9.1 Analisadores de sintaxe e semântica

O Analisador Sintático e suas regras estão descritas no arquivo *src/-parser.py*. Ele é invocado anteriormente ao Analisador Semântico que foi declarado no arquivo *src/syntax.py*.

Foi utilizada a ferramenta *PLY*[1] para a implementação do analisador sintático, especificamente o módulo *ply.yacc*. *Yacc* significa *"Yet Another Compiler Compiler"*, e é uma ferramenta comum para construção de analisadores sintáticos e compiladores, reescrita como uma biblioteca *Python*.

9.2 Símbolo Inicial

A primeira regra criada define a regra inicial da gramática, chamada de *PROGRAM* e demonstrada no trecho de código 12

Código Fonte 12: Função p_PROGRAM

```
def p_PROGRAM(p: LexToken) -> None:
    """
    PROGRAM : STATEMENT
            | FUNCLIST
            | empty
    """
    p[0] = p[1]
```

9.3 Regras da gramática

Em concordância com a documentação da biblioteca *PLY* cada regra da gramática foi implementada como uma função, como mostra o trecho de código 13:

Código Fonte 13: Função p_DATATYPE

```
def p_DATATYPE(p: LexToken) -> None:
    """
    DATATYPE : INTEGER
             | FLOATING_POINT
             | STRING
    """
    p[0] = p[1]
```

As *docstrings* de cada função descrevem a especificação das regras de produção da gramática, e os corpos das funções representam as ações realizadas pelas regras.

Cada função recebe um argumento p, que consiste em uma lista de símbolos e valores da regra correspondente. Utilizando o exemplo do trecho de código 13, a lista p e a declaração da função se dão como segue:

Código Fonte 14: Função p_DATATYPE Exemplificada

```
def p_DATATYPE(p: LexToken) -> None:
    """
    DATATYPE : INTEGER
             | FLOATING_POINT
             | STRING
    """
    #  ^           ^
    # p[0]       p[1]

    p[0] = p[1]
```

9.4 Produções vazias

Definimos uma produção vazia da seguinte maneira:

Código Fonte 15: Função p_empty

```
def p_empty(p: LexToken) -> None:
    "empty :"
     pass
```

E regras de produção que contém a produção vazia utilizam o termo *empty* para denotá-las, como exemplificado:

Código Fonte 16: Regra com produção vazia

```
def p_PARAMLISTCALL(p: LexToken) -> None:
    """
    PARAMLISTCALL : LABEL COMMA PARAMLISTCALL
                  | LABEL
                  | empty
    """
    if len(p) > 2:
        p[0] = (p[1], p[3])
    else:
        p[0] = p[1]
```

Aproveitando o exemplo da função p_PARAMLISTCALL (trecho de código 16), quando uma regra de produção possui quantidade de termos variável, essas situações foram testadas analisando o tamanho da lista de Tokens, através de *len(p)*.

Além disso, quando houve a necessidade de retornar mais de um termo por função, como no caso já mencionado, onde *PARAMLISTCALL* pode produzir *LABEL COMMA PARAMLISTCALL*, foram utilizadas estruturas de dados para retornar os múltiplos termos.

9.5 Declaração de Variáveis

A declaração de variáveis, definida pela regra de prodrução *VARDECL*, declarada no trecho de código 17, foi construída da seguinte forma:

Código Fonte 17: Função p_VARDECL

```
def p_VARDECL(p: yacc.YaccProduction) -> None:
    """
    VARDECL : DATATYPE LABEL OPTIONAL_VECTOR
    """
    variable_type = p[1]
    variable_label = p[2]
    variable_values = p[3]
    variable_lineno = p.lineno(2)
```

```
    # save variable as entry table
    variable = EntryTable(
        label=variable_label,
        datatype=variable_type,
        values=variable_values,
        lineno=variable_lineno,
    )
    # get current scope
    current_scope = scopes.seek()
    # add variable to current scope
    if current_scope is not None:
        current_scope.add_entry(variable)
```

O corpo da produção separa os dados relevantes daquela declaração de variável que serão utilizados para criar uma entrada na tabela de símbolos e após esta etapa o escopo atual é buscado. Se ele existir a variável é adicionada à sua table. Na função `add_entry` ocorre o tratamento de declaração de variáveis repetidas no mesmo escopo conforme foi explicado na seção 7.

9.6 Tratamento de Erros

É importante que o *parser* não interrompa sua execução ao encontrar um erro, pois seria pouco eficiente e prejudicial para o funcionamento da análise sintática. Ao invés disso, foi utilizado um método de tratamento de erros sintáticos para que o *parser* possa notificar o erro, e se possível continuar a execução, dem modo que possíveis outros erros sejam identificados.

Para isso, foi criada a função *p_ error* que recebe um token *p* e imprime a linha e a posição em que o erro ocorreu, como declarado a seguir:

Código Fonte 18: Trecho de syntax.py

```
def p_error(p: yacc.YaccProduction) -> None:
    raise InvalidSyntaxError(f"Syntax error at token {p}")
```

Quando um erro de sintaxe ocorre, a biblioteca *yacc.py* chama a função *p_ error* com o token problemático como argumento.

Os erros estão reunidos no arquivo `output.py`, na entrega 3 foram adicionados novos erros que ocorrem na etapa semântica. Quando ocorre um erro o programa lança uma exceção que é capturada no arquivo `main.py`.

Código Fonte 19: Seção do main.py que trata os erros ocorridos na etapa de análise semântica

```
try:
    syntax_parser = yacc.yacc(
        start="PROGRAM", check_recursion=True, debug=False
    )
    result = syntax_parser.parse(src, debug=False, lexer=lexer)
    print_separator()
    print("Syntax parser result:\n")
    pprint(result)
except Exception as error:
    print(f"Erro na etapa de análise semântica: {error}")
    sys.exit(-1)
print("Análise semântica feita com sucesso! Não houveram erros!")
```

Um erro de variável declarada repetidamente no mesmo escopo é gerado da seguinte maneira:

```
\raise VariableInScopeError(f"Variável {entry.label} na linha
       {entry.lineno} já foi declarada na linha {lineno}")
```

Com o template de string é possivel adicionar informações do contexto em que ocorreu o erro para facilitar na hora de depurar o algoritmo. Por exemplo, ao executar o arquivo `test-code/semantic-analysis/invalid-variable-declaration.lua` a seguinte output é impressa no terminal:

```
Erro na etapa de análise sintática:  Variável i na linha 16 já
foi declarada na linha 11
```

9.7 Saída em casos de sucesso

Em casos onde não existem erros nos códigos enviados ao compilador será impressa na tela uma mensagem de sucesso para cada etapa. Ao fim da etapa de análise semântica também será impressa um dicionário com duas estruturas: 'expressions' representando a árvore de expressões gerada pelo compilador e 'scopes' que representa os escopos identificados pelo programa junto com a sua tabela de símbolos. A tabela de símbolos reune informações relevantes da declaração da variável como a sua localização e o seu tipo.

Em casos de insucesso na compilação serão mostradas mensagens de erro conforme demonstrado na seção 9.6

9.8 SDDs L-Atribuídas

Atributos de uma SDD podem ser de dois tipos: **sintetizados** e **herdados**. Uma SDD é L-atribuída se não possui ciclos de dependência entre produções.

Atributos sintetizados são atributos dos símbolos não terminais à esquerda da produção, e podem tomar valores apenas dos termos à direita da produção. Por exemplo, assumindo uma gramática com a seguinte produção:

$$A \quad \rightarrow \text{BC}$$

O atributo de A é dependente dos atributos de B e C e portanto, é um atributo sintetizado.

O atributo de um símbolo não-terminal à direita de uma produção é um atributo herdado. Eles podem tomar valores tanto dos à esquerda quanto dos à direita das produções. Por exmeplo, na gramática

$$A \quad \rightarrow \text{BC}$$

O atributo de B é dependente tanto de A quanto de C e portanto é um atributo herdado.

Uma SDD é L-atribuída se possui atributos sintetizados e herdados com a restrição de que atributos herdados só podem receber valores de termos à esquerda.

Para garantir que as SDDs criadas sejam L-atribuídas, foi feito com que apenas atributos sintetizados ou herdados das produções à esquerda possam ser utilizados. Dessa forma, atributos herdados possuem uma única direção de fonte e destino. Isso faz com que não sejam criados ciclos de dependencia entre pais e filhos na SDD.

10 Entradas e Saídas

10.1 Programas de validação

Foram feitos diveros programas de exemplos para testar as funcionalidades do trabalho. Existem exemplos feitos para testar códigos com erros e o comportamento do compilador ao rodar eles.

Os códigos com erros são os seguintes: *invalid-break-operator.lua* (que faz uso indevido da palavra-chave *break*), *invalid-operations.lua* (que faz uso operações inválidas entre variáveis) e *invalid-variable-declaration.lua* (que faz declaração de variáveis já declaradas no mesmo escopo). Existem exemplos válidos que são versões dos códigos errados citados anteriormente, porém modificados para funcionar da maneira certa e respeitando a linguagem.

10.2 Programas de exemplo

Foram feitos quatro arquivos com um mínimo de 100 linhas para testar o compilador. O primeiro algoritmo simula um sistema de notas de uma turma de alunos com as notas das provas geradas de acordo com a sua posição. O programa calcula as médias de cada aluno e salva numa lista de notas finais. Também é executada uma função que imprime o nome de cada aluno, suas notas, sua média final e diz se o aluno foi aprovado ou não.

O segundo programa (*program2.lua*) simula um sistema de controle de estoque de produtos genéricos. Cada produto tem nome, descrição e preço. É possível adicionar novos produtos, atualizar os dados de um produto e deletar um produto. Também trata erros para não permitir a adição de um produto quando o estoque estiver cheio e avisa quando o produto solicitado não existe. Ao final do programa a função *main()* é chamada e nela são feitas algumas chamadas das funções, definidas anteriormente. São adicionados 5 produtos diferentes seguido de uma tentativa de adicionar outro produto com o estoque cheio, logo após um produto é apagado e outro é atualizado.

O terceiro programa (*program3.lua*) simula um sistema de reserva de assentos de um cinema/teatro. Nele o usuário pode definir o nome do seu estabelecimento, que será exibido na tela de inicialização, e também o tamanho máximo de filas de assentos. Também pode consultar o status de quantos assentos estão ocupados, quantos estão livres e qual o total de assentos. Foram definidas funções para lidar com a reserva de assentos e também para liberar uma posição.

A função de reservar verifica se a posição escolhida é válida e se está livre, caso esteja então coloca o id do usuário na posição do assento. Caso não esteja livre o programa informa que está ocupado e então faz uma sugestão de assentos vizinhos livres. A função *main* inicializa um cinema novo com 25 assentos (5 filas e 5 assentos) com o nome de: 'Praia de Belas Cinema'. Então, imprime o status do cinema e tenta fazer duas reservas, a segunda reserva é apenas para demonstrar que não é possível reservar um lugar já ocupado.

O código implementado em (*program4.lua*) se trata de uma biblioteca simples de operações matemáticas, nela existem funções definidas: calcular potência entre 2 números, exponenciais com a constante de Euler, valor absoluto, arredondar o input para o menor valor inteiro, arredondar o input para o maior valor inteiro e arredondar o input para o mais próximo inteiro. Também inclui uma função main que roda testes básicos e imprime o resultado esperado e o resultado encontrado. A constante de Euler foi fixada em *2.718* arbitrariamente.

10.3 Output análise semântica

A execução do compilador para um programa `.lua` gera uma saída que pode ser impressa no terminal ou salva em um arquivo `.txt` dependendo do modo de execução, que contem informações sobre as análises semânticas e sintáticas, que são descritas nas seções a seguir.

10.3.1 Árvore de expressão EXPA

As árvores de expressão são geradas de acordo com as regras de produção da gramática definida anteriormente na seção 4.1. Para os programas fornecidos, ela reúne todas as expressões presentes no código, separadas por escopo, ou seja, a árvore de expressão de um dado programa `.lua` pode conter múltiplas sub-árvores, dependendo da quantidade de escopos que esse programa possui. Os nodos da árvore são armazenados da seguinte maneira:

- Identificador único dos nodos
- Número da linha onde o nodo se encontra
- A árvore de valores, composta por:
  - O valor do nodo
  - a sub-árvore do nodo à esquerda
  - a sub-árvore do nodo à direita

As sub-árvores são organizadas de maneira similar, contendo o valor do nodo, e os nodos à esquerda e à direita. Abaixo é apresentado um trecho da árvore de expressão gerada a partir da execução do programa `program1.lua`, fornecido com o trabalho encaminhado.

Código Fonte 20: Árvore de expressão gerada a partir do program1.lua

```
{'Node Id:': '1cdb3cc3-9fce-40da-8d47-3cb897c61ede',
  'lineno': 39,
  'tree': {'left': {'left': None, 'right': None, 'value': 'i'},
           'right': {'left': None, 'right': None, 'value': 1},
           'value': '+'}},
 {'Node Id:': '5d5ea80c-1b04-48db-bd6d-a8940d1d8524',
  'lineno': 46,
  'tree': {'left': {'left': None, 'right': None, 'value': 'i'},
           'right': {'left': None, 'right': None, 'value': 1},
           'value': '+'}},
 {'Node Id:': '792f93ca-010f-4287-a74e-796272390d56',
  'lineno': 47,
  'tree': {'left': {'left': None, 'right': None, 'value': 'j'},
           'right': {'left': None, 'right': None, 'value': 1},
           'value': '+'}},
 {'Node Id:': '57b4c820-cbcb-40f1-8aac-571bcc64835d',
  'lineno': 48,
  'tree': {'left': {'left': None, 'right': None, 'value': 'n_tests'},
           'right': {'left': None, 'right': None, 'value': 6},
           'value': '*'}},
 {'Node Id:': '8f571fe1-67de-4791-a24e-706093addbd9',
  'lineno': 58,
  'tree': {'left': {'left': None, 'right': None, 'value': 'i'},
           'right': {'left': None, 'right': None, 'value': 1},
           'value': '+'}}
```

10.4 Tabela de Símbolos

A tabela de símbolos é gerada para cada programa compilado, com os tipos das variáveis e seus escopos organizados em um dicionário da linguagem `Python`. Ela contém as variáveis do programa compilado, com o tipo (*datatype*), o identificador (*label*), a linha onde se encontra a variável e seus valores (caso seja um vetor). Abaixo está apresentada um trecho da tabela de símbolos gerada através da execução do `program1.lua`

Código Fonte 21: Estrutura EntryTable gerada a partir do program1.lua

```
'table': {'datatype': 'int',
          'label': 'n_students',
          'lineno': 13,
          'values': []},
          {'datatype': 'int',
          'label': 'n_tests',
          'lineno': 16,
          'values': []},
          {'datatype': 'string',
          'label': 'student_name',
          'lineno': 19,
          'values': []},
         {'datatype': 'float',
          'label': 'student_test_grade',
          'lineno': 22,
          'values': []},
         {'datatype': 'int',
          'label': 'birth_year',
          'lineno': 25,
          'values': []}
```

Referências Bibliográficas

Referências

[1] *Ply (Python Lex-Yacc).* URL: https://ply.readthedocs.io/en/latest/ply.html. (Acessado em: 03/07/2021).

[2] *Princeton University LL(1) Parser visualization.* https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1. (Acessado em: 12/09/2021).